

# РЕАЛИЗАЦИЯ ПОЛЕЗНЫХ АЛГОРИТМОВ НА C++



**Дмитро Кедик**

# **Implementing Useful Algorithms in C++**

**Dmytro Kedyk**

# РЕАЛИЗАЦИЯ ПОЛЕЗНЫХ АЛГОРИТМОВ НА C++

**Дмитро Кедик**

Санкт-Петербург  
«БХВ-Петербург»

2024

УДК 004.4'236  
ББК 32.973.26-018  
К33

**Кедик Д.**

К33 Реализация полезных алгоритмов на C++: Пер. с англ. — СПб.: БХВ-Петербург, 2024. — 1024 с.: ил.  
ISBN 978-5-9775-1862-8

Книга с подробным описанием всевозможных алгоритмов, которые принято реализовывать на C++ в силу высоких требований к скорости и наращиванию мощности алгоритмов. Алгоритмы относятся к следующим предметным областям: машинное обучение и нейронные сети, статистика, криптография, оптимизация, перемножение матриц, хеширование, строковые алгоритмы, случайные леса, методы работы с числами, сортировка, кластеризация, графовые алгоритмы и другие темы, касающиеся программной инженерии. Затронуты вопросы командной разработки алгоритмов.

Книгу можно использовать как справочник по алгоритмам для программистов и исследователей и как учебное пособие для студентов соответствующих специальностей. Также будет полезна при подготовке к собеседованиям.

УДК 004.4'236  
ББК 32.973.26-018

**Группа подготовки издания:**

Руководитель проекта	<i>Олег Сивченко</i>
Зав. редакцией	<i>Людмила Гауль</i>
Перевод с английского	<i>Михаила Райтмана</i>
Редактор	<i>Григорий Добин</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Оформление обложки	<i>Зои Канторович</i>

Copyright © 2020 by Dmytro Kedyk  
Translation Copyright © 2024 by BHV. All rights reserved  
Перевод © 2024 BHV. Все права защищены.

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20

ISBN 979-8-60532-530-7 (англ.)  
ISBN 978-5-9775-1862-8 (рус.)

© Dmytro Kedyk, 2020  
© Перевод на русский язык, оформление.  
ООО "БХВ-Петербург", ООО "БХВ", 2024



# Оглавление

<b>Предисловие .....</b>	<b>19</b>
<b>1. С чего всё начинается.....</b>	<b>21</b>
1.1. Введение .....	21
1.2. Каким должен быть алгоритм? .....	21
1.3. Логика принятия решений .....	23
1.4. Доказательство базовой правильности .....	25
1.5. Асимптотическая запись .....	27
1.6. Машинные модели.....	27
1.7. Рандомизированные алгоритмы .....	29
1.8. Измерение эффективности.....	29
1.9. Типы данных .....	30
1.10. Эксперименты с алгоритмами .....	32
1.11. Управление памятью .....	33
1.12. Оптимизация кода.....	34
1.13. Рекурсия .....	36
1.14. Стратегии вычислений .....	37
1.15. Выбор среди нескольких алгоритмов .....	37
1.16. Создание параллельных алгоритмов.....	38
1.17. Реализация алгоритмов .....	39
1.18. Рекомендуемые курсы для студентов, изучающих информатику .....	40
1.19. Некоторые стратегии обучения .....	42
1.20. О проектах всей книги.....	43
1.21. Список рекомендуемой литературы.....	44
<b>2. Основы разработки программного обеспечения .....</b>	<b>45</b>
2.1. Введение .....	45
2.2. Обзор цикла разработки.....	45
2.3. Требования .....	46
2.4. Проектирование структуры компонентов .....	47
2.5. Проектирование отдельных компонентов и написание кода .....	47
2.6. Шаблоны .....	48
2.7. Управление ошибками .....	52
2.8. Тестирование.....	54
2.9. Просмотр кода .....	56
2.10. Релиз .....	56
2.11. Обслуживание .....	57
2.12. Оценка .....	58
2.13. Следование формальному процессу .....	58

2.14. Управление данными пользователя .....	59
2.15. Список рекомендуемой литературы.....	59
<b>3. Советы по вопросам карьерного роста и прохождения собеседований .....</b>	<b>60</b>
3.1. Введение .....	60
3.2. Отклик на вакансию .....	60
3.3. Составление резюме .....	60
3.4. Поведенческие вопросы и собеседования .....	61
3.5. Технические собеседования.....	63
3.6. Более сложные вопросы.....	67
3.7. Собеседование по проектированию систем .....	68
3.8. Обсуждение предложения.....	68
3.9. Как красиво уйти с текущей работы?.....	69
3.10. Боритесь с самуспокоенностью .....	69
3.11. Советы по дополнительной подготовке.....	70
3.12. Список рекомендуемой литературы.....	70
<b>4. Основы компьютерного права .....</b>	<b>71</b>
4.1. Введение .....	71
4.2. Интеллектуальная собственность.....	71
4.3. Патенты .....	72
4.4. Коммерческие тайны .....	73
4.5. Авторские права.....	74
4.6. Товарные знаки.....	75
4.7. Управление интеллектуальной собственностью .....	76
4.8. Контракты .....	76
4.9. Лицензии .....	77
4.10. Трудовые соглашения .....	78
4.11. Конфиденциальность.....	79
4.12. Киберпреступления .....	80
4.13. Выступления в качестве свидетеля-эксперта .....	80
4.14. Советы по дополнительной подготовке.....	81
4.15. Список рекомендуемой литературы.....	81
<b>5. Фундаментальные структуры данных .....</b>	<b>82</b>
5.1. Введение .....	82
5.2. Вспомогательные функции.....	82
5.3. Вектор.....	88
5.4. Блочный массив .....	92
5.5. Связанный список.....	93
5.6. Свободный список со сборкой мусора .....	97
5.7. Стек.....	101
5.8. Очередь.....	102
5.9. Деревья .....	105
5.10. Битовые алгоритмы .....	106
5.11. Набор битов.....	109
5.12. Поиск объединения.....	115
5.13. Примечания по реализации.....	116
5.14. Комментарии.....	116

5.15. Советы по дополнительной подготовке.....	117
5.16. Список рекомендуемой литературы.....	117
<b>6. Генерация случайных чисел .....</b>	<b>118</b>
6.1. Введение .....	118
6.2. Краткий обзор теории вероятностей .....	118
6.3. Генерация псевдослучайных чисел .....	120
6.4. Класс генераторов псевдослучайных чисел Xorshift .....	122
6.5. Вихрь Мерсенна.....	123
6.6. Генератор MRG32k3a.....	123
6.7. Алгоритм RC4.....	125
6.8. Выбор генератора .....	126
6.9. Использование генератора.....	127
6.10. Создание выборок из распределений.....	127
6.11. Генерация выборок из дискретных распределений с ограниченным диапазоном.....	129
6.12. Генерация выборок из особых распределений.....	133
6.13. Генерация случайных объектов.....	137
6.14. Генерация выборок из многомерных распределений.....	139
6.15. Метод Монте-Карло .....	140
6.16. Примечания по реализации.....	146
6.17. Комментарии.....	147
6.18. Советы по дополнительной подготовке.....	147
6.19. Список рекомендуемой литературы.....	148
<b>7. Сортировка.....</b>	<b>149</b>
7.1. Введение .....	149
7.2. Сортировка вставками.....	149
7.3. Быстрая сортировка.....	150
7.4. Сортировка слиянием.....	153
7.5. Целочисленная сортировка.....	154
7.6. Векторная сортировка .....	155
7.7. Сортировка перестановкой.....	158
7.8. Выбор.....	159
7.9. Множественный выбор .....	160
7.10. Поиск .....	161
7.11. Примечания по реализации.....	161
7.12. Комментарии.....	162
7.13. Советы по дополнительной подготовке.....	163
7.14. Список рекомендуемой литературы.....	163
<b>8. Динамически сортируемые последовательности .....</b>	<b>164</b>
8.1. Введение .....	164
8.2. Требования .....	164
8.3. Список с пропусками.....	165
8.4. Декартово дерево .....	170
8.5. Итераторы деревьев.....	176
8.6. Дополнения и варианты API.....	178
8.7. Векторные ключи .....	179
8.8. Расширение LCP для деревьев .....	179

8.9. Префиксное дерево.....	185
8.10. Тернарное декартово дерево.....	186
8.11. Сравнение производительности .....	191
8.12. Примечания по реализации.....	192
8.13. Комментарии.....	192
8.14. Советы по дополнительной подготовке.....	194
8.15. Список рекомендуемой литературы.....	194
<b>9. Хеширование.....</b>	<b>196</b>
9.1. Введение .....	196
9.2. Хеш-функции .....	196
9.3. Универсальные хеш-функции.....	200
9.4. Неуниверсальные хеш-функции .....	203
9.5. Скользящие хеш-функции.....	205
9.6. Коллекция хеш-функций.....	206
9.7. Хеш-таблицы.....	206
9.8. Цепочка хеш-таблиц.....	206
9.9. Хеш-таблица с линейным зондированием.....	211
9.10. Оценка времени .....	215
9.11. Фильтр Блума.....	216
9.12. Примечания по реализации.....	217
9.13. Комментарии.....	218
9.14. Советы по дополнительной подготовке.....	219
9.15. Список рекомендуемой литературы.....	219
<b>10. Приоритетные очереди .....</b>	<b>221</b>
10.1. Введение .....	221
10.2. API.....	221
10.3. Бинарная куча .....	221
10.4. Индексированные кучи .....	224
10.5. Примечания по реализации.....	229
10.6. Комментарии.....	229
10.7. Список рекомендуемой литературы.....	230
<b>11. Алгоритмы графов.....</b>	<b>231</b>
11.1. Введение .....	231
11.2. Основы.....	231
11.3. Представление графов.....	232
11.4. Поиск .....	235
11.5. Примеры задач поиска .....	237
11.6. Минимальное остовное дерево.....	239
11.7. Кратчайшие пути .....	240
11.8. Алгоритмы потока .....	243
11.9. Двудольное сопоставление .....	247
11.10. Устойчивое сопоставление .....	248
11.11. Задача назначения.....	249
11.12. Генерация случайных графов .....	250
11.13. Примечания по реализации.....	251
11.14. Комментарии.....	251

11.15. Советы по дополнительной подготовке.....	252
11.16. Список рекомендуемой литературы.....	252
<b>12. Разные алгоритмы и методы .....</b>	<b>253</b>
12.1. Введение .....	253
12.2. Превращение статических структур данных в динамические .....	253
12.3. Обеспечение устойчивости структур данных .....	254
12.4. Хранение кеша .....	255
12.5. Вектор с $k$ -битным размером слова .....	258
12.6. Объединение множества на интервалах .....	259
12.7. Создание первых $N$ простых чисел .....	259
12.8. Генерация всех возможных перестановок.....	260
12.9. Генерация всех возможных сочетаний .....	261
12.10. Генерация всех подмножеств .....	262
12.11. Создание всех разделов.....	262
12.12. Генерация всех зависимых объектов .....	263
12.13. Примечания по реализации.....	263
12.14. Комментарии.....	263
12.15. Советы по дополнительной подготовке.....	264
12.16. Список рекомендуемой литературы.....	264
<b>13. Алгоритмы для работы с внешней памятью.....</b>	<b>265</b>
13.1. Введение .....	265
13.2. Диски и файлы .....	265
13.3. Структура файла .....	268
13.4. Работа с файлами CSV .....	269
13.5. Модель ввода/вывода .....	273
13.6. Вектор внешней памяти .....	277
13.7. Сортировка.....	280
13.8. Векторные структуры данных .....	282
13.9. Дерево $B^+$ .....	283
13.10. Комментарии.....	291
13.11. Советы по дополнительной подготовке.....	291
13.12. Список рекомендуемой литературы.....	292
<b>14. Строковые алгоритмы .....</b>	<b>293</b>
14.1. Введение .....	293
14.2. Поиск по одному шаблону .....	293
14.3. Поиск по нескольким шаблонам .....	296
14.4. Регулярные выражения .....	297
14.5. Расширенные шаблоны .....	299
14.6. Алгоритмы поиска расстояния между строками .....	301
14.7. Обратный индекс .....	305
14.8. Суффиксный индекс .....	306
14.9. Синтаксическое дерево .....	310
14.10. Введение в краткие структуры данных.....	310
14.11. Примечания по реализации.....	314
14.12. Комментарии.....	315

14.13. Советы по дополнительной подготовке.....	315
14.14. Список рекомендуемой литературы.....	316
<b>15. Сжатие .....</b>	<b>317</b>
15.1. Введение .....	317
15.2. Основные ограничения .....	317
15.3. Энтропия .....	317
15.4. Битовый поток .....	318
15.5. Коды.....	320
15.6. Статические коды .....	320
15.7. Коды Хаффмана.....	324
15.8. Сжатие словаря .....	328
15.9. Кодирование серий байтов .....	331
15.10. Перемещение на передний план.....	332
15.11. Преобразование Берроуза — Уилера.....	333
15.12. Примечания по реализации.....	336
15.13. Комментарии.....	336
15.14. Советы по дополнительной подготовке.....	337
15.15. Список рекомендуемой литературы.....	337
<b>16. Комбинаторная оптимизация .....</b>	<b>338</b>
16.1. Введение .....	338
16.2. Теория сложности.....	338
16.3. Типичные сложные задачи .....	340
16.4. Алгоритмы приближения.....	344
16.5. Эвристика жадного построения.....	345
16.6. Ветвление и границы.....	346
16.7. Поиск крачайшего пути в пространстве с нижними границами.....	352
16.8. Локальный поиск .....	360
16.9. Применение локального поиска к некоторым задачам .....	366
16.10. Алгоритм имитации отжига.....	370
16.11. Повторный локальный поиск.....	375
16.12. Генетические алгоритмы.....	377
16.13. Анализ производительности .....	383
16.14. Подготовка данных для конкретной задачи .....	384
16.15. Многоцелевая оптимизация.....	384
16.16. Обработка ограничений .....	385
16.17. Стохастические задачи.....	390
16.18. Общие рекомендации .....	390
16.19. Примечания по реализации.....	391
16.20. Комментарии.....	391
16.21. Советы по дополнительной подготовке.....	394
16.22. Список рекомендуемой литературы.....	395
<b>17. Большие числа .....</b>	<b>397</b>
17.1. Введение .....	397
17.2. Представление.....	397
17.3. Сложение и вычитание.....	399
17.4. Операции сдвига .....	400

17.5. Умножение .....	401
17.6. Деление .....	402
17.7. Преобразование в десятичное число .....	404
17.8. Возведение в степень .....	404
17.9. Вычисление логарифма .....	405
17.10. Целочисленный квадратный корень .....	405
17.11. Наибольший общий делитель .....	406
17.12. Модульная инверсия .....	406
17.13. Проверка числа на простоту .....	407
17.14. Рациональные числа .....	408
17.15. Примечания по реализации .....	410
17.16. Комментарии .....	410
17.17. Советы по дополнительной подготовке .....	411
17.18. Список рекомендуемой литературы .....	411
<b>18. Вычислительная геометрия .....</b>	<b>412</b>
18.1. Введение .....	412
18.2. Расстояния .....	412
18.3. VP-дерево .....	413
18.4. k-d-дерево .....	417
18.5. Решение задач при большом числе измерений .....	422
18.6. Структуры данных для геометрических объектов .....	423
18.7. Точки .....	423
18.8. Геометрические примитивы .....	424
18.9. Выпуклая оболочка .....	425
18.10. Развертка плоскости .....	426
18.11. Комментарии .....	427
18.12. Советы по дополнительной подготовке .....	428
18.13. Список рекомендуемой литературы .....	428
<b>19. Обнаружение и исправление ошибок .....</b>	<b>429</b>
19.1. Введение .....	429
19.2. Бинарные полиномы .....	429
19.3. Многочлены над конечными полями .....	429
19.4. Обнаружение ошибок .....	430
19.5. Каналы и коды .....	432
19.6. Вычисление конечного поля .....	434
19.7. Полиномы над элементами поля Галуа .....	435
19.8. Коды Рида — Соломона .....	438
19.9. Границы кодов минимального расстояния с фиксированным алфавитом .....	442
19.10. Булевы матрицы .....	443
19.11. Коды проверки на четность с низкой плотностью .....	444
19.12. Примечания по реализации .....	449
19.13. Комментарии .....	450
19.14. Советы по дополнительной подготовке .....	451
19.15. Список рекомендуемой литературы .....	451
<b>20. Криптография .....</b>	<b>452</b>
20.1. Введение .....	452
20.2. Шифрование файлов .....	452

20.3. Длина ключа.....	454
20.4. Хранилище ключей.....	455
20.5. Криптографическое хеширование.....	456
20.6. Обмен ключами .....	456
20.7. Другие протоколы .....	456
20.8. Примечания по реализации.....	457
20.9. Комментарии.....	457
20.10. Совет по дополнительной подготовке .....	458
20.11. Список рекомендуемой литературы.....	458
<b>21. Вычислительная статистика .....</b>	<b>459</b>
21.1. Введение .....	459
21.2. Оценки .....	459
21.3. Оценщики.....	462
21.4. Поиск наиболее эффективных оценщиков .....	469
21.5. Некоторые особенности асимптотики .....	473
21.6. Оценка нормальной CDF .....	473
21.7. Оценка CDF T-распределения .....	474
21.8. Подробнее о доверительных интервалах.....	475
21.9. Границы среднего значения в конечной выборке.....	479
21.10. Доверительные интервалы для общих метрик местоположения.....	481
21.11. Выпадающие значения и надежные выводы .....	484
21.12. Функции оценок.....	486
21.13. Измерение времени выполнения алгоритма.....	487
21.14. Корреляционный анализ .....	488
21.15. Начальная загрузка .....	490
21.16. Когда использовать начальную загрузку? .....	504
21.17. Проверка гипотез.....	505
21.18. Сравнение тестов .....	507
21.19. Эффективное использование тестов .....	509
21.20. Валидация исследований .....	514
21.21. Сравнение совпадающих пар.....	515
21.22. Множественные сравнения.....	517
21.23. Сравнение совпадающих кортежей.....	519
21.24. Сравнение независимых выборок .....	520
21.25. Перестановочные тесты .....	522
21.26. Сравнение нескольких альтернатив в нескольких предметных областях .....	526
21.27. Работа с данными расчета.....	529
21.28. Тестирование различий в распределении .....	531
21.29. Сравнение данных с распределением .....	532
21.30. Сравнение распределений двух выборок.....	534
21.31. Анализ чувствительности.....	535
21.32. Последовательность Соболя.....	537
21.33. Планирование экспериментов: основные идеи.....	541
21.34. Марковская цепь Монте-Карло .....	544
21.35. Байесовские методы .....	547
21.36. Поиск лучшей альтернативы с помощью моделирования .....	549
21.37. Расчет размера выборки.....	552
21.38. Анализ временных рядов .....	553



21.39. Использование статистики на практике.....	565
21.40. Анализ решений.....	567
21.41. Примечания по реализации.....	569
21.42. Комментарии.....	570
21.43. Советы по дополнительной подготовке.....	576
21.44. Список рекомендуемой литературы.....	578

## **22. Численные алгоритмы: введение и матричная алгебра..... 582**

22.1. Введение .....	582
22.2. Арифметика с плавающей точкой .....	583
22.3. Ошибки при использовании арифметики с плавающей точкой .....	584
22.4. Ошибка приближения.....	588
22.5. Показатели устойчивости и состояния .....	589
22.6. Ошибка оптимизации .....	591
22.7. Другие общие темы .....	593
22.8. Разработка надежного численного программного обеспечения .....	595
22.9. Матричная алгебра .....	599
22.10. Матричные нормы .....	602
22.11. LUP-разложение .....	603
22.12. Разложение Холецкого.....	608
22.13. Ленточные матрицы .....	609
22.14. Решение тридиагональных матричных уравнений.....	611
22.15. Ортогональные преобразования.....	611
22.16. QR-разложение .....	613
22.17. Собственные значения и собственные векторы симметричной матрицы .....	615
22.18. Разложение по сингулярным значениям (SVD).....	618
22.19. Собственные значения и собственные векторы асимметричной матрицы.....	622
22.20. Разреженные матрицы.....	627
22.21. Итерационные методы для разреженных матриц.....	633
22.22. Итерационные методы для собственных значений .....	634
22.23. Введение в интервальную арифметику.....	636
22.24. Примечания по реализации.....	639
22.25. Комментарии.....	639
22.26. Советы по дополнительной подготовке.....	642
22.27. Список рекомендуемой литературы.....	642

## **23. Численные алгоритмы: работа с функциями ..... 645**

23.1. Введение .....	645
23.2. Быстрое преобразование Фурье .....	645
23.3. Интерполяция: общие идеи.....	649
23.4. Полиномиальная интерполяция из существующих данных.....	652
23.5. Полиномы Чебышева .....	655
23.6. Кусочная интерполяция .....	662
23.7. Сплаины — для работы с существующими данными .....	668
23.8. Сравнение методов интерполяции .....	672
23.9. Интегрирование .....	674
23.10. Многомерное интегрирование.....	681
23.11. Оценка функции.....	685
23.12. Оценка производных .....	685

23.13. Решение нелинейных уравнений и систем .....	691
23.14. Поиск всех корней в одномерном случае .....	707
23.15. Обыкновенные дифференциальные уравнения.....	708
23.16. Решение жестких ОДУ .....	713
23.17. Краевые задачи для ОДУ .....	717
23.18. Уравнения с частными производными: некоторые размышления .....	719
23.19. Некоторые выводы .....	720
23.20. Примечания по реализации.....	721
23.21. Комментарии.....	721
23.22. Советы по дополнительной подготовке.....	727
23.23. Список рекомендуемой литературы.....	728
<b>24. Численная оптимизация .....</b>	<b>731</b>
24.1. Введение.....	731
24.2. Некоторые общие идеи .....	731
24.3. Минимизация унимодальной функции с одной переменной .....	732
24.4. Минимизация многомерных функций: введение и координатный спуск.....	735
24.5. Линейный поиск .....	738
24.6. Алгоритмы линейного поиска .....	742
24.7. Методы, не требующие вычисления производных.....	748
24.8. Негладкая минимизация.....	753
24.9. Глобальная минимизация.....	756
24.10. Оптимизация в дискретном множестве .....	770
24.11. Стохастическая оптимизация .....	772
24.12. Алгоритмы стохастической аппроксимации .....	773
24.13. Линейное программирование .....	776
24.14. Некоторые соображения о нелинейном программировании .....	779
24.15. Примечания по реализации.....	780
24.16. Комментарии.....	781
24.17. Советы по дополнительной подготовке.....	786
24.18. Список рекомендуемой литературы.....	787
<b>25. Введение в машинное обучение.....</b>	<b>790</b>
25.1. Введение .....	790
25.2. Что такое машинное обучение?.....	790
25.3. Математическое обучение .....	791
25.4. Прогнозирование и структурный вывод.....	796
25.5. Оценка риска предиктора.....	796
25.6. Источники риска.....	799
25.7. Контроль сложности.....	800
25.8. Ошибка аппроксимации .....	802
25.9. Устойчивость .....	805
25.10. Оценка риска стратегии обучения.....	806
25.11. Принятие решений и выбор модели.....	810
25.12. Общие стратегии выбора модели .....	812
25.13. Смещение, дисперсия и бэггинг .....	813
25.14. Паттерны проектирования .....	816
25.15. Подготовка данных.....	817
25.16. Масштабирование.....	819

25.17. Обработка пропущенных значений.....	821
25.18. Выбор признаков .....	821
25.19. Ядра.....	827
25.20. Обучение в реальном времени.....	829
25.21. Работа с не векторными данными .....	830
25.22. Масштабное обучение .....	830
25.23. Выводы .....	832
25.24. Примечания по реализации.....	833
25.25. Комментарии.....	833
25.26. Советы по дополнительной подготовке.....	839
25.27. Список рекомендуемой литературы.....	840

## **26. Машинное обучение: классификация ..... 842**

26.1. Введение.....	842
26.2. Стратификация по метке класса .....	842
26.3. Оценка риска .....	844
26.4. Сведение мультикласса к двум классам .....	848
26.5. Контроль сложности.....	851
26.6. Наивный классификатор Байеса.....	853
26.7. Ближайший сосед .....	856
26.8. Дерево решений .....	859
26.9. Механизм опорных векторов.....	866
26.10. Линейный SVM.....	868
26.11. Ядро SVM.....	873
26.12. Мультиклассовый нелинейный SVM .....	878
26.13. Нейронная сеть .....	880
26.14. Ансамбли рандомизации.....	889
26.15. Обучение в реальном времени.....	891
26.16. Бустинг .....	894
26.17. Масштабное обучение .....	898
26.18. Экономичное обучение .....	898
26.19. Несбалансированное обучение .....	903
26.20. Выбор признаков .....	906
26.21. Сравнение классификаторов.....	906
26.22. Примечания по реализации.....	909
26.23. Комментарии.....	910
26.24. Советы по дополнительной подготовке.....	919
26.25. Список рекомендуемой литературы.....	919

## **27. Машинное обучение: регрессия ..... 922**

27.1. Введение.....	922
27.2. Оценка риска.....	922
27.3. Управление сложностью.....	924
27.4. Линейная регрессия .....	925
27.5. Регрессия лассо.....	926
27.6. Регрессия ближайших соседей .....	930
27.7. Дерево регрессии.....	931
27.8. Регрессия случайного леса.....	934
27.9. Нейронная сеть .....	935

27.10. Выбор признаков .....	936
27.11. Сравнение производительности .....	936
27.12. Примечания по реализации.....	937
27.13. Комментарии.....	938
27.14. Советы по дополнительной подготовке.....	941
27.15. Список рекомендуемой литературы.....	942
<b>28. Машинное обучение: кластеризация .....</b>	<b>943</b>
28.1. Введение .....	943
28.2. Постановка .....	943
28.3. Внешняя оценка .....	945
28.4. Внутренняя оценка и выбор количества кластеров .....	948
28.5. Расчет устойчивости.....	951
28.6. Кластеризация в евклидовом пространстве.....	953
28.7. Кластеризация в метрическом пространстве .....	957
28.8. Спектральная кластеризация .....	960
28.9. Эксперименты.....	964
28.10. Примечания по реализации.....	964
28.11. Комментарии.....	965
28.12. Советы по дополнительной подготовке.....	969
28.13. Список рекомендуемой литературы.....	969
<b>29. Машинное обучение: прочие задачи .....</b>	<b>971</b>
29.1. Введение.....	971
29.2. Обучение с подкреплением.....	971
29.3. Функция, присваивающая значения.....	972
29.4. Поиск часто встречающихся комбинаций предметов .....	974
29.5. Полуконтролируемое обучение.....	975
29.6. Оценка плотности .....	976
29.7. Обнаружение выпадающих значений .....	976
29.8. Примечания по реализации.....	977
29.9. Комментарии.....	977
29.10. Список рекомендуемой литературы.....	978
<b>30. Отстойник: не слишком полезные алгоритмы и структуры данных .....</b>	<b>979</b>
30.1. Введение .....	979
30.2. Сортировка связанного списка .....	979
30.3. Частичная сортировка .....	981
30.4. Сжатое префиксное дерево .....	981
30.5. Хеширование кукушкой .....	987
30.6. Немного приоритетных очередей .....	990
30.7. Младший общий предок (LCA) и запрос с минимальным диапазоном (RQM) .....	996
30.8. Знаковый ранговый критерий Уилкоксона для двух выборок.....	997
30.9. Критерий Фридмана для согласованных выборок.....	998
30.10. MADS-подобный алгоритм оптимизации .....	999
30.11. Алгоритм классификации бустинга SAMME .....	1000
30.12. Бустинг в задаче регрессии.....	1001
30.13. Вероятностная кластеризация .....	1003
30.14. Иерархическая кластеризация .....	1007

30.15. Кластеризация на основе плотности .....	1010
30.16. Не представленные реализации .....	1012
30.17. Советы по дополнительной подготовке .....	1013
30.18. Список рекомендуемой литературы .....	1014
<b>31. Приложение: примечания о языке C++ .....</b>	<b>1015</b>
31.1. Введение .....	1015
31.2. Путеводитель по литературе, посвященной C++ .....	1015
31.3. Советы по дополнительной подготовке .....	1015
31.4. Список рекомендуемой литературы .....	1016
<b>Предметный указатель .....</b>	<b>1017</b>

# Предисловие

Разумеется, в этой книге ошибок нет, но вы все же сообщайте автору о тех ошибках, которые найдете.  
*Форман Эктон*

Из этой книги вы узнаете, как правильно реализовать алгоритмы и структуры данных, достойные реализации (а есть и такие, которые этого не достойны!). Предполагается, что вы уже знакомы с основными понятиями на уровне вводного курса алгоритмов. В этой книге:

- ◆ рассмотрено множество специализированных тем;
- ◆ по более общим, стандартным темам приведен дополнительный материал, необходимый для их полного понимания. Например: «Как реализовать приоритетную очередь с отслеживанием элементов для алгоритмов поиска кратчайшего пути?»;
- ◆ предлагаются однозначные решения, какие алгоритмы и структуры данных лучше в той или иной ситуации использовать. В некоторых завершающих книгу главах именно эта информация, возможно, станет для читателя самой полезной. Но понимать предпосылки тех или иных решений все же важнее.

Иногда известные алгоритмы оказывались недостаточно хороши, поэтому я придумывал свои собственные. Вы не сможете найти их в каком-либо другом источнике. Вот некоторые из них:

- ◆ свободный список со сборкой мусора (см. главу 5. *Фундаментальные структуры данных*);
- ◆ куча сумм (см. главу 6. *Генерация случайных чисел*);
- ◆ реализация универсальной хеш-функции для массивов на основе исключаящего ИЛИ со сдвигом (см. главу 9. *Хеширование*);
- ◆ использование проверки знака для принятия решений и сокращения дерева регрессии (см. главу 26. *Машинное обучение: классификация*);
- ◆ эффективная реализация k-метода (см. главу 28. *Машинное обучение: кластеризация*).

В идеальном мире каждый ключевой шаг в создании алгоритма должен быть общеизвестен, тщательно исследован и проверен. Мои находки тоже нуждаются в такой проверке, хотя, надеюсь, по крайней мере некоторые из них обладают реальной научной ценностью. Если вы занимаетесь исследованиями, не стесняйтесь направлять любые вопросы мне на электронную почту.

В качестве строительных блоков для более сложных алгоритмов и структур данных я использую решения, описанные в книге, а не взятые из стандартной библиотеки шаблонов C++ STL. Это дает возможность лучше тестировать код и предоставляет опреде-

ленные преимущества с точки зрения удобства работы. Например, в моей собственной реализации вектора, которая позволяет удобно работать с последним элементом, я выловил бесчисленное множество ошибок, связанных с проверками границ в `operator[]`. Разумеется, для реальной разработки предпочтительнее использовать устоявшиеся библиотеки, поскольку они стандартны и более надежны. Мои примеры тоже неплохо проверены, но могут служить лишь отправной точкой для хорошей реализации.

Для простоты я не задействую в книге многие новейшие функции C++. Например, семантика перемещения позволяет повысить эффективность кода, если у объектов есть эффективные деструкторы, но в остальных случаях достаточно встроенной оптимизации от компилятора. Тем не менее в книге эти процессы описаны достаточно углубленно, поэтому для ее освоения необходимы некоторые знания в области вычислительной техники и математики. В главах, посвященных специализированным темам типа численных алгоритмов, вам потребуется знакомство с определенными разделами математики — например, с линейной алгеброй. Некоторые более поздние главы опираются на более ранние. Упражнений в книге нет, но в ряде глав представлены проекты, которые вы можете развить. Обычно они требуют исследования и тщательной реализации, до которой я, как правило, не доходил.

Эту книгу лучше всего использовать в качестве дополнения к книгам, в которых рассматриваются базовые понятия (особенно это касается последних глав). Я не утверждаю, что мои реализации идеальны. Они, скорее, представляют собой первый шаг в правильном направлении. Я внимательно отнесся к составлению списка литературы, и если вы не найдете в нем некоторых популярных книг по той или иной теме, то это, скорее всего, связано с тем, что я подошел к нему весьма избирательно. В книгах других авторов, как правило, приводится больше примеров и математики, но гораздо меньше деталей реализации, и лучше познакомиться с несколькими разными изданиями, чтобы лучше усвоить материал.

Предыдущие издания этой книги назывались «Обобщенные алгоритмы и структуры данных в C++», и я обновлял их каждые два года. Немалого труда стоило исправить все ошибки и учесть все отзывы, чтобы дойти до «стабильной версии». Поэтому в ближайшие несколько лет нового издания не будет, хотя мой интерес ко многим затронутым темам еще велик. Пожалуйста, присылайте любые отзывы на мою электронную почту [igmdk@msn.com](mailto:igmdk@msn.com) или разместите отзыв на Amazon — мне страшно интересно узнать, что вам понравилось, а что нужно дополнить или улучшить.

Приготовьтесь узнавать что-то новое в каждой главе!

Все примеры кода и некоторые иллюстрации можно скачать здесь:

**<https://github.com/dkedyk/ImplementingUsefulAlgorithms>.**

# 1. С чего всё начинается...

Желаю, чтобы мое желание не исполнилось.  
*Дуглас Хофштадтер*

Если вы не знаете, как что-то сделать, то на компьютере вы это тоже не сделаете.  
*Неизвестный автор*

## 1.1. Введение

Нетривиальные алгоритмы приходится реализовывать редко, так как они бывают уже доступны в тех или иных API. В программировании большая часть усилий уходит на логическое моделирование системы, выполняемое экспертами в предметной области. Но иногда важно понимать, что делает API в принципе, и иметь возможность изменить или расширить имеющуюся реализацию. Кроме того, кто-то должен разрабатывать и поддерживать API, а для различных встроенных устройств это делается обычно либо с нуля, либо путем адаптации имеющихся аналогичных API.

В этой книге будут рассмотрены — за некоторыми исключениями — только те алгоритмы, которые предназначены для реального применения. Теоретические алгоритмы, существующие, по всей видимости, только благодаря любопытству исследователей, хорошо изложены в книге [1.6]<sup>1</sup>.

В этой главе рассматриваются некоторые темы, к которым нам предстоит неоднократно вернуться.

## 1.2. Каким должен быть алгоритм?

*Алгоритм* — это четко определенное преобразование входных данных в выходные. Эффективный доступ и обновление сохраненных данных обеспечивает *структура данных*. Для поддержания успешной разработки программного обеспечения и алгоритм, и структура данных должны быть:

- ♦ *корректными* — то есть давать удовлетворительное решение поставленной задачи. Чтобы признать алгоритм и его реализацию правильными, нужны формальные доказательства, всестороннее тестирование или интуиция. В корректности алгоритма и структуры данных никогда нельзя быть уверенным на сто процентов, потому что формальные доказательства не всегда можно получить, интуиция может ошибаться, а логика, к которой редко обращаются, трудно проверяется. Корректность может опираться и на вероятность событий, поэтому алгоритм, который ошибается реже,

---

<sup>1</sup> Список рекомендуемой литературы приводится в конце каждой главы.



чем в ваш компьютер бьет молния, можно считать правильным. Алгоритм должен сообщать об ошибках — например, о невозможности выделения памяти;

- ◆ *расширяемыми* — то есть применимыми к общим проблемам при внесении небольших изменений. В реальных задачах иногда присутствуют дополнительные ограничения, из-за которых требуется доработка существующих алгоритмов. Алгоритм, применимый к некоторому семейству задач, более ценен, чем набор более эффективных специальных алгоритмов для решения каждой конкретной задачи. Реализация считается расширяемой, если ее код можно без изменений применять ко множеству задач. Всегда отдавайте предпочтение коду, допускающему многократное использование, даже если придется в разумных пределах потерять в простоте и эффективности. В то же время использование структуры частной задачи почти всегда позволяет найти лучшее решение, поэтому и специфические алгоритмы имеют право на существование;
- ◆ *относительно простыми* для понимания, реализации в различных средах и использования в качестве абстрактного строительного блока. Простота *понимания* измеряется временем, требующимся для того, чтобы вникнуть в рутинную работу алгоритма и ее отладить, простота *реализации* — количеством строк правильно структурированного кода, а простота *использования* — чистотой API с точки зрения количества функций, их параметров и наличия хороших значений по умолчанию. Для простоты понимания важно наличие полезной информации об алгоритме, включая теоремы, описывающие его поведение, и подробности, необходимые для его надежной и эффективной реализации;
- ◆ *достаточно эффективными* с точки зрения затрат ресурсов, по крайней мере, в практических задачах. Чаще всего речь идет о процессорном времени и памяти. Когда процессор перебирает каждый бит, затраты памяти вряд ли превысят затраты процессора, но память важнее, потому что она может подождать еще час, а добавить лишнюю планку ОЗУ вы вряд ли сможете. К тому же она в процессе ожидания не утрачивается, в отличие от времени. Важны и другие ресурсы — такие как блокировки при конкурентном выполнении, дескрипторы файлов, сетевые порты, запросы к другим процессам и электроэнергия. Часто самыми дорогостоящими факторами являются оплата времени разработки и время ожидания ответа системы. Но пользователь не улавливает время отклика интерфейса, если оно меньше 150 мс, поэтому оптимизировать ниже этого значения надобности нет;
- ◆ *законными* — алгоритмы и их компоненты иногда являются интеллектуальной собственностью. Патенты и коммерческая тайна защищают идеи, товарные знаки и авторские права. Для использования защищенного программного обеспечения, в том числе и с открытым исходным кодом, требуется специальный договор, называемый *лицензией* (см. главу 4. Основы компьютерного права).

Правильный, эффективный и законный алгоритм уже будет *полезен*. А простота и расширяемость сокращают время разработки и обслуживания. В идеале алгоритм должен быть полезен, прост и расширяем, но единого понятия на этот счет не придумали. Термины вроде «полезный актив» (используемый в черновиках) или «достойный применения» больше сбивают с толку, чем вносят ясности. Но как бы там ни было, система, состоящая из таких алгоритмов, правильно и эффективно обрабатывает запросы пользователей и позволяет быстро добавлять новые функции и отлаживать их.

Многие полезные алгоритмы, позволяющие найти экономически ценные решения, имеют очень сложную реализацию, и специалисты в предметной области разрабатывают их библиотеки годами. Это дало им возможность получить впечатляющие результаты в линейном программировании, поиске кратчайших путей в сетях континентальных дорог, вычислительной геометрии и т. п. И даже если вы обнаружите в коде таких библиотек ошибки, вы не сможете самостоятельно их исправить, не зная точно, что делает код, и не имея гарантии, что на все изменяемые вами фрагменты кода имеется лицензия. Например, вам не удастся быстро учесть пользовательские сообщения об ошибках по проблемам, возникающим при использовании библиотек, как бы эти проблемы ни были редки.

Так что даже для реализации простых алгоритмов лучше использовать наилучшую доступную библиотеку, а не самодельное решение. И заменить самодельный код библиотекой — это весьма правильный подход. Структурируйте библиотеки слоями, чтобы максимально приспособить их для повторного использования и обеспечить эффективное нахождение повторно используемых компонентов.

В то же время для реализации более сложных задач из некоторых специальных областей требуются *более технически сложные* алгоритмы, потому что простые алгоритмы уже не справляются. В этом случае полезные алгоритмы оказываются слишком сложными, и к их разработке приходится достаточно плотно привлекать отраслевых специалистов (речь может идти о месяцах совместной работы). Примерами таких алгоритмов могут служить криптографические протоколы, компиляторы, алгебра разреженных матриц, нейронные сети, дорожная навигация через GPS, численные вычисления элементарных функций и т. п. Такие алгоритмы иногда подробно обсуждаются в специальных трудах, посвященных им, и в этой книге не затрагиваются, за исключением кратких упоминаний в комментариях к соответствующим главам. Эти алгоритмы могут содержать тысячи строк кода для обработки специальных случаев, и за их正确ностью очень тщательно следят.

Алгоритмы — это всего лишь инструменты для решения задач. В реальной жизни важно то, насколько хорошо с их помощью решается конкретная задача, и вы можете быть уверены, что набора готовых алгоритмов, непроницаемых как «черный ящик», на все задачи не хватит. Обычно требуется подумать, чтобы преобразовать задачу в форму, подходящую для решения с помощью такого «черного ящика», и часто приходится выбирать между несколькими вариантами. Кроме того, тот «черный ящик», который вы хотите получить, и лучший, который получить можно, иногда будут сильно отличаться друг от друга, и на их адаптацию может уйти немало сил.

## 1.3. Логика принятия решений

Моделировать требования к функциональности и свойства алгоритмов можно по-разному и с использованием различной логики. В этом разделе описаны наиболее важные подходы. Их глубинные тонкости бесполезны для большинства разработчиков, ибо доказательства правильности алгоритмов составляют специалисты. Нам важно знать только основы.

Например, зададимся вопросом: «Заканчивается ли выполнение этого цикла?». Вы могли бы сказать: «Очевидно, да», но стали бы вы использовать его в алгоритме для кардиостимулятора? *Логика* предполагает наличие *аксиом* для определения истинности

утверждений в языке. То есть, чтобы принять решение о завершении цикла, необходимо знать, как каждая строка кода влияет на условия завершения. Логика называется *полной*, если каждое истинное утверждение доказуемо. В общем случае бывает невозможно доказать, что цикл завершается, — например, если цикл перебирает четные числа, пока не будет найдено число, не являющееся суммой двух простых чисел. То, что этот цикл бесконечный, следует из гипотезы Гольдбаха (маленькая техническая подробность: вам понадобятся арифметика произвольной точности и неограниченная память).

В логике высказываний есть истинные/ложные переменные и утверждения, образованные с помощью логических операторов: И (&), НЕ (!), ИЛИ (|), СЛЕДУЕТ ( $\rightarrow$ ), исключающее ИЛИ ( $\wedge$ ), NAND (#) и др. Их использование помогает упростить операторы `if` за счет логических выражений — например:

$$(x \& (x \rightarrow y)) \rightarrow y.$$

Таблица истинности содержит результат высказывания для всех вариантов значений. Каждый приведенный ранее оператор определяется таблицей истинности из четырех строк. Табличная система обычно более эффективна. Два высказывания, соединенные оператором, должны давать определенные значения, чтобы оператор мог их получить. Пусть  $T(x)/F(x)$  означает, что  $x$  истинно или ложно соответственно. Отсюда следуют правила разложения:

- ◆  $T(x \& y) \leftrightarrow Tx \& Ty$ ;
- ◆  $F(x \& y) \leftrightarrow Fx|Fy$ ;
- ◆  $T!x \leftrightarrow Fx$ ;
- ◆  $F!x \leftrightarrow Tx$ .

С помощью таких правил мы в конечном итоге придем к противоречию (т. е. докажем ложность) или исчерпаем варианты разложения (т. е. докажем истинность). Подробности и примеры доказательств см. в книге [1.9].

Логика первого порядка — это логика высказываний с использованием операторов  $\forall$  (для каждого/любого) и  $\exists$  (существует). Она намного лучше подходит для связанных с циклами задач, однако не каждое ложное утверждение доказуемо (потому что невозможно перечислить все случаи). Впрочем обычно это не составляет большой проблемы. Добавление указанных операторов влечет за собой новые аксиомы и табличные правила (опять же, обратитесь к труду [1.8], ибо вдаваться здесь в технические подробности мы не станем). Язык логического программирования Prolog автоматизирует рассуждения в виде таблиц, и разработчики в лучшем случае используют интуитивные правила рассуждений (например, *modus ponens*, о котором вам рассказывали на лекциях по дискретной математике), правильность которых подтверждается таблицами.

Логика второго порядка позволяет проводить количественную оценку множеств и является наиболее естественной. Но она неполная, потому что доказуемые операторы  $\exists$  приводят к противоречиям. Поэтому лучше чаще использовать более простую логику.

Логика знания — это логика первого порядка, в которой агент  $i$  может знать истинность утверждения  $x$ . Отсюда следуют дополнительные аксиомы:

- ◆  $K_i x \rightarrow x$  — если агент что-то знает, то это истина;
- ◆  $K_i x \rightarrow K_i K_i x$  — агент знает, что он знает;

- ◆  $!K_i x \rightarrow K_i !K_i x$  — агент знает, чего он не знает;
- ◆  $K_i(x \rightarrow y) \rightarrow (K_i x \rightarrow K_i y)$  — позволяет агенту вывести новые знания.

Обратите внимание, что  $x \rightarrow K_i x$  ложно.

Есть и другие логики, полезные для выражения различных свойств алгоритмов:

- ◆ *вера* — агент может поверить во что-то ложное;
- ◆ *временная логика* — истинность утверждения зависит от дискретного времени и *всегда* и *в конечном итоге* имеет дополнительные операторы. Например, *сбой питания и сигнал с обеих сторон не может быть зеленым* — одни из правил работы светофоров.

## 1.4. Доказательство базовой правильности

Входные и выходные свойства — это соответственно *предусловия* и *постусловия*. Они формируют *контракт алгоритма*, который гарантирует, что входные данные, удовлетворяющие предусловиям, приводят к выходным данным, удовлетворяющим постусловиям. У многих алгоритмов существуют доказательства правильности. Они говорят, что последовательность операторов программы преобразует общее предусловие так, что в конце выполнения общее постусловие становится истинным. Но на практике полезнее полуформальные рассуждения, которые позволяют интуитивно считать программу в некоторой степени правильной. Формальные же рассуждения были предложены в 70-х годах прошлого века и никогда не пользовались популярностью, кроме особых случаев — таких как аппаратная верификация.

Входные данные, не удовлетворяющие предусловиям, обычно приводят к *неопределённому поведению*. Хорошие API проверяют входные данные и проверяют их правильность. В случаях, когда входные проверки ценнее самого алгоритма, имеет смысл пропустить их, оставив комментарий. Кроме того, некоторые входные данные нельзя проверить в принципе. Например, как проверить переданную пользователем функцию или компаратор? Даже для тривиального теста требуется, по крайней мере, возможность создавать элементы, к которым они применяются, что в шаблонном и универсальном коде C++ реализовать нельзя.

Возможно, генерация исключений в этом случае является более гибким механизмом и позволяет коду вызывающей стороны восстанавливать работу, но утверждения лучше в том смысле, что вызывающая сторона должна сначала проверить их. Бытует мнение, что всегда следует ожидать точного выполнения предварительных условий. Я видел лишь одно потенциальное исключение из этого правила — передача рандомизированной функции решателю уравнений, основанному на бинарном поиске (см. главу 23. *Численные алгоритмы: работа с функциями*), но здесь использование стандартного механизма решения уравнений сомнительно. Та или иная конкретная реализация может работать с незначительными нарушениями предварительных условий из-за более безопасного кодирования, но при этом другая схожая реализация может дать сбой.

Некоторые постусловия, которые мы явно не задавали, относятся к *несущественному поведению*. Например, точный формат вывода объекта в текстовый поток обычно может изменяться, и полагаться на его точность нельзя. Подобные ситуации также обычно не прописаны в документации.

*Инвариант* — это свойство, которое сохраняется во время вычислений, особенно циклов. Такие свойства не являются частью контракта, но позволяют оценивать правильность после определенного этапа вычислений. Например, представим задачу поиска наименьшего и преднаименьшего элемента в целочисленном массиве, где в качестве результата надо вернуть индексы (это вопрос с собеседования, и мы вернемся к нему позже в этой главе). Общее решение состоит в том, чтобы завести две переменные с индексами и постепенно обновлять их в цикле. Но попытка решить это без инварианта обычно приводит к проблемам, потому что трудно исправить плохую инициализацию переменных в цикле. Простой инвариант состоит в том, что после просмотра  $k$  из  $n$  элементов минимальный элемент определен правильно, а предминимальный равен  $-1$  или определен правильно. Решение состоит в том, чтобы инициализировать минимальный элемент равным  $0$ , предминимальный — равным  $-1$ , а цикл начать с индекса  $1$ .

Функция, вызывающая другую функцию, предполагает, что последняя выполняет свой контракт. Предварительные условия вызываемого объекта распространяются на вызывающего, но у высокоуровневой функции иногда бывает трудно описать точный контракт. Например, правильно написанная потоковая передача видео выдает сообщение об ошибке в случае проблем с сетью. Наилучший подход заключается в том, чтобы высокоуровневая функция брала на себя ответственность за предварительные условия функций, которые она вызывает, и осмысленно передавала их вызывающей стороне только в случае необходимости, зачастую сопровождая их дополнительным контекстом. Разработчику API часто приходится учитывать такие явления заранее, а также придерживаться логических строительных блоков, которые ожидают видеть пользователи.

Рассуждение в терминах предусловий, постусловий и инвариантов, вероятно, ограничивает практические формальные рассуждения. Алгоритмы должны быть спроектированы так, чтобы быть правильными с самого начала, и не нужно переходить к написанию кода до тех пор, пока не будут пройдены все тесты. Разработка *ad hoc* тоже полезна — например, если продумать основную логику, написать код, провести тесты и предусмотреть частные случаи. Но алгоритм всегда будет неполон без понимания его свойств, а отсутствие понимания должно вас по меньшей мере беспокоить. Тем не менее некоторые алгоритмы (например, *bootstrap* — см. главу 21. *Вычислительная статистика*) часто используются без полного понимания предусловий и постусловий, поскольку условия эти еще не определены.

Использование различной логики позволяет составить формальные доказательства свойств алгоритмов, но для нетривиальных задач нужны автоматизированные инструменты, которые, впрочем, тоже способны справиться лишь с относительно небольшими задачами. Полезны бывают, например, *статический анализ* (в том числе предупреждения компилятора о потенциальных проблемах — таких как неиспользуемые или неинициализированные переменные) и *проверка модели* (попытка найти контрпример для определенного оператора путем перебора всех возможных примеров).

Дополнительную информацию о полуформальных рассуждениях можно найти в трудах [1.1] и [1.3].

## 1.5. Асимптотическая запись

Алгоритм, работающий с входными данными из  $n$  элементов, использует  $r(n)$  ресурсов, что для достаточно большого  $n$ , любой константы  $b$  и некоторой константы  $c$  равно:

- ◆  $O(f(n))$  тогда и только тогда, когда  $r(n)/f(n) \leq c$ ;
- ◆  $o(f(n))$  тогда и только тогда, когда  $r(n)/f(n) < b$ ;
- ◆  $\Theta(f(n))$  тогда и только тогда, когда  $r(n)/f(n) = c$ ;
- ◆  $\omega(f(n))$  тогда и только тогда, когда  $r(n)/f(n) > b$ ;
- ◆  $\Omega(f(n))$  тогда и только тогда, когда  $r(n)/f(n) \geq c$ .

Регулярно используются только первые два варианта записи. Чаще всего в качестве  $f(n)$  задействуются функции  $1$ ,  $\lg(n)$ ,  $\sqrt{n}$ ,  $n$ ,  $n \lg(n)$ ,  $n^2$ ,  $n^3$ ,  $2^n$  и  $n!$ . В алгоритме следует применять самые медленнорастущие функции. Чтобы проверить, какая из двух функций растёт быстрее, можно использовать правило Лопиталя.

Асимптотическая запись упрощает анализ, отбрасывая постоянные факторы. Выполнение вычисления  $O(f(n))$  на протяжении  $O(g(n))$  раз эквивалентно  $O(f(n)g(n))$ . Например, если тело цикла ввода данных выполняется за время  $O(1)$ , общее время выполнения равно  $O(n)$ .

Сверхлинейное потребление памяти и суперквадратичное время выполнения в типовых задачах не масштабируются. Но не следует бояться алгоритмов, которым нужно чуть больше, если алгоритм является единственно верным. Например, самые полезные операции с матрицами (см. главу 22. *Численные алгоритмы: введение и матричная алгебра*) занимают  $O(n^2)$  памяти для хранения матрицы и требуют  $O(n^3)$  времени, чтобы сделать с ней что-то полезное. А эти операции, между прочим, регулярно используются во множестве алгоритмов оптимизации и машинного обучения. Они масштабируются до среднего размера входных данных ( $n \approx 1000$ ), а этого вполне достаточно для многих практических задач, которые в противном случае вообще не имели бы эффективных решений.

## 1.6. Машинные модели

Чтобы подчеркнуть основные функции оборудования, требуются сильно упрощенные интерфейсы. *Машинная модель* представляет собой набор операций с известными контрактами и эффективностью. Некоторые из них в основном теоретические, но все они важны. Все программы состоят из инструкций на языке ассемблера, напрямую поддерживаемых ЦП и другим оборудованием. ЦП работает с целочисленными словами размера  $w = 32$  или  $64$  бита, что позволяет использовать  $2^w$  ячеек памяти. Формат *double* позволяет реализовать вещественную арифметику с ограниченной точностью. В нем отводится 1 бит для знака  $s$ , 52 — для слова  $w$  и 11 — для показателя степени  $e$ . В результате вещественные числа представляются как  $\pm(1 + m) \cdot 2^e$  (подробнее см. в главе 22. *Численные алгоритмы: введение и матричная алгебра*). ЦП обращается к кешам, памяти, дискам и другим ресурсам для получения данных и одновременно выполняет множество программ, конкурирующих за ресурсы. Упрощенные модели предполагают разные эксплуатационные расходы. У современных компьютеров они выглядят следующим образом:

- ◆ арифметические, логические и битовые операции над числами `int` и `double`, а также вызовы функций выполняются за  $O(1)$  тактов ЦП, при этом все операции, кроме деления, занимают примерно одинаковое время, а условные операции замедляют конвейерную обработку ЦП;
- ◆ чтение из кеша и памяти и запись в них занимают  $O(1)$  тактов, но регистры ЦП примерно в 100 раз быстрее, чем память, и в 10 раз быстрее, чем кеш. Иногда сложность получается больше, чем  $> O(\lg(n))$ ;
- ◆ динамическое выделение и высвобождение памяти выполняются программно за время  $O(1)$  даже для больших массивов без затрат на конструкторы и деструкторы;
- ◆ доступ к таким устройствам, как диски, внешние накопители, экраны и сетевые маршрутизаторы, занимает тысячи циклов, но сводится к  $O(1)$ , если размер передаваемых данных ограничен;
- ◆ ожидание снятия блокировки может занять много времени, а управление ею занимает сотни циклов.

В моделях, разработанных для упрощения анализа, небольшие постоянные факторы игнорируются, а рассматривают только определенные узкие места:

- ◆ *оперативная память слов* — все операции выполняются над словами размера  $w > \lg(\text{размер ввода})$  и занимают время  $O(1)$ . Память не ограничена. Не затрагивает другие ресурсы. Большинство алгоритмов работают так;
- ◆ *реальная оперативная память* — оперативная память слов с бесконечной точностью `double`. Не выполняется обработка ошибок округления;
- ◆ *репрезентативные операции* — все операции, кроме предполагаемых узких мест, выполняются без затрат и помогают сосредоточиться на соответствующих операциях;
- ◆ *ввод/вывод* — доступ к различным устройствам, особенно к дискам, занимает большое время  $O(1)$  при размере данных  $\leq B$ . Полезно, когда узким местом является доступ к устройству;
- ◆ *PRAM* — оперативная память слов с неограниченным числом параллельных процессоров и общей памятью. Связь между процессорами выполняется без затрат. Отражает пределы масштабируемости.

Некоторые модели вместо затрат определяют верхние границы и технически не являются машинными моделями:

- ◆ *модель реального времени* — на вычисление ответа отводятся сроки, несоблюдение которых обходится слишком дорого;
- ◆ *поток* — память  $O(1)$ , входные данные поступают непрерывно. Обработка каждого элемента данных может осуществляться в режиме реального времени — например, в работе сетевого маршрутизатора. Обобщает модель ввода/вывода;
- ◆ *массивные данные* — обработка триллионов входных данных, хранящихся на диске или передаваемых в потоковом режиме. Требуется масштабируемая обработка, возможна сложность  $O(n \lg(n))$ .

## 1.7. Рандомизированные алгоритмы

Использование случайных чисел очень полезно в вычислениях, но, к сожалению, не понято сообществом программистов. Давайте сначала рассмотрим общие игры.  $\forall$  в игре  $\exists$  — оптимальная рандомизированная стратегия, даже если  $\exists$  — оптимальная детерминированная стратегия. Это называется *равновесием Нэша*. Например, в игре «Камень-ножницы-бумага» случайный выбор опирается на среднее значение, но детерминистской стратегии можно научиться. Рандомизация же может позволить обрабатывать входные данные непредсказуемо с хорошей ожидаемой эффективностью, несмотря на гораздо худший детерминированный наихудший случай. В рандомизированных алгоритмах принятия решений используется генератор псевдослучайных чисел — такой как `rand()`.

*Алгоритм Монте-Карло* дает неверные ответы с небольшой вероятностью. Особым случаем является алгоритм принятия решения, в котором «да» верно, а «нет» неверно с вероятностью  $p$ . Запустив его  $k$  раз и получив ответ «нет», мы снижаем вероятность отказа до  $p^k$ . Более общая стратегия состоит в том, чтобы объединять ответы из разных прогонов большинством голосов, но ее сложнее анализировать.

У *алгоритма Лас-Вегаса* есть ожидаемая эффективность. Если повторять алгоритм Монте-Карло с верификатором ответа до тех пор, пока не будет найден правильный ответ, получится алгоритм Лас-Вегаса. Вероятность ошибки или чрезмерной неэффективности рандомизированного алгоритма часто пренебрежимо мала. Чтобы проанализировать алгоритм Лас-Вегаса, рассмотрим его удачу:

- ◆ худшая удача и входные данные дают худший случай;
- ◆ наилучшая удача и входные данные дают лучший случай;
- ◆ лучшая удача и худшие входные данные дают нижнюю границу.

Рандомизированные алгоритмы часто оказываются наиболее эффективными — например, для поиска строк в большом тексте (см. главу 14. *Строковые алгоритмы*), проверки простоты числа (см. главу 17. *Большие числа*) и т. д.

При использовании алгоритмов Лас-Вегаса возникает небольшая проблема, которая выражается в зависимости метрик от времени выполнения. В некоторых случаях это влияет на результаты непредсказуемым образом — например, при одинаковых значениях разные случайные перестановки приводят к разным значениям минимума. Для алгоритмов, дающих приближенный ответ, качество результата в разных запусках может отличаться.

## 1.8. Измерение эффективности

Для измерения затрат ресурсов можно использовать один из следующих способов:

- ◆ *максимальный/худший случай* — дает полную уверенность, но может оказаться намного выше среднего;
- ◆ *усреднение* по случайному выбору алгоритма или случайных входных данных — в первом случае результат не зависит от ввода и его не всегда можно использовать. В некоторых случаях медианное значение лучше среднего, но оно редко используется;



- ◆ *Pr(использование > некоторое число)* — действует как наихудший случай для алгоритмов Лас-Вегаса при пренебрежимо малом значении;
- ◆ *амортизированная эффективность* — средний наихудший случай для последовательности операций. Иногда допускает дорогостоящие операции;
- ◆ *конкурентный коэффициент* — максимальный мультипликативный коэффициент, на который совершенно удачливый алгоритм превосходит *онлайн-алгоритм* при последовательном получении входных данных и выдаче лучшего ответа. Например, замена кеша LRU является 2-конкурентной. Вычислить этот коэффициент математически сложно, и это делается исследователями конкретных алгоритмов;
- ◆ *сглаженная эффективность* — среднее значение случайно возмущенных входных данных для наихудшего случая. Это нечто среднее между наихудшим случаем и случайным входом, предназначенное для алгоритмов с плохим наихудшим случаем и хорошей практической производительностью (см. [1.7]). Это тоже лучше оставить исследователям.

Каждый показатель эффективности может быть:

- ◆ *точным* — в метрике сложности присутствуют коэффициенты  $O(1)$  и соответствующие члены более низкого порядка. Имеет смысл использовать только в том случае, если примитивные операции имеют точную стоимость, но сложны и редко выполняются. В лучшем случае вы получите приблизительные результаты, потому что современные компиляторы выполняют множество оптимизаций;
- ◆ *асимптотическим* — эффективные решения не зависят от времени и машины и легко поддаются сравнению, но скрывают огромные коэффициенты  $O(1)$  и обосновывают алгоритмы, которые работают хорошо только на нереалистичных входных данных;
- ◆ *экспериментальным* — оценивается при выполнении алгоритма на различных задачах.

Максимальное, среднее и амортизированное значения при одинаковой сложности более информативны, но в некоторых случаях амортизированное значение лучше среднего. Например, алгоритм сложностью  $O(\lg(n))$ , амортизированный  $O(1)$  и ожидаемый  $O(1)$ , является ожидаемо амортизированным  $O(\lg(n))$ . Худший случай распространяется на зависимые задачи.

Контракт алгоритма определяет теоретико-информационную нижнюю границу эффективности любого алгоритма для той или иной задачи. Например, если нужно прочитать каждый бит входных данных.

## 1.9. Типы данных

Существуют теоретические свойства типов данных, которые помогают писать более эффективный код. Система C++ STL была задумана с учетом этого (подробнее см. в [1.9]).

*Тип данных* — это множество значений. Он может быть:

- ◆ *хорошо формируемым* — если любая битовая последовательность, назначенная объекту этого типа, преобразуется в допустимое значение. Например, если Boolean — это char, у которого 1 = true и 0 = false, то что значит символ «2»?

- ◆ *частично формируемым* — если конструктор по умолчанию создает разрушаемые объекты, которым можно выполнять присвоения. У алгоритма не может быть контракта, если правильность требует конструирования объекта из переданных данных;
- ◆ *неоднозначным* — если одна и та же последовательность битов представляет множество логических значений. Например, в датах с двумя последними цифрами года не различаются столетия;
- ◆ *уникально представленным* — если каждое значение соответствует уникальной битовой последовательности. Например, для логического значения, представленного байтом, 0 соответствует false, а все остальное — true;
- ◆ *копируемым* — если у него есть имеет деструктор, конструктор копирования и оператор присваивания. Это минимальный набор, необходимый для размещения объектов в структурах данных.

*Значение* объекта — это то, что можно получить из его открытого интерфейса, и оно является подмножеством состояния объекта. Функция называется *регулярной*, если замена входных данных на эквивалентные значения не влияет на выходное значение. Например, генераторы случайных чисел, устройства чтения данных и таймеры не являются регулярными. Любая функция, использующая свойства входных данных, к которым неприменима операция равенства — например, адреса памяти, также не является регулярной. Нерегулярные функции сложно тестировать, поэтому их принято раскладывать на регулярные и нерегулярные компоненты.

*Представлением* типа может быть:

- ◆ слово;
- ◆ указатель на тип;
- ◆ массив типов;
- ◆ структура типов.

Функции, определенные для типа, составляют его *вычислительную базу*. База называется:

- ◆ *минимальной* — если с ее помощью можно выполнять все полезные операции с незначительными потерями эффективности. Для простоты рекомендуется использовать меньшие вычислительные базы и не поддаваться искушению добавлять дополнительную функциональность;
- ◆ *выразительной* — если в нее входят предназначенные для удобства функции, реализованные на основе минимальной базы. Например, моя реализация вектора (см. главу 5. *Фундаментальные структуры данных*) может выполнять операции в векторном пространстве и добавлять векторы элементов.

Следует определять классы с минимальной базой, а для расширения функциональности использовать сторонние функции или слои и шаблоны фасада (см. главу 2. *Основы разработки программного обеспечения*). У сложных типов полная вычислительная база бывает неизвестна, особенно если она расширена за счет дополнений. В библиотеках предоставляется достаточно полный набор операций, которых хватает для нужд практически всех пользователей, но иногда приходится реализовывать индивидуальную функциональность.

Тип без указателей — это *простые структуры данных* (POD, plain old data). Структура данных обладает *постоянством адреса*, если только удаление не делает адрес элемента недействительным. *Концепция* — это множество типов данных с общей вычислительной основой. Например, копируемые типы определяют содержащийся в них элемент. Это похоже на абстрактную алгебру, где выполняются вычисления в структурах с ограниченными возможностями. В C++ используются схожие шаблоны, поскольку задействуются только подтипы и функции, определенные внутри типа.

## 1.10. Эксперименты с алгоритмами

Для изучения поведения алгоритма:

1. Выберите факторы, которые хотите изучить. В статистике *фактором* называют любое влияющее на измерения свойство — например, размер входных данных или их категория сложности для конкретной задачи. Выберите размер входных данных и удваивайте его, пока позволяет терпение. Алгоритм и выбор его параметров — это тоже полезные факторы.
2. Выберите метрики (измеряемые показатели). Вы можете измерить использование ресурсов и количество вычислительных операций в дорогостоящих процедурах, а также специализированные показатели — такие как качество решения в задаче оптимизации. Число операций — это универсальная для разных машин метрика, но различия в оптимизации машины и компилятора влияют на время выполнения и затраты памяти.
3. Выберите входные данные. Постарайтесь представить каждое значение фактора, применив разумную дискретизацию в случае непрерывных входных данных или целочисленных входных данных с большим диапазоном. Вот список часто встречающихся типов входных данных и их проблем:
  - реальные примеры используемых в производстве данных обычно не достать;
  - данные из общедоступных тестовых библиотек имеют отклонения, присущие конкретной задаче;
  - случайно сгенерированные данные могут получить свойства, из-за которых алгоритмы будут вести себя не так, как с реальными входными данными.
4. Выполняйте тесты в контролируемой среде с малой нагрузкой на систему, чтобы только выбранные вами факторы значительно влияли на результат. Запустите детерминированные алгоритмы один раз для каждого набора данных. Если алгоритмы или среда случайны, проведите необходимое количество запусков для получения достаточно точных оценок. Здесь может возникнуть множество проблем. Например, во время моих экспериментов система Windows 7 отдавала на алгоритм  $\frac{1}{6}$  процессорного времени, а Windows 10 —  $\frac{1}{10}$ . Кроме того, оптимизация компилятора позволяет сократить время выполнения примерно в четыре раза, если процесс выполняется несколько дней.
5. Анализируйте результаты с помощью статистических методов:
  - линейная регрессия (или лассо — см. главу 27. *Машинное обучение: регрессия*) по количеству операций и использованию ресурсов может предсказать последнее как функцию первого;

- регрессия по количеству операций или использованию ресурсов и функциям размера входных данных ( $1$ ,  $\lg(n)$ ,  $\sqrt{n}$ ,  $n$ ,  $n\lg(n)$  и  $n^2$ ) точно отражает асимптотическую производительность. Это методика исследовательского анализа, поэтому, найдя лучший вариант функции, примените выбранную функцию к новым данным для подтверждения, чтобы избежать просмотра данных и многократного тестирования;
  - если размеры входных данных определяются путем удвоения, регрессия с логарифмической функцией использования ресурсов или количества операций может предсказать полиномиальную метрику производительности.
6. При желании опубликуйте свои результаты. Они должны обладать *научной ценностью* — то есть порождать ценные, новые и воспроизводимые выводы. Обязательно укажите, какие использовались входные данные и реализации. Многие работы не обладают научной ценностью, так как несут в себе:
- уже опубликованные ранее выводы. Обычно это считается нормальным, только если проведенный эксперимент стал лучше или существенно отличается;
  - неуниверсальные метрики, предназначенные для специализированных машин. Как правило, это не проблема, если сравнение все же можно выполнить. Иногда выполнить сравнение помогает отображение относительных, а не абсолютных значений, а также данные о настройках машины и компилятора;
  - результаты с неподтвержденными данными статистического анализа;
  - результаты, которые не приводят к полезным выводам.

Это простой шаблон, которого недостаточно для статистически значимого сравнения производительности алгоритмов. Распространенной ошибкой является использование пары входных данных и утверждение о превосходстве одного алгоритма над другим на основе результата теста значимости средних значений. В *главе 21. Вычислительная статистика* мы рассмотрим, как организовать такое сравнение, чтобы оно имело смысл. Кроме того, выбор алгоритма осуществляется по другим критериям, а не по экспериментальному сравнению производительности. Например, у алгоритмов сортировки оценивается и сравнивается константа у времени выполнения  $O(n\lg(n))$ , и именно по ней выполняется сравнение.

## 1.11. Управление памятью

Операционная система распределяет память в виде больших байтовых массивов размером от 512 Кбайт до 4 Мбайт, называемых *страницами*. Любой исполняемый файл использует одну или более страниц. В C++ страницы `new` и `delete` делятся на более мелкие блоки. Приложение, которому требуется больше памяти, запрашивает еще одну страницу (или несколько, если запрос превышает размер страницы). Неиспользованные страницы возвращаются.

На некоторых машинах  $x$ -байтовые переменные должны находиться по адресу, кратному  $x$ . Компилятор выполняет выравнивание адресов, если переменные не преобразуются и не переинтерпретируются. Например, выделение переменной `char x[8]`, размещение там числа `double` и обращение к нему как к `double` может вызвать аппаратное исключение. Размеры структур:

- ◆ `struct Aligned {double d; char c;} — 16;`
- ◆ `struct NotAligned{char c[9];} — 9.`

Диспетчер памяти ведет заполнение значений и учет всех выделенных элементов. Он выполняет выравнивание адресов, не зная типа, и поддерживает необработанные запросы памяти `malloc` и `free`, используемые конструкторами при выделении страниц `new` и `delete`. У любого менеджера памяти существуют теоретические ограничения эффективности использования памяти в худшем случае, но на практике даже в долго работающих системах они не достигаются. Хорошо реализованные менеджеры памяти — например, комбинации *списков первого соответствия* и *отдельных списков* (см. [1.2]) — обладают следующими характеристиками:

- ◆ операции занимают большое время  $O(1)$ ;
- ◆ потери памяти на одно выделение составляют 4–8 байтов;
- ◆ невозможно использовать часть памяти, состоящую из небольших несвязанных фрагментов.

В большинстве приложений выделяется много мелких элементов — см. главу 5. *Фундаментальные структуры данных*, где мы поговорим о системе выделения со сборкой мусора.

## 1.12. Оптимизация кода

Многие программисты выполняют слишком много или слишком мало оптимизаций. Обоснование многих таких решений также часто бывает ошибочно. В конечном итоге эффективность определяют выходные данные сборки компилятора. Если отладка не требуется, нужно применять максимальный уровень оптимизации, который создает переносимый исполняемый файл и не меняет поведение (на презентации, где я присутствовал, эксперт по суперкомпьютерам IBM заявил, что, исходя из его опыта, слишком высокий уровень оптимизации может изменить поведение во время выполнения). Компилятор может, в числе прочего:

- ◆ вычислять константные выражения;
- ◆ менять порядок инструкций;
- ◆ встраивать функции, заменяя их вызов их телом;
- ◆ преобразовывать оператор `switch` в последовательность `if` или массив указателей перехода, индексированных константами переключения.

Для изучения процесса генерации сборки вам пригодится интерактивная компиляция на сайте **godbolt.org**.

Вы можете помочь компилятору, если ваш код будет:

- ◆ универсальным — один раз опишите общую функциональность и используйте ее повторно. Не выполняйте копирование и не реализуйте вручную раздутые оптимизации, такие как встраивание, развертывание циклов или замена дорогостоящих операций последовательностью дешевых операций. Не обобщайте код чрезмерно, обрабатывая ненужные случаи. Удалите любой неиспользуемый код;
- ◆ простым — коротким и самодокументируемым. Используйте единый стиль и короткие описательные имена, а комментируйте только высокоуровневые действия. Ком-

ментарии не влияют на поведение во время выполнения, поэтому поддерживать их сложнее. Кроме того, программисты, которые не пишут самодокументирующийся код, вряд ли смогут объяснить его работу в комментариях;

- ◆ максимально ограниченным — по возможности используйте константы, минимизируйте область видимости переменных, используйте минимальное количество операторов `return` и т. д.

Эффективность должна исходить главным образом из лучших алгоритмов и логики предметной области. Но иногда помогает и ручная оптимизация. Вот некоторые предложения и подсказки на этот счет:

- ◆ остерегайтесь специфического для конкретной машины и неопределенного поведения. Например, на моей машине  $-5 \% 2 = -1$ ;
- ◆ профилируйте код и анализируйте, на что он тратит больше всего ресурсов. Попробуйте сократить затратную активность или оптимизировать ее, улучшив логику. Например, отключение утверждений путем определения `NDEBUG` почти ничего не дает, но существенно связывает вам руки с точки зрения отладки;
- ◆ использование `void*` вместо шаблонов предотвращает встраивание и оптимизацию, в которых используется информация о функции. Используйте его только для того, чтобы скрыть реализацию, скомпилировав файл `src` как библиотеку с оболочкой шаблона для безопасности типов;
- ◆ оптимизируйте выравнивание для объектов в массивах — компилятор вставляет фиктивные байты для выравнивания, а  $\text{sizeof}(\text{структура}) \geq \sum \text{sizeof}(\text{элементы структуры})$ . Сначала объявляйте большие объекты. Размер массива равен размеру его элементов, а размер структуры равен ее `sizeof` — например:
  - `struct Wasteful{char a; double d; char b;}` — размер 24;
  - `struct Smart{double d; char a; char b;}` — размер 16;
- ◆ компилятор может не оптимизировать вычисления с плавающей точкой или код, который зависит от их результата, потому что это обычно меняет семантику. Например, сложение не является ассоциативным из-за округления, да и порядок вычислений менять нельзя;
- ◆ цикл проверяет условие завершения на каждой итерации, а компилятор перемещает его наружу, если может доказать, что это не меняет семантику. Так что на всякий случай дорогие условия лучше вынести наружу;
- ◆ загрузка большого блока данных из памяти, его обработка и переход к следующему блоку повышает эффективность кэширования;
- ◆ попробуйте сделать так, чтобы вычисления с вложенными циклами и другими простыми узкими местами не зависели от условных выражений, — компилятор угадывает, какая инструкция пойдет в конвейер следующей, и, если она ошибочна, ЦП повторяет вычисления;
- ◆ возвращайте большие объекты по ссылке, чтобы избежать лишнего копирования, если это не повредит интерфейсу или если компилятор не сможет оптимизировать копию.
- ◆ Не используйте *стражи* — многие алгоритмы предполагают наличие элементов  $\infty$  или нулевых элементов, чтобы упростить представление или избежать проверок, но:

- предсказание переходов снижает стоимость сохраненных сравнений;
- общий тип не имеет логической  $\infty$ ;
- стражу может потребоваться дополнительное пространство, которого нет;
- вычисления обычно труднее для понимания;
- ◆ вычисляйте как можно больше во время компиляции, но не слишком усложняйте код. Уменьшите количество кода под шаблоном. Например, векторы указателей C++ имеют специализацию `void*`. Если шаблоном должен быть только метод класса, не делайте класс шаблоном;
- ◆ выделение памяти блоками, равными степеням двойки, эффективно для любой системы выделения. Имейте в виду, что ОС не поддерживает массивы большего размера;
- ◆ избегайте статических и глобальных переменных, поскольку они могут занять значительный объем памяти. Помещайте переменные в стек с максимально ограниченной областью действия;
- ◆ преобразование массива логических значений в битовый набор экономит место, но замедляет доступ.

В работе [1.5] вы можете найти больше подробной информации и других советов.

## 1.13. Рекурсия

Рекурсивная функция помещает локальные переменные в стек, вызывает сама себя и ожидает возвращаемого значения. Для эффективности следует поместить переменные и параметры, которые не изменяются и не появляются перед рекурсивным вызовом, внутрь блока `{}` или сделать их членами класса.

Глубина рекурсии ограничена размером стека ОС. Она также влияет на локальные переменные и не является переносимой (например, на моем компьютере 2011 года при выделении массива `int` размера  $> 2^{19}$  в стеке происходит сбой). Попробуйте сделать любой рекурсивный алгоритм, требующий большого стека, нерекурсивным. Это также оптимизирует накладные расходы на вызовы функций и позволяет выполнять встраивание.

*Хвостовая рекурсия* — это явление, когда рекурсивный вызов является последним оператором функции. Чтобы преобразовать его в итерацию:

1. Оберните циклом тело функции.
2. Вынесите любые запомненные переменные вонне цикла.
3. В конце цикла установите значения параметров, переданных рекурсивному вызову, и удалите их.

Эта методика позволяет упростить код, если в нем не используются ссылки. Чтобы удалить общую рекурсию, сохраните стек записей — это позволит алгоритму возобновить свою работу после возврата, и напишите вокруг рекурсивной задачи цикл, который прервется, когда стек опустеет. Обычно это приводит к некоторым упрощениям — так, некоторые переменные, передаваемые в качестве аргументов, могут не находиться

в стеке. Чтобы стало проще думать, сформулируйте алгоритм нерекурсивно, мысленно выполнив его.

Вы можете быстро анализировать многие рекурсивные алгоритмы, используя *основную теорему* (см. [1.10]): пусть затраты ресурсов:

$$R(n) = f(n) + aR(n/b)$$

для  $n >$  константы  $C$  и  $O(1)$  в противном случае, причем  $k = \log_b(a)$ .

Тогда  $R(n)$  равно:

- ◆  $\Theta(n^k)$ , если  $f(n) = \Theta(n^c)$  при  $c < k$ . Общая работа на следующем уровне рекурсии уменьшается геометрически.
- ◆  $\Theta(n^c \lg(n)^j + 1)$ , если  $f(n) = \Theta(n^c \lg(n)^j)$  при  $c = k$ . Суммарная работа на всех уровнях  $O(\lg(n))$  одинакова.
- ◆  $\Theta(n^c)$ , если  $f(n) = \Theta(n^c)$  при  $c > k$ . Суммарная работа на следующем уровне увеличивается в геометрической прогрессии.

## 1.14. Стратегии вычислений

Существует несколько стратегий, которые позволяют решить многие проблемы:

- ◆ *разделяй и властвуй* — разделяй входные данные на части и комбинируй ответы на них. Например, алгоритмы быстрой сортировки и сортировки слиянием (см. главу 7. *Сортировка*) разбивают массив и работают с каждой частью отдельно;
- ◆ *жадная стратегия* — на каждом шаге предпринимайте самые перспективные действия. Например, алгоритм Прима (см. главу 11. *Алгоритмы графов*) итеративно добавляет вершину с наименьшей стоимостью к текущему MST;
- ◆ *динамическое программирование* — когда решение представляет собой последовательность из  $n$  действий, а  $E[\text{общая стоимость}] = \sum E[\text{стоимость каждого действия}]$ , и чтобы найти наилучшее действие для шага  $i$ , нужно выбрать лучшие действия для шагов с 0 по  $i - 1$ , а затем для шага  $i$ . Например, путь из  $A$  в  $B$ , проходящий через  $C$ , оптимален, если пути из  $A$  в  $C$  и из  $C$  в  $B$  оптимальны, поэтому из  $B$  нужно найти наилучший путь к каждому ближайшему  $C$  и проверить, какой из них ведет к наилучшему общему пути. Динамическое программирование обычно применяется, когда проблема менее эффективно решается с помощью рекурсии. Например, при вычислении  $n$ -го числа Фибоначчи рекурсия пересчитывает результаты, такие как число с номером  $(n - 2)$ , которое динамическое программирование запоминает;
- ◆ *смешивание алгоритмов* — если два алгоритма лучше всего подходят для разных случаев, используйте каждый из них в подходящей ситуации. Например, в сортировке STL используется детерминированная быстрая сортировка по медиане трех точек с переключением на пирамидальную сортировку, если стек слишком глубокий, что дает быстрое среднее время выполнения и гарантию  $O(n \lg(n))$ .

## 1.15. Выбор среди нескольких алгоритмов

Для программиста обычно выбор прост — использовать все, что предлагает выбранный API. В этом разделе рассматривается случай, когда вы сами пишете API.



Большинство алгоритмов легальны и корректны, но различаются показателями расширяемости, простоты и эффективности. Выигрыш в простоте приводит к экономическому выигрышу, но выигрыш в эффективности происходит только в том случае, если алгоритм находится на критическом пути, а для пользователя важна дополнительная эффективность. Так что *выбирайте самый простой и достаточно эффективный алгоритм, если немного более сложный алгоритм оказывается не сильно эффективнее.*

Эффективность с малым постоянным коэффициентом обычно имеет значение для широко используемых библиотек. Реализация сразу всех алгоритмов, каждый из которых наиболее эффективен для конкретного случая, и выбор одного из них во время выполнения — это сущий кошмар с точки зрения обслуживания кода. Время разработки стоит больше, чем даст возможная экономия ресурсов, за исключением случаев использования нескольких простых алгоритмов. Но эта стратегия тоже может быть полезной, например когда один алгоритм задействуется для инициализации, а другой — для итеративного уточнения. В таких случаях следует комбинировать алгоритмы с одинаковыми затратами ресурсов, не думая о том, что какой-либо из них оказывается асимптотически намного медленнее или расходует гораздо больше памяти, чем остальные.

В идеале следует избегать размышлений и автоматизировать как можно больше. Для задач, которые полностью автоматизировать нельзя, важно знать, какие факторы следует учитывать при выборе алгоритма. С точки зрения любого важного свойства алгоритма лучше избежать плохого выбора, чем сделать лучший.

Во многих задачах бывает сложно выбирать алгоритмы. Например, в задачах глобальной численной оптимизации (см. главу 24. *Численная оптимизация*) неясно даже то, как подобрать алгоритмы для гибридного подхода, который пробует всё понемногу. В сферах, где разрабатываются новые алгоритмы, а старые становятся всё более устоявшимися, возникает искушение использовать только то, что считается отраслевым стандартом. Для первой попытки это хорошая идея, но следует не бояться экспериментировать с новыми идеями или комбинациями. При подготовке этой книги я внес полезные изменения во многие стандартные алгоритмы.

## 1.16. Создание параллельных алгоритмов

*Параллелизм* полезен, когда приложение выполняет очень длительные вычисления или взаимодействует с несколькими ресурсами. Например, во многих вычислениях используются кластеры компьютеров, а браузер отображает части веб-страницы до того, как она полностью загрузится. Но в масштабируемых вычислениях, не использующих операции ввода/вывода, обычно параллелизм не требуется. Задачу можно разложить по следующим компонентам:

- ♦ *входные данные* — каждый процессор обрабатывает свою часть данных. Применяется к алгоритмам «разделяй и властвуй», рандомизированным алгоритмам, повторно запускаемым со многими случайными начальными значениями, и системам, обрабатывающим независимые события. Например, в модели PRAM поиск по несортированному массиву занимает время  $O(n)$  и параллельное время  $O(\lg(n))$  за счет рекурсивного предоставления каждой половине подмассива отдельному процессору.

- ◆ *функциональность* — каждый процессор выполняет определенное преобразование (например, стиральная и сушильная машины).

Существуют следующие способы реализовать параллелизм:

- ◆ *отдельные процессы* — связь осуществляется через *передачу сообщений* либо напрямую, либо через маршрутизатор. Процессы могут выполняться на разных машинах. Стоимость сообщений соответствует модели ввода/вывода, а операционной системе требуются дополнительные ресурсы на каждый процесс;
- ◆ *отдельные потоки* — потоки стоят меньше, чем процессы, и используют общую память, как предполагается в PRAM. Для корректной реализации одновременного доступа применяются блокировки. Но программирование с блокировками сложно и неэффективно, если одной блокировки ждет много потоков. Иногда используют *акторную модель*, которая представляет собой абстракцию передачи сообщений.

При блокировке публичных методов структура данных является атомарной, но не полностью безопасной. Например, возникает *состояние гонки*, когда два потока проверяют, пуст ли стек, и выталкивают из него значение, даже если эти методы заблокированы. Блокировку можно использовать для увеличения сложности и производительности, но уменьшения применимости, в следующих местах:

- ◆ диапазон кода, содержащий вызовы структур данных;
- ◆ публичные методы структур данных;
- ◆ разрешение на запись одному потоку и на чтение неограниченному числу потоков;
- ◆ части структуры данных, доступные в текущий момент.

Еще одна проблема — *дедлоки* («мертвые блокировки»). Если два потока ожидают двух блокировок, и каждый держит одну из них, то ни один из них не может ничего делать. Такая ситуация невозможна, если все потоки получают блокировки в одном и том же порядке.

Следует распараллеливать самый высокий уровень программы, а не функциональность более низкого уровня. При этом код обычно получается эффективнее и проще в обслуживании.

## 1.17. Реализация алгоритмов

Хотя в этой книге я описываю свои реализации распространенных алгоритмов, практически никогда не бывает просто создать реализацию по типичному описанию из учебника. Этот процесс гораздо более сложен:

1. Прочитать введение, чтобы уловить основную идею алгоритма.
2. Написать код базовой функциональности.
3. Проверить это на нескольких простых примерах.
4. Ознакомиться с другими источниками, рассмотреть и проанализировать варианты.
5. Поэкспериментировать с многообещающими вариантами, если таковых несколько.
6. Написать окончательную версию со всеми выбранными вариантами.
7. Очистить код.

8. Написать комплексные, желательно автоматизированные, тесты.
9. Читать новые источники и исследования в поисках улучшений.

Самая большая проблема заключается в том, что почти всегда для полного понимания недостаточно одного источника, и эта книга — не исключение. А вот базового введения с большим количеством примеров в сочетании с книгой уже достаточно.

Распространенная ошибка — игнорировать пункт 8. Тесты для многих реализаций я написал спустя годы, и мне пришлось заново знакомиться со всеми деталями, необходимыми для обработки примеров и исправления найденных ошибок.

Никакая реализация никогда не бывает окончательной. Как правило, много работы требует реализация пункта 9, поскольку в источниках постоянно производится исправление крупных и мелких ошибок, добавление тестов и внесение улучшений, использование лучших подкомпонентов. Лучше иметь готовую, но полезную реализацию, чем идеальную.

В большинстве глав есть раздел *«Примечания по реализации»*, в котором я рассказываю о некоторых конкретных трудностях в реализации определенных алгоритмов и структур данных, а также о шагах, которые я предпринял для их решения.

## 1.18. Рекомендуемые курсы для студентов, изучающих информатику

В колледже я посещал 6–9 занятий в семестр, и далее приведены наиболее полезные курсы с разных факультетов.

### ◆ Математика:

- Исчисление — основная информация.
- Линейная алгебра — численные алгоритмы, оптимизация, машинное обучение и т. д.
- Теория вероятностей — основа рандомизированных алгоритмов и большей части математического моделирования.
- Математическая статистика — вероятностное обоснование статистического вывода.

### ◆ Желательно:

- Стохастические процессы — моделирование и понимание некоторых алгоритмов.
- Абстрактная алгебра — теоретические концепции, относящиеся к проектированию многих дискретных систем.
- Математический анализ — для анализа алгоритмов, знакомства с доказательствами и общей математической зрелости.
- Численные методы — чтобы знать, как компьютеры решают уравнения, оценивают функции и т. д.
- Комплексный анализ — для анализа некоторых численных методов.

## ◆ Экономика (желательно):

- Микроэкономика — принятие решений, включая теорию полезности.
- Теория игр — принятие решений с учетом информации о других заинтересованных сторонах.

## ◆ Информатика (многие из этих курсов были факультативными):

- Алгоритмы и структуры данных — ядро вычислительной техники, это следует изучить по максимуму.
- Вычислимость и сложность — пределы вычислений.
- Архитектура компьютера — как работает аппаратное обеспечение.
- Операционные системы — как работает ОС, это важно для общего понимания.
- Сети — основные протоколы связи.
- Базы данных — хранение данных на диске и доступ к ним с помощью SQL.
- Распределенные системы — следующий шаг после сетей и ОС, важный для понимания реальных структур системы.
- Искусственный интеллект — решение игр и головоломок, логика и вероятностные вычисления.
- Машинное обучение — как компьютеры собирают статистику.

## ◆ Желательно:

- Вычислительная геометрия — с сильным акцентом на алгоритмы и структуры данных.
- Оптимизация — линейное программирование, алгоритмы аппроксимации и мета-эвристика.
- Программирование в специальной среде — использование различных API для создания реальных приложений.

Прохождение математики на специализированных курсах наиболее полезно, потому что ее разделы весьма сложно изучать самостоятельно, и на них вы получите интуитивное представление о том, как работают те или иные вычисления. Может быть сложно включить в расписание все занятия, которые вы хотите посетить, особенно, если вас больше интересует получение степени, но существуют стратегии, которые могут помочь:

- ◆ посещайте как можно больше курсов AP (Advanced Placement)<sup>2</sup> — многие колледжи учитывают это при поступлении, а некоторые предлагают летние и регулярные курсы для старшеклассников;
- ◆ запланируйте предварительные условия — пройдите основные курсы как можно скорее, потому что курсы более высокого уровня могут предлагаться редко, а цепочка обязательных требований может быть длинной;

---

<sup>2</sup> Advanced Placement (AP) — программа углубленного изучения учебных предметов в школах США, Канады и Великобритании. Она готовит старшеклассников к поступлению в вузы.

- ◆ посещайте летние и зимние курсы — откажитесь от общеобразовательных предметов, чтобы освободить место для более важных тем;
- ◆ запишитесь на комбинированную программу бакалавриата/магистратуры — возможно, вы сможете получить степень магистра за год вместо обычных двух, поступив в аспирантуру досрочно.

## 1.19. Некоторые стратегии обучения

- ◆ Жадная — активно поглощать любую новую информацию и пытаться сразу ее понять. Это самый эффективный способ обучения, кроме случаев, когда информация плохо представлена и бесполезна.
- ◆ Ленивая — читать материалы по выбранной теме во многих источниках, не тратя слишком много усилий на каждый. Цель этого подхода — задействовать подсознательную обработку. Можно воспользоваться несколькими различными источниками, которые дополняют друг друга.

Ленивая стратегия использовалась чемпионом мира по шахматам Тиграном Петросяном во время подготовки к матчам — он быстро отыгрывал недавние гроссмейстерские партии, и с тех пор эта стратегия стала использоваться в советских шахматных школах. Этот подход также хорош, если вы хотите стать экспертом, поскольку в любом отдельно взятом источнике может быть упущено что-то важное, особенно, если идея сложная или глубоко техническая. Эксперт не должен сомневаться в своих знаниях и уметь правильно отвечать на вопросы, и работа с несколькими источниками помогает и в том и в другом.

Простой способ сделать работу более эффективной — концентрироваться на одной задаче за раз, чтобы держать в уме один мысленный контекст. Например, Дональд Кнут читает сразу все электронные письма раз в несколько месяцев, но не обязательно прибегать к настолько экстремальной форме этого подхода. Люди психологически неспособны к многозадачности. В лучшем случае мы переключаем контекст, как однопроцессорная ОС, причем ценой замедления работы. Тем не менее, если все хорошо продумать и концентрироваться на задаче в течение небольшого фиксированного промежутка времени, такой подход позволяет делать перерывы без потерь в эффективности и избегать выгорания. Для больших проектов это единственная рабочая стратегия, а промежуточные задачи можно выполнять совместно с другими.

Формальное образование, пожалуй, наиболее полезно, когда вы пробуете учиться самостоятельно, что неизбежно, если речь идет о более сложных или специализированных темах. Как правило, вам приходится раздобыть несколько книг по интересующей теме и изучить их. В идеале книга должна быть:

- ◆ понятной — хорошо написанной, качественно объясняющей понятия и содержащей множество примеров и иллюстраций;
- ◆ релевантной — включать только полезный материал. Здесь особое внимание следует обратить на собственные исследования авторов, которые не подтверждены другими исследователями;
- ◆ полной — включать весь полезный материал по теме.

У качественных книг есть определенные отличительные черты:

- ◆ старые, классические книги с большим количеством хороших отзывов, как правило, хуже, чем новые, хорошо написанные книги. Все дело в том, что терминология, доступные технологии производства и аудитория с годами меняются. Сейчас предполагается, что современные читатели из-за нехватки времени должны иметь возможность быстро усвоить материал, а в старых книгах это не учитывалось. Предметная область тоже часто развивается и меняется;
- ◆ книги авторов программного обеспечения, как правило, более практичны, а книги, написанные преподавателями, более читабельны. Но легкость чтения не должна доводиться до крайности — лучше исходить из того, что читатель уже должен что-то знать по теме, чем писать совсем примитивно или превращать книгу в «краткое введение» или «практическое руководство» по API;
- ◆ английская докторантура, как правило, менее математическая, чем европейская. Это также отражается на содержании книги и стиле авторов;
- ◆ в *хороших* книгах авторы избегают сложных тем в угоду легкости чтения, а в *отличных* книгах сложные темы объясняются с достаточным количеством примеров и мотивации. С теоремами хочется разбираться, когда они применимы, поэтому мотивация и примеры помогают больше, чем доказательства.

Когда в нескольких книгах не находится интересующей вас информации, обычно дело не в том, что вопрос очевидный, а в том, что о нем мало известно, потому о нем и стараются вообще ничего не писать. Я сталкивался с подобным во многих различных областях, поэтому, как правило, лучше прочитать несколько последних книг и статей и не надеяться, что ответы найдутся в известных и часто цитируемых старых книгах, поскольку обычно это не так. Просто примите эти, казалось бы, противоречия и двигайтесь дальше.

Для экономии времени необходимо выбирать хорошие источники материала. Если вы нашли статью, прочитайте аннотацию, чтобы решить, стоит ли продолжать чтение. Если это книга, прочитайте оглавление, рецензии — например, в магазинах Amazon и Google, и оценки читателей (но имейте в виду, что среди всех критериев хорошей книги, оценки, как правило, отражают легкость чтения. А еще бывает, что профессионалы *намеренно* хвалят работу друг друга). Зачастую исследовательские работы содержат больше информации, чем книги, да к тому же они в основном доступны бесплатно. Но обычно тема развивается в последовательности статей, а потом выходит книга, в которой вся информация систематизируется.

## 1.20. О проектах всей книги

- ◆ Каждый нетривиальный фрагмент кода должен сопровождаться автоматизированным тестом.
- ◆ У каждой функции проверьте правильность констант и проверяйте входные данные с помощью утверждений.
- ◆ Убедитесь, что определения переменных файла заголовка являются статическими константами, чтобы избежать множественного определения при включении в несколько файлов \*.crr.

## 1.21. Список рекомендуемой литературы

- 1.1. Backhouse R. (2003). Program Construction. Wiley.
- 1.2. Bryant R., & David Richard O. H. (2015). Computer Systems: a Programmer's Perspective. Addison-Wesley.
- 1.3. Gries D. (1981). The Science of Programming. Springer.
- 1.4. Herlihy M., & Shavit N. (2020). The Art of Multiprocessor Programming. Elsevier.
- 1.5. Hyde R. (2020). Write Great Code, Vol. 2: Thinking Low-Level, Writing High-Level. No Starch.
- 1.6. Kao M. Y. (Ed.). (2016). Encyclopedia of Algorithms. Springer.
- 1.7. Müller-Hannemann M., & Schirra S. (Eds). (2010). Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice. Springer.
- 1.8. Smullyan R. M. (2014). A Beginner's Guide to Mathematical Logic. Dover.
- 1.9. Stepanov A. A., & MacJones P. R. (2019). Elements of Programming. Semigroup Press.
- 1.10. Wikipedia (2013). Master theorem. [https://en.wikipedia.org/wiki/Master\\_theorem](https://en.wikipedia.org/wiki/Master_theorem). Accessed July 30, 2013.

## 2. Основы разработки программного обеспечения

### 2.1. Введение

Создание алгоритма, структуры данных или большой программы в принципе ничем не отличается от создания чего-либо еще:

1. Решите, что вам нужно.
2. Напишите код.
3. Проверьте, что он выполняет задачу.
4. Исправьте ошибки.

Такой порядок действий хорош для учебных проектов, но не для реальной разработки, в которой встречаются свои сложности:

- ◆ это только начало — нужно еще как минимум развернуть код в системе клиента;
- ◆ если вы допустили несколько ошибок, то ваша работа уже не тянет на 5 с плюсом. Клиенты уйдут и, возможно, подадут на вас в суд вместе с третьими лицами, такими как государственные регулирующие органы;
- ◆ вся работа выполняется командами — разные люди принимают решения, пишут код и, возможно, даже проверяют и развертывают его. Нужны методы, позволяющие облегчить этот процесс для всех.

### 2.2. Обзор цикла разработки

Для работы необходим отлаженный процесс взаимодействия *разработчиков, менеджеров и клиентов* с учетом того, что у них у всех противоречащие потребности, — они пытаются максимизировать экономическую ценность и работать эффективно:

- ◆ клиенты заказывают новые системы и новые функции для существующих систем;
- ◆ менеджер по продукту решает, какие из них стоит внедрить, и уточняет требования;
- ◆ разработчик пишет код и определяет, что из заданного осуществимо, сколько это займет времени и как повлияет на многие другие обстоятельства.

Тогда цикл разработки принимает следующий вид:

1. Менеджер по продукту формулирует требования, получая от клиентов информацию о том, что им нужно, а от разработчика — что технологически реализуемо.
2. Разработчик проектирует высокоуровневую структуру компонентов, разбивая работу на задачи. Это согласовывается с менеджером по продукту и уточняется, соответствуют ли задачи потребностям.



3. Разработчик выполняет задачи одну за одной.
4. Разработчик тестирует код, а менеджер продукта проверяет его на соответствие требованиям.
5. Код выпускается.
6. Клиент проверяет, что функциональность соответствует требованиям.
7. Вышеперечисленные пункты повторяются по мере необходимости.
8. Код поддерживается и изменяется по мере выхода новых требований, обнаружения новых проблем и технологических изменений.

Этот процесс более подробно описан в оставшейся части главы, содержание которой основано на моем опыте, мнениях различных экспертов и многолетнем чтении литературы по программной инженерии. По каждому пункту можно найти множество ссылок, и некоторые из них я привел для вас в конце главы для дальнейшего чтения.

Из-за особенностей работы нашего разума, непредвиденных проблем от других вовлеченных сторон и значительного влияния небольших изменений кода на результат, для решения любых проблем требуется максимально быстрая и частая обратная связь. На этом принципе строится *agile-разработка программного обеспечения*. Популярным методом является *Scrum*, но команды почти всегда реализуют процесс по-своему, пропускавая или видоизменяя некоторые предусмотренные им действия. Как правило, обычно они сначала формулируют минимальные требования к функциональности, а затем по мере необходимости добавляют дополнительные функции. Этот подход эффективен, т. к. позволяет выбирать два параметра из трех: время, бюджет и полнота функциональности. Никогда не отказывайтесь от качества, если речь не идет о менее важных функциях, т. к. низкое качество реализации замедляет работу в целом.

Автоматизированные инструменты — такие как IDE, профилировщики, статический анализ, системы контроля версий, отладчики и автозаполнение кода — весьма полезны в работе разработчика. Для менеджеров продуктов наиболее важным инструментом, как правило, являются системы отслеживания задач/ошибок — такие как, например, Jira.

Наиболее важный навык разработчика — это способность абстрактно моделировать проблему. В какой-то степени он развивается со временем, из опыта и практики решения проблем. Разработчик также должен иметь в виду, что из двух рабочих программ одна может быть лучше другой.

## 2.3. Требования

Требуемая функциональность обычно определяется неформально или через *сценарии использования* — т. е. через описание процедуры взаимодействия пользователя с системой. Для работы над одной небольшой функцией также применимы частичные сценарии использования.

Обычно разработчику достаточно любого четкого описания требуемой функции. Зачастую это текст с простыми рисунками, размещенный в системе вроде Jira. Существует много книг по программной инженерии, посвященных специализированным системам нотации, таким как язык UML, который служит для описания классов или структур

компонентов, но применять их расточительно. Разработчики часто отклоняются от требований, а менеджеры по продукту инициируют изменения.

На требования часто влияют внешние факторы. Например, применение стандартов дизайна графического пользовательского интерфейса позволяет сделать взаимодействие с пользователем более продуктивным и последовательным. Разработчиков, естественно, интересует код, а не графический интерфейс, но пользователь-то видит именно его. Выполнять работу качественно легче, когда есть руководство по стилю или другие требования к ее качеству, и это касается не только графических интерфейсов. К тому же наличие стандартов снимает часть задач с менеджеров по продуктам, чтобы они могли сосредоточиться на других их аспектах.

Часто единственным источником документации служит руководство пользователя и инструкции или программы тестирования, да и они постоянно обновляются. Требования и проектные решения зачастую обсуждаются неофициально и аннулируются последующими изменениями. Люди, которые принимают решения, смутно помнят, что было раньше, и порой переходят в другие отделы или увольняются.

## 2.4. Проектирование структуры компонентов

Каждый компонент должен иметь свою ответственность и по максимуму повторно использоваться. Создаваемая *архитектура системы* должна иметь модульную, расширяемую структуру компонентов, позволяющую вносить изменения без существенных последствий для всего проекта или увеличения его сложности. Избегайте связывания и создавайте компоненты, которые можно тестировать по отдельности.

Большое значение имеет командная динамика:

- ◆ коммуникация внутри команды проще и эффективнее, чем между командами, поэтому структура компонентов стремится следовать структуре команды, а разные команды должны работать над независимыми функциональностями;
- ◆ необходимо четко разделять владение кодом, чтобы не столкнуться с трагедией общего пользования, когда никто не хочет очищать или перепроектировать общий код.

## 2.5. Проектирование отдельных компонентов и написание кода

Исследования разработчиков показывают, что они обычно работают *над одной небольшой проблемой за раз*, разбираясь в ней, придумывая начальное решение, проверяя его на некоторых входных данных и пересматривая до тех пор, пока оно не станет правильным. Вместо того чтобы следовать процессу «сверху вниз» или «снизу вверх», лучше мысленно создать некую «область работы», локализирующую небольшую проблему. Таким образом, хотя проектирование структуры компонентов обязательно осуществляется сверху вниз, для отдельных компонентов более важен гибкий процесс. Еще один распространенный подход — это ориентация на риск, т. е. работа над потенциально проблемным API или сложной частью интеграции.

Перед тем как начать разрабатывать систему, подумайте, что можно повторно использовать из предыдущих или похожих проектов или других доступных API. В большин-

стве нетривиальных проектов перед началом работы *требуется изучить* свойства рассматриваемых технологий, потребности пользователей и параметры среды развертывания. Даже выбрать API обычно сложно, т. к. вариантов много. Разработчик должен понимать базовую функциональность, учитывать любые неочевидные ограничения, снимать проблемы и достаточно хорошо знать, что происходит у API «под капотом». Часто разработчики кодируют функциональные возможности, не полностью понимая даже их API, — обычно это происходит потому, что для исследования нужен прототип, или нет времени, или реализация — это самый простой способ научиться.

Первым шагом в создании компонента является создание его API — общедоступного интерфейса. Создать хороший API сложно. В качестве общего правила следует проектировать его, как если бы он стал частью стандартной библиотеки языка. В частности, необходимо использовать интуитивно понятные имена для классов, функций, переменных и т. д., сохранять последовательность и избегать необоснованного использования «магических чисел» в качестве параметров. Приходящие на проект позже программисты чаще следуют соглашениям, принятым в кодовой базе, а не своим привычкам, и из-за этого плохие имена переменных и функций редко исправляются. Также следует избегать глубокой вложенности путем использования нескольких операторов `return`. API-интерфейсы должны проверяться вдвое тщательнее, потому что их сложнее изменить. Необходима хорошая объектная ориентация, но не следует закладывать слишком много функциональности «на будущее», т. к. большая часть такого «будущего» никогда не наступит. Кроме того, разработчики API должны создавать и поддерживать доступную документацию с достаточным количеством примеров кода для любого компонента, который будет повторно использоваться другими командами.

Обычно на чтение кода уходит гораздо больше времени, чем на его написание. Таким образом, инвестиции в удобство чтения непременно окупаются. Даже для понимания своего собственного кода разработчикам приходится реконструировать ментальную модель, а затем вносить в нее изменения. Если вы полагаетесь на конкретную документацию — например, веб-страницу, где объясняется редкое использование определенного API, поместите ссылку на нее в комментарии. Если таких случаев много, рассмотрите возможность создания кешированного репозитория документации для проекта. Это также облегчает введение в проект новых разработчиков. *Одноразовый код* сильно отличается от *постоянного кода*: первый предназначен для разовых задач, таких как научные эксперименты или классные задания, а второй необходимо поддерживать.

## 2.6. Шаблоны

Шаблоны (паттерны) — это своего рода рецепты по проектированию систем. Лежащие в их основе идеи обычно просты, а описание многих шаблонов вы можете найти в работе [2.2]. В литературе много внимания уделено реализациям шаблонов на языках, использующих наследование, тогда как в C++ шаблоны более эффективны. Наследование иногда является сложной концепцией — например, модифицируемый квадрат не является модифицируемым прямоугольником из-за несовместимости базового класса. Далее приведено краткое описание многих полезных шаблонов.

◆ *Архитектурные* — для организации компонентов системы:

- *слои* — присваивание компонентам уровней и вызов функции более низкого или того же уровня из высокоуровневых. Шаблон используется в каждой системе, где

высокоуровневый компонент бизнес-логики вызывает алгоритм или другие библиотеки;

- *конвейер* — преобразование данных с помощью последовательности компонентов. Каждый из них выполняет определенную работу над входными данными и передает результат далее. Так работает, например, связка из стиральной машины и сушилки;
- *модель-представление-контроллер* (model-view-controller, MVC) — отдельные компоненты обрабатывают данные, представления данных и их реакции на действия. Модель содержит данные и список подписанных представлений, которым она отправляет обновления. Контроллер манипулирует представлениями или изменяет данные для обработки запросов. Представления можно организовать в *конвейер преобразования* и отделить данные от их представлений. Также полезно группировать действия по объектам, на которые они воздействуют. Усовершенствованием шаблона MVC является *поток* (см. <https://facebook.github.io/flux/docs/in-depth-overview.html>):
  - контроллер переименовывается в *диспетчер*, который перенаправляет действия только на подписанные модели;
  - модель называется *хранилищем* и полностью отвечает за реагирование на действия, отправленные диспетчером;
  - в дополнение к лучшей объектной ориентации (т. к. хранилище выполняет часть работы контроллера) этот шаблон обеспечивает прямую связь. Таким образом, во многих хранилищах отправка действий друг другу через диспетчер более масштабируема, чем прямая связь в MVC;
- *доска объявлений* — несколько источников знаний обновляют данные в общем репозитории. Используется, например, когда несколько человек решают головоломку. Задействуется для задач, требующих сложной стратегии решения, — в основном для ИИ;
- *отражение* — взаимодействие с компонентами с помощью дополнительного интерфейса, скрытого от пользователей. Применяется для отправки команд конфигурации, сбора статистики использования и т. д.;
- *декларативная конфигурация* — компоненты инициализируются с помощью редактируемых файлов настроек, что делает развертывание более гибким;
- *реплицированные компоненты* — репликация самых важных компонентов, возможно, с использованием другого варианта проектирования. Компонент-менеджер скрывает репликацию от вызывающей стороны, отправляя запрос всем и возвращая первый или большинство ответов. Это позволяет выполнять параллельную обработку и отключать компоненты для проведения обслуживания;
- *объект контекста* — вместо сохранения состояния во время взаимодействия компонентов передается объект с состоянием. Позволяет не сохранять состояние;
- *безопасный интерфейс* — публичные методы не доверяют никому, а приватные — доверяют всем. Например, в случае параллелизма публичные методы запрашивают блокировки, а приватные — нет. В целях безопасности публичные методы проверяют авторизацию, а приватные — нет;

- *нормализация данных* — избегайте избыточности данных, чтобы свести к минимуму риск получения неверных и противоречивых данных. Например, отдавайте предпочтение дешевому перерасчету, а не дублированию;
- ◆ *шаблоны проектирования* — для проблем более высокого уровня:
  - *фабрика* — чтобы создать сложный объект с желаемыми свойствами, его создание инкапсулируется в отдельную функцию, изолирующую сложность;
  - *строитель* — если для построения объекта требуется несколько шагов, вместо конструктора с большим количеством параметров используйте объект, который позволяет создавать части и получать только полный результат;
  - *прототип* — чтобы скопировать сложный объект, создайте копирующий метод клонирования. В C++ использует конструктор копирования и оператор присваивания (конструктор перемещения необязателен);
  - *одиночка* — чтобы ограничить количество экземпляров объекта одним, создайте статическую функцию, которая возвращает его, выполняя построение при первом вызове. Одновременное использование должно выполняться с блокировкой. Имейте в виду, что такой объект лишь немногим лучше, чем глобальная переменная, и этот шаблон по праву имеет плохую репутацию. Во многих случаях лучше иметь один экземпляр, принадлежащий компоненту, и предоставлять доступ к нему другим — т. е. решать, кто является владельцем;
  - *команда* — действия выполняются объектами с методом `execute()`. В C++ это делает `operator()`. Используется для придания действиям состояния и делает их универсальными;
  - *стратегия* — чтобы изменить поведение во время выполнения, создается объект-член, вызывающий изменение. Для поведения, определяемого статически, используется код, а для динамического — указатель на командный объект;
  - *композит* — объект содержит список подобъектов, из которых он состоит, и операция над ним применяется к его собственным данным и каждому подобъекту. Например, чтобы отрисовать объект методом `draw()`, вызывается метод `draw()` для каждого содержащегося в нем подобъекта;
  - *фасад* — создается интерфейс для объединения нескольких связанных систем, возможно, с модифицированным API. Например, если необходимо вызвать несколько функций в определенном порядке, создайте одну функцию с наименьшим количеством параметров, выполняющую эти вызовы;
  - *прокси* — один объект подменяется другим. Например, `operator[]` для набора битов STL возвращает ссылочный объект, который можно построить из `bool` и присвоить ему;
  - *инкапсулированная реализация* — интерфейс и реализация находятся отдельно друг от друга. Реализация API хранится в секрете, и вы можете обновлять общую библиотеку без перекомпиляции вызывающей программы. В C++ используется *указатель на идиому реализации* (*pointer to implementation idiom, PIMPL*);
  - *итератор* — чтобы просмотреть все элементы в структуре данных, создается объект, который может эффективно получать доступ к элементам и перебирать их в нужном порядке;

◆ *шаблоны безопасности:*

- *авторизация* — передается объект с правами доступа ко всему, что проверяет авторизацию. Например, если пользователь вошел в систему, объект содержит данные для входа;
- *брандмауэр* — запросы проверяются на безопасность перед передачей приложениям. Компонент брандмауэра проверяет то, что не учитывается во время авторизации. Например, он может отбрасывать запросы, которые слишком часто поступают от одного клиента;
- *доступ на основе ролей* — создаются роли, каждая из которых имеет ограниченный набор необходимых привилегий. Это упрощает управление привилегиями и обеспечивает безопасный доступ по мере необходимости;

◆ *шаблоны параллелизма и распределения:*

- *брокер* — каждый компонент взаимодействует с другими через прокси. Например, приложения, которым требуется доступ к сети, используют маршрутизатор. Это сокращает количество подключений и разделяет сетевые функции;
- *реактор* — поток ожидает события и помещает их в одну из нескольких очередей, тогда как другие потоки последовательно обрабатывают события в каждой очереди. Обеспечивает гибкую одновременную обработку;
- *проактор* — запросы асинхронны, обработчик ожидает результатов, идентифицируемых уникальными токенами завершения;

◆ *шаблоны управления ресурсами:*

- *получение ресурса* — это инициализация (resource acquisition is initialization, RAII). Заключается в получении ресурсов в конструкторах и освобождении их в деструкторах. Обеспечивает освобождение ресурсов в случае исключений и использования множества `return`. В C++ применяется с оператором `{}`;
- *поиск* — служба каталогов отслеживает доступные службы. Например, можно зарегистрировать службы на сетевых маршрутизаторах, чтобы сделать запросы на маршрутизацию более эффективными;
- *пул ресурсов* — хранение кеша ресурсов, часть из которых могут быть арендованными. Относится к приведенным далее шаблонам;
- *ленивое получение* — доступ к ресурсу организуется в последний момент и освобождается как можно быстрее. Максимально увеличивает доступность ресурсов для других приложений;
- *жадное получение* — получение ресурсов на старте. Повышает надежность, поскольку доступ к ресурсам сразу ограничивается и защищается;
- *частичное получение* — доступ к ресурсам осуществляется поэтапно. Например, онлайн-видео буферизуется и начинает воспроизводиться еще до полной загрузки;
- *аренда* — выдача ресурсов на ограниченное время и отзыв доступа по его истечении. Например, отмена авторизации для неактивных пользователей;
- *подсчет ссылок* — любой ресурс сохраняет счетчик и освобождает его, когда он обнуляется. Полезно при доступе к ресурсу из нескольких мест и использовании

динамической памяти. В С++ это делается с помощью *интеллектуального указателя*.

◆ *идиомы программирования* — это шаблоны, специфичные для языка. Например:

- цикл в обратном порядке с заданными начальными и конечными итераторами (массивы С++ массивы позволяют указывать на элемент, следующий за последним, но не на элемент, предшествующий первому):

```
From iter = end to iter ≠ begin
    --iter
    Loop work
```

- чтобы объявить глобальную константу — такую как  $\pi$ , в большинстве языков заводится чистая функция, которая ее возвращает;
- в С++, чтобы определить операторы, которые принимают шаблонные аргументы одного и того же типа, их делают функциями `friend` такого типа, — метод, часто используемый в этой книге;
- в С++ массивы можно передавать синтаксисом `C` ради общности, но это снижает безопасность типов, поэтому, возможно, лучше так не делать;
- публичные методы используют публичные интерфейсы — например, без объявлений частных типов;
- в языке без ссылок, вроде Java, объект результата можно передавать в качестве аргумента, и функция изменит свои поля данных;
- в целях обеспечения безопасности исключений во время операции не изменяйте состояние до тех пор, пока результаты работы потенциально проблемного кода не будут защищены (проблемным кодом считается, например, выделение ресурсов или некоторые параллельные вычисления);
- если константная функция вызывается как константными, так и неконстантными функциями, по мере необходимости приведите результат к константе.

Шаблоны проявляются во многих областях. Например, при работе со временем:

- ◆ невозможно преобразовать будущее время в отметку времени — процесс зависит, например, от решения правительства отказаться от перехода на летнее время, поэтому следует использовать время UTC;
- ◆ задания, запланированные перед переходом на летнее время, могут выполняться дважды или вообще не выполняться.

## 2.7. Управление ошибками

Работа компонента может *нарушиться* из-за:

- ◆ ошибки в коде;
- ◆ отказа зависимого компонента — такого как база данных, сетевое соединение или система выделения памяти;
- ◆ задержки из-за обслуживания слишком большого количества запросов;
- ◆ *ошибки модели вычислений*, связанной с повреждением памяти или сбоем питания.

Чтобы избежать нарушений работы важной системы или уменьшить вероятность их возникновения, рассмотрите следующие методы:

- ◆ ограничение влияния ошибок;
- ◆ сообщение об ошибках вызывающим абонентам;
- ◆ исправление ошибок;
- ◆ приведение системы в исправное состояние.

Для критических компонентов следует разработать процедуры управления любыми типами ошибок. Далее приведены соответствующие шаблоны, описанные в работе [2.2]:

- ◆ *проверка работоспособности* — проверка всех предварительных условий, возможных инвариантов и постусловий. Сводит к минимуму влияние ошибки на состояние компонента и помогает выявлять ошибки. Для сложных алгоритмов некоторые проверки могут оказаться невыполнимыми из-за необходимости пересчета;
- ◆ *трассировка выполнения* — критические вычисления ведут сводку входных данных, инвариантов и использованных ресурсов. Чаще всего отслеживается потребление (сколько ресурсов используется), насыщение (сколько мощностей осталось) и ошибки. Это позволяет обнаруживать источники ошибок и динамически настраивать компоненты. Например, менеджер компонентов может перезапустить компонент, создавший трассировку в начале вычисления, но не создавший трассировку по его завершении через некоторое время;
- ◆ *известное хорошее состояние* — создайте способ перевести систему в такое состояние. Перезапуск устраняет большинство проблем в большинстве систем. Держите предыдущую версию компонента наготове для развертывания на случай сбоя новой;
- ◆ *избыточные данные* — позволяют с высокой вероятностью избежать ошибок модели вычислений. Например, если в банке случится сбой питания после списания со счета источника, но до зачисления на счет получателя, без избыточности кто-то потеряет деньги;
- ◆ *выделенные компоненты* — заведите несколько копий компонентов, каждая из которых служит определенной части функциональности. Отказ копии влияет только на ее часть;
- ◆ *несколько компонентов* — работа продолжается, если несколько компонентов перестают отвечать. Компоненты должны быть отдельными процессами на разных компьютерах и в разных местах. Это также помогает при развертывании, поскольку позволяет переносить пользователей со старой версии на новую, постепенно увеличивая процент запросов к новой версии;
- ◆ *репликация компонентов* — позволяет избежать случайных ошибок. Используется в системах, которые не должны ошибаться, — например, в системе управления самолетом. Благодаря трем независимо разработанным компонентам выбор ответа большинства существенно снижает вероятность ошибок из-за обработки одного непредсказуемого отказа;
- ◆ *ограничение функциональности* — переводите неисправный компонент в режим ограниченной функциональности, в котором он выполняет только критические операции. Это позволяет предотвратить будущие ошибки и повысить производительность.



Компонент считается *надежным*, если он эффективно обрабатывает нарушения пред-варительных условий.

Для тестирования надежности часто используется шаблон *внедрения отказов* (*хаос-инжиниринг*). В рамках этого шаблона вы сознательно отключаете несущественные или избыточные части системы, чтобы увидеть, сохраняется ли основная функциональность, и если нет, проверяете, как ведут себя механизмы трассировки, аварийных сигналов и восстановления, что позволяет оценить работу в случае реального сбоя. Например, для некоторых функций финансового трейдинга такие проверки в соответствии с государственными постановлениями должны выполняться два раза в год. Созданная Netflix система Chaos Monkey случайным образом отключает те или иные части сети и работает постоянно. Но обычно требуется научный подход, при котором система вручную анализируется на предмет потенциальных проблемных зависимостей, после чего удаляются несущественные компоненты, ошибочно считаемые важными. Также хорошо бы обеспечить независимый, а не каскадный отказ компонентов. Даже если проблем не возникает, это хороший способ научить разработчиков справляться с потенциальными проблемами в будущем.

Вместо отключения компонентов можно также замедлить выдачу ответов, чтобы оценить их влияние на общую скорость системы. В частности, вы получите информацию о том, где в системе возникают узкие места при увеличении объемов запросов. Обычно для этого нужны специальные инструменты.

## 2.8. Тестирование

Непротестированный код часто содержит ошибки. Тестирование подразумевает запуск алгоритма с подмножеством возможных входных данных, чтобы проверить, удовлетворяют ли выходные данные контракту. Если в это подмножество не входят все возможные входные данные, тестирование не гарантирует отсутствие ошибок, но увеличивает уверенность в их отсутствии, а заодно позволяет проверить наличие проблем с производительностью.

Разделите входные данные на классы так, чтобы для любого условия контракта существовал репрезентативный класс, а внутри класса входные данные выбирались случайным образом. Если это сложно или неэффективно, используйте случайные входные данные для создания большего охвата. И не страшно, если каких-то редких данных в течение конечного времени тестирования не появится.

*Модульное тестирование* проверяет отдельные компоненты, а *интеграционное тестирование* — работу всей системы. Теоретически последнего достаточно, чтобы охватить все сценарии, но первое позволяет быстрее обнаружить ошибки, потому что системные тестовые случаи могут пропустить трудную для вызова логику компонента. *Автоматическая регрессия* запускает систему на множестве входных данных. Созданный набор тестов обычно разрастается, начав с простого *пакета тестов разработки*, который проверяет, работает ли основная функциональность, и превращается со временем в полный набор регрессионных тестов для проверки разных сценариев использования. Важна и возможность автоматически проверять определенные свойства полученных результатов — такие как правильность и предполагаемая эффективность, чтобы уверенно ставить по результатам теста оценку «пройден/не пройден».

Лучше всего добавлять тесты сразу после написания кода, потому что вы сможете взять проверенные примеры и преобразовать их в автоматизированные тесты, не успев забыть, как работает эта функция, и переключиться затем на другую задачу. Добавление тестов в существующий код следует выполнять постепенно, по одному тесту за раз, т. к. даже тривиальный тест лучше, чем отсутствие теста, — хотя бы потому, что это шаг в правильном направлении.

*Тестирование методом «черного ящика»* — это проверка системы без знаний о ее реализации, а в тестировании *методом «белого ящика»* они используются для выбора входных данных. Для больших систем последнее невозможно, потому что большой объем сложного кода может реализовывать простую бизнес-логику. Для небольших алгоритмов обычно производится тестирование по методу «белого ящика», т. к. требуется проверять граничные случаи — например, самые маленькие и самые большие входные данные, которыми разработчики обычно пренебрегают.

При отладке алгоритма рассмотрите возможность использования автоматизированных инструментов для просмотра трассировки стека и вывода значений переменных. Также часто используют следующие распространенные стратегии:

- ◆ *бинарный поиск* — выводить операторы «привет» в определенных точках кода, пока не найдется строка, которая приводит к сбою;
- ◆ *продолжение* — когда есть правильная и неправильная программы, схожие между собой и оснащенные одинаковыми тестами, внесите последовательность изменений, чтобы превратить правильную в неправильную, и выполняйте тестирование после каждого из них. Первое изменение, которое приведет к поломке, обычно содержит ошибку.

Тестирование может с высокой степенью достоверности гарантировать отсутствие ошибок. Если выполнение варианта использования дает значение потери 1 в случае сбоя и 0 в противном случае, после  $n$  правильных вариантов вероятность увидеть проблему становится равной  $\leq 3/n$  с достоверностью 95% (см. главу 21. *Вычислительная статистика*). Алгоритмы тестирования, которые в качестве результата выдают вещественные числа, устроены сложнее, потому что ответы меняются от запуска к запуску, но все еще являются правильными. Здесь следует определить безопасный диапазон правильного ответа — например, необходимую точность. Генераторы случайных чисел не проходят тесты, если частота отказов механизма экспоненциально высока.

К *производственному коду* обычно применяется обширное тестирование. Недостаточно при этом проверять все граничные случаи и перехватывать исключения при неправильных входных данных и других ошибках в коде. Производственный код используется многими вызывающими сторонами и не должен содержать ошибок. В нем также должны быть предусмотрены «хуки» для отладки (шаблон «отражение») и возможность логирования выполняемых вычислений для предполагаемой отладки в будущем. Процесс совершенствования почти никогда не завершается полностью, и всегда находится, что можно улучшить. Для развертывания в рабочей среде необходимо понять, достаточно ли хорош код, соблюдены ли определенные стандарты качества, пройдено ли тестирование и проверка кода.

Еще одна полезная техника — *фаззинг*. Это тестирование программы со случайными входными данными с ограниченной проверкой результатов. Важно, например, убедиться, что никакие входные данные не вызывают сбоев. Поэтому хочется, чтобы код про-

ходил как можно больше проверок. С помощью специализированных инструментов можно обнаруживать и более сложные ситуации — типа нехватки памяти. Большинство таких инструментов также позволяют сохранять проблемные входные данные и создавать из них отчеты об ошибках.

Работая с наследуемым кодом, можно начать с *тестирования характеристик*. Для этого надо сначала запустить систему, чтобы получить некоторый набор результатов. Затем проверить, что любые изменения, будь то рефакторинг или обновление компонентов, приводят к таким же результатам. В конце концов такие тесты превращаются в автоматизированные регрессионные тесты, после чего код может превратиться из наследуемого в актуальный.

Используемые тесты также являются частью документации. В идеале они выполняются после каждого изменения кодовой базы и должны всегда быть актуальны. Любые комментарии и документацию с примерами можно превратить в тесты, проверяющие ожидаемый ответ. Затем тесты можно использовать для автоматической генерации документации.

## 2.9. Просмотр кода

Прежде всего ожидается, что разработчик сам проверяет свой код. Часто для этого используются какие-то рутинные или новые автоматические тесты. После этого код регистрируется в системе контроля версий — такой как *git*. Но перед добавлением в проект код должен быть проверен кем-то еще.

Просмотр кода — это очень эффективный способ проверки. Другой разработчик смотрит на разницу между новым и существующим кодом в проекте и решает, принять ли эти изменения. Они могут быть отклонены по многим причинам, например:

- ◆ ошибки, не обнаруженные при тестировании;
- ◆ отсутствие ясности в коде или в комментариях;
- ◆ существует более удачный способ выполнить задачу;
- ◆ несоответствие стиля или стандарта кодирования.

Я считаю, что одного просмотра кода достаточно, хотя во многих проектах требуется проводить его несколько раз.

## 2.10. Релиз

Релиз (выпуск) новых функций — важнейшая задача, и выполнять ее надо правильно, — это ведь не просто сдача домашнего задания в школе. После тестирования со стороны разработчика и проверки кода программное обеспечение разворачивается из основного репозитория в *тестовой среде* и тщательно проверяется разработчиками, специальными тестировщиками, менеджерами по продуктам, а иногда даже клиентами. Просмотр кода и регулярное тестирование всех функций позволяют существенно снизить количество ошибок. Если вы заметите ошибку, просмотрите кодовую базу на предмет схожих ошибок — скорее всего, вы найдете хотя бы одну такую же ошибку, которая появилась из-за «копирования/вставки».

Следующий шаг — релиз кода для клиентов. Вы должны сообщить всем заинтересованным сторонам о выпуске и дать четкие инструкции о том, как вернуть все к прежнему состоянию в случае возникновения проблем. В некоторых компаниях этот процесс может сопровождаться аудитом, особенно если программное обеспечение является частью регулируемого государством рабочего процесса — например, в сфере торговли. Любые критические изменения вносить проблематично, потому что они дорого обходятся пользователям, хотя и бесплатны или даже полезны для разработчиков.

Важно не выпускать ПО для всех сразу. При первом выпуске стоит охватить лишь часть клиентов или особую группу клиентов. Затем еще несколько и т. д. Также можно поэтапно загружать программу на разные компьютеры в кластере, где работает ваше программное обеспечение. Это полезно во многих отношениях:

- ◆ пропущенная ошибка коснется лишь небольшого процента пользователей. А поскольку ошибки пропускаются часто, такой подход позволяет снизить репутационные риски;
- ◆ правила могут требовать поддержания определенного качества обслуживания — например, в случае локальных отключений электроэнергии и т. п. Таким образом, любые сбои из-за ошибок будут иметь ограниченное влияние;
- ◆ во многих системах есть механизмы аварийного переключения, при которых отправленный сервису сигнал посылается на другой сервер, если на него не ответили. В такой ситуации ошибка, из-за которой сервер упал, приведет к падению всего кластера, если выпустить новую версию везде одновременно.

В случае с программным обеспечением, которое пользователи загружают и запускают локально, поэтапный выпуск реализовать сложнее, но все же можно организовать его одним из следующих способов:

- ◆ создание стабильных и последних версий — многие проекты с открытым исходным кодом так и делают;
- ◆ выбор случайной версии в зависимости от известной информации о пользователе — такой как его IP-адрес;
- ◆ сохранение в программе и новых, и старых функций, позволяя ей случайным образом решать, какие из них использовать. Возможно, этот выбор должен производиться во время установки.

Многие компании создают группы *инженеров по надежности* (site reliability engineering, SRE), которые управляют процессом выпуска вместо разработчиков. Благодаря наличию автоматически создаваемых отчетов об ошибках и сбоях некоторые из задач выпуска могут выполняться даже автоматически путем развертывания на большем количестве клиентов каждые несколько дней, если в предыдущем выпуске не сообщалось об ошибках. В книге [2.1] подробно описаны многие аспекты SRE.

## 2.11. Обслуживание

Текучка кадров среди разработчиков, отсутствие документации и устаревшие технологии затрудняют поддержку систем. Некоторые системы, если у них простая бизнес-логика, часто полностью переписываются. Но обычно программные системы за долгие годы разработки становятся большими. Обычно это естественный процесс, а не резуль-

тат заранее продуманного проектирования. Код изменяется в результате приобретения новых знаний и замены текущих реализаций компонентов более качественными. Возможности улучшения инфраструктуры и пересмотр требований часто делают будущие разработки более эффективными, и в них необходимо инвестировать для обеспечения устойчивости программ. Обновление функциональности, особенно API, затрудняется из-за закона *Хайрама*: «любое действие непременно повлияет на кого-либо». Так, часто невозможно избежать недовольства отдельно взятых пользователей, когда поддерживать их любимую функциональность становится нерентабельно.

Переписывать всю систему проблематично, потому что *некоторые важные сценарии использования существуют только в коде* и вызываются слишком редко, чтобы на них вообще обратили внимание. Миграцию баз данных особенно сложно реализовывать, потому что данные можно неправильно изменить, закодировать для обхода системных ограничений, забыть обновить до актуального состояния или проверять с помощью специальной и никому не известной логики исправления ошибок. *Автоматическая регрессия* всех вариантов использования — лучший способ избежать ошибок, хотя при этом могут быть пропущены труднотестируемые варианты.

## 2.12. Оценка

Точная оценка того, сколько времени займет та или иная работа, попросту невозможна из-за уникальных черт каждого проекта, которые выявляются только во время проектирования или реализации. Чтобы получить приблизительную оценку, обычно лучше всего взять медиану оценок нескольких достаточно компетентных разработчиков (математическое обоснование этого подхода приведено в *разд. 25.13. Смещение, дисперсия и бэггинг*). Чтобы повысить точность оценки, проведите обсуждение и выявите причины различий. Иногда предпочитают применять *стори-пойнты* — стратегию использования менее точной категории «размера футболки» (маленький, средний, большой, очень большой) вместо более точной и обзывающей оценки.

Новые функции планируются в зависимости от бизнес-приоритетов — например, количества пользователей, которым это выгодно, и внешних требований. Даже при agile-разработке часто требуется долгосрочное планирование — к примеру, когда юридические требования не допускают продления сроков. Хотя функциональные преимущества часто недооцениваются, основанное на нем соотношение затрат и результатов и оценка ресурсов разработчиками позволяют лучше понять, какую функцию развивать следующей.

## 2.13. Следование формальному процессу

Полезные мероприятия по развитию приносят пользу. Однако следует *устранить наиболее расточительные* из них. Обычно их легко выявить, потому что полезные активы прошли валидацию — например, тестирование, проверку кода, документирование, инструменты автоматизации и т. д. Но иногда трудно сказать, является ли действие полезным, поскольку это может зависеть от конкретного проекта. Таким образом, процесс разработки должен быть рекомендательным, а не предписывающим. Тем не менее некоторая степень формальности повышает производительность, поэтому согласовывайте процесс, следуйте ему и по возможности ищите способы его улучшить. Как минимум

стоит автоматизировать все стандартные процедуры — такие как сборка и развертывание тестовой среды.

## 2.14. Управление данными пользователя

Многие приложения, особенно мобильные, содержат персонализированные функции, требующие использования данных клиентов. В этих случаях необходимо учитывать естественные и юридические вопросы конфиденциальности. Основная идея здесь — использовать как можно меньше данных:

- ◆ не запрашивайте ненужные данные;
- ◆ не храните данные, которые пользователь может легко ввести повторно;
- ◆ дайте пользователю понять, какие данные собираются, как они хранятся и т. д.

Полезно классифицировать пользовательские данные на:

- ◆ *конфиденциальные* (например, пароли и номера кредитных карт) — не собирайте их до тех пор, пока они не понадобятся, и храните их в безопасном месте;
- ◆ *персональные* — например, имя, геолокация, определенные личные предпочтения, которые позволяют пользователю выбирать те или другие функции;
- ◆ *нечувствительные* — например, IP-адрес, настройки приложения. Их можно использовать свободно, сохраняя при этом конфиденциальность.

## 2.15. Список рекомендуемой литературы

- 2.1. Beyer B., Jones C., Petoff J., & Murphy N. R. (2016). Site Reliability Engineering: How Google Runs Production Systems. O'Reilly.
- 2.2. Buschmann F., Henney K., & Schmidt D. (2007). Pattern-oriented Software Architecture Vol. 4: A Pattern Language for Distributed Computing. Wiley.
- 2.3. Hoffman D. M., & Weiss D. M. (Eds). (2001). Software Fundamentals: Collected Papers by David L. Parnas. Addison-Wesley.
- 2.4. Humble J., & Kim G. (2018). Accelerate: The Science of Lean Software and Devops: Building and Scaling High Performing Technology Organizations. IT Revolution.
- 2.5. Poppendieck M., & Poppendieck T. (2003). Lean Software Development: An Agile Toolkit. Addison-Wesley.
- 2.6. Winters T., Manshreck T., & Wright H. (Eds). (2020). Software Engineering at Google: Lessons Learned from Programming Over Time. O'Reilly.

## 3. Советы по вопросам карьерного роста и прохождения собеседований

Лучший вопрос на собеседовании — попросить кандидата написать письмо бабушке.  
*Александр Степанов, из презентации «Советы юному программисту»*

### 3.1. Введение

Поможет ли эта книга успешнее пройти собеседование? Если у вас есть несколько месяцев на изучение ее глав, вплоть до *главы 11. Алгоритмы графов*, однозначно да. Если же вы располагаете лишь несколькими неделями или днями, лучше откройте работу [3.3] и прочитайте ее не менее трех раз.

То, что я рассказываю в этой главе, основано главным образом на моем собственном опыте, но многое также взято из других книг и многочисленных постов в онлайн-блогах.

### 3.2. Отклик на вакансию

Начинающий вы или более опытный специалист — процесс будет одинаковым. Не совершайте ошибки, обращаясь только в одну компанию. Лучше выбрать несколько компаний и провести небольшое исследование: заглянуть на веб-сайты компаний (для получения основной информации), посетить сайты **glassdoor.com** (где можно узнать степень удовлетворенности их сотрудников) и **level.fyi** (статистика зарплат), а также Википедию (как источник непредвзятой информации). В идеале компания должна производить перспективный и интересный для вас основной продукт и хорошо относиться к своим сотрудникам в плане оплаты труда и управления. Как правило, крупные компании могут позволить себе обучать новых сотрудников, а стартапам нужны очень быстро обучающиеся или уже знакомые с конкретными технологиями специалисты. На сайте практически любой компании имеется раздел с вакансиями, куда можно подавать заявки, хотя некоторые компании, особенно хедж-фонды, просят напрямую отправлять электронные письма в отдел кадров.

Найти порядка двадцати более или менее подходящих компаний не трудно. Кое-кто вам вовсе не ответит, в других вы провалите собеседование, а некоторые не предложат ничего интересного. Но вы также должны получить пару хороших предложений, из которых можно будет выбрать. Кроме того, ваши первые собеседования станут хорошей подготовкой к остальным.

### 3.3. Составление резюме

Предположим, вы нашли компанию, в которой хотели бы работать, подали заявку, и после разговора с рекрутером вас пригласили на собеседование. Цель компании —

отличить подходящего кандидата от неподходящего и продать позицию. Интервьюеры оценивают ваши способности решать задачи, умение программировать, навыки общения, общую мотивацию и интерес к работе компании.

Процесс начинается с резюме. Приведу несколько советов, которые помогут написать хорошее резюме:

- ◆ посмотрите в Интернете примеры хороших и плохих резюме, чтобы сформировать о них общее представление. Если в вашем университете есть семинар или услуга по составлению резюме, можно воспользоваться ими (у меня тоже был такой плохонький, но все же в какой-то мере полезный семинар);
- ◆ следуйте стандартному шаблону: заголовок с контактными данными, образование, стажировки, специальные проекты, навыки. Интервьюеры и рецензенты резюме хотят увидеть в нем хороший средний балл в университете и несколько проектов по программированию, чтобы во время собеседования их можно было обсудить. Для старших разработчиков опыт важнее образования;
- ◆ не пишите о своих целях — в их описании легко ошибиться, а пользы будет мало;
- ◆ трижды проверьте орфографию и грамматику — если в вашем резюме есть ошибки, они будут и в вашем коде. Следите за чистотой шрифтов и форматирования — однажды я видел резюме, написанное комбинацией шрифтов Times New Roman (старый вариант по умолчанию в MS Word) и Cambria (новый вариант по умолчанию). Не следует разукрашивать макет резюме различными «бантиками» — стремитесь обеспечить простоту его чтения;
- ◆ не делайте упор на конкретные языки или технологии. Компаниям нужны специалисты по решению задач, и вы можете попасть впросак, если заявите, что являетесь экспертом в какой-то области, а это окажется не так. Большинство кандидатов с длинным списком языков программирования имеют практические знания лишь одного или двух, а к остальным обращались пару раз для создания небольших проектов;
- ◆ не указывайте не относящийся к делу опыт — такой как работа в ресторане колледжа. В пунктах про образование и предыдущие места работы не включайте занятия и проекты, которые не заслуживают хотя бы нескольких минут беседы;
- ◆ профессиональные услуги по написанию первого резюме могут стоить денег, но затем вы сможете обновлять его самостоятельно;
- ◆ встречайтесь с друзьями и просматривайте резюме друг друга. Обратная связь всегда полезна.

## 3.4. Поведенческие вопросы и собеседования

Обычно собеседование начинается с обсуждения вопросов опыта или образования кандидата. У опытных кандидатов основное внимание уделяется первому, а у недавних выпускников — второму. Выпускникам полезно проходить стажировки, потому что их можно обсудить во время собеседования, а заодно они помогут подготовиться к вопросам, связанным с методиками командной разработки программного обеспечения. В резюме должен быть четко прописан любой предыдущий опыт. Не знать чего-либо — это нормально, но нужно уметь все объяснить. Следует убрать из резюме опыт, который не



имеет значения или не годится для обсуждения. Обычно я прошу описать сложную часть «вашего любимого» проекта, но многие интервьюеры выбирают любой проект. Если вы не носитель языка, помните, что общение — это в первую очередь способность объяснять что-либо другому человеку логически, а не идеальное владение языком.

Популярной техникой для описания опыта является STAR (Situation, Task, Action, Results) — опишите ситуацию, поставленную задачу, предпринятые вами действия и полученные результаты. Кандидат должен понимать и уметь использовать технологию решения хотя бы в виде «черного ящика». Не допускайте раскрытия информации о проектах, которая является коммерческой тайной, т. к. это незаконно и создает впечатление вашей безалаберности. Иногда бывает сложно сдержаться, когда на собеседовании просят рассказать подробности.

Многие компании проводят для опытных кандидатов *поведенческие собеседования*. Это следующий шаг по сравнению с простыми открытыми вопросами, связанными с проектом. Вас могут попросить: «Опишите свой опыт в ситуации  $x$ », где  $x$  — это обычно ситуация, в которой раскрываются ваши лидерские качества — например, обращение к другой команде. Но может быть и менее приятная ситуация — например, обработка отрицательной обратной связи. Чтобы хорошо ответить на такие вопросы, нужно:

- ◆ иметь под рукой список ваших самых впечатляющих проектов за последние годы, о которых есть что рассказать, не раскрывая тайн;
- ◆ быть готовым признать мелкие ошибки и объяснить, какой урок вы из них извлекли;
- ◆ быть честным и стоять на своем, когда вам бросают вызов. Например, однажды меня спросили, почему я выполняю какую-то второстепенную работу вместо основной работы над проектом, на что я ответил, что по возможности уделяю небольшой процент времени от основного проекта второстепенным, чтобы они тоже продвигались.

Имеет смысл так выстраивать свою карьеру в компании, где вы сейчас работаете, чтобы быть готовым отвечать на поведенческие вопросы в будущем. Вы можете попробовать достичь максимального эффекта от работы, участвуя в создании наиболее важных проектов, — тех, что приносят максимальную пользу при минимуме усилий. Также имеет смысл предпочесть более крупные проекты более мелким, потому что о них легче рассказывать. Все это в совокупности повысит будущую оценку вашей производительности в компании, где вы сейчас работаете.

Вам также могут задать вопросы по базовым знаниям. Как правило, они разумны и основаны на том, что кандидат должен знать или заявляет, что знает. Например, любого кандидата можно спросить о выделении памяти в стеке или в куче. У тех, кто проходил курс по операционным системам или имеет опыт многопоточного программирования, можно спросить про дедлоки. Типичные вопросы звучат примерно так:

- ◆ сравните любое сбалансированное дерево и любую хеш-таблицу с точки зрения пользователя — например, на примере реализации встроенного словаря Python или JavaScript;
- ◆ опишите, что происходит при запуске рекурсивной функции, которая выделяет целое число в куче и вызывает себя без базового условия.

## 3.5. Технические собеседования

Большинство компаний больше любит кандидатов с сильным критическим мышлением и базовыми знаниями, чем кандидатов с богатыми знаниями и средним критическим мышлением, потому что разработка программного обеспечения — это постоянное обучение. Поэтому основная часть собеседования посвящена техническим вопросам. Основные типы вопросов:

- ◆ написание кода — реализовать простой алгоритм. Правильный ответ демонстрирует навыки программирования и базовые способности решения задач;
- ◆ головоломки — в идеале чисто математические, а не нестандартные. Головоломки позволяют глубже проверить навыки решения задач.

В последнее время наблюдается тенденция задавать только вопросы по написанию кода, а некоторые компании даже явно требуют проводить собеседования именно так.

Коммуникативный аспект гораздо важнее, чем кажется. Интервьюеры стараются побудить вас обсудить проблему и задать уточняющие вопросы, иногда даже намеренно опуская важные детали. Многие кандидаты берутся за решение задачи, не до конца понимая ее требования. Крайне важно сначала загрузить задачу в свой разум. Учитывайте всю информацию, которую услышали, потому что обычно вся она и нужна для решения проблемы.

Существует хороший психологический подход под названием «солдат/ученый». Военные организации выработали стандарт профессионального поведения: быть честным, следовать указаниям в меру своих возможностей, сообщать о неудачах и быть уверенным в себе. То же самое и у ученых, которые совместно работают над решением задач, оценивают работу других членов команды (интервьюер может делать так же) и отказываются от собственных идей в пользу лучших на текущий момент. В качестве противоположного примера можно привести продавца, который защищает свой продукт (в рассматриваемом случае — решение задачи), несмотря ни на что.

Для решения большинства вопросов достаточно это иметь четкую инвариантную часть решения. Не выдавайте непроверенный набросок кода в виде готового решения. Сперва мысленно протестируйте его на нескольких примерах.

Также обратите внимание на API и постарайтесь обработать крайние случаи. Если это проблематично, уточните у интервьюера требования, но не игнорируйте проблему.

Интервьюеры используют одни и те же вопросы или берут их из различных источников. Изучите такие источники, чтобы знать, чего ожидать. Лучше подготовленные кандидаты обычно добиваются большего успеха. Подготовка к собеседованию похожа на подготовку к SAT в старшей школе — она помогает получить более высокий балл. Но не стоит надеяться, что вам когда-нибудь зададут вопрос, который вы уже слышали. Я был на множестве собеседований, и такого никогда не случалось. Но похожие вопросы, к которым можно применить уже известные вам методы решения, вполне могут быть.

Эффективные вопросы обладают решающей силой — подобно статистическим тестам (см. главу 21. *Вычислительная статистика*). То есть плохой или хороший ответ должен соответственно означать неподходящего и подходящего кандидата. Приведу несколько эффективных вопросов из моего опыта:

- ◆ вопросы попроще на проверку базовых навыков кодирования:
  - переверните строку, состоящую из одиночных «логических» символов, — например:  $f("123hi45") = "45ih123"$ ;
  - вычислите  $n$ -е число ряда Фибоначчи — например:  $f(7) = 21$ ;
  - найдите индексы двух наименьших различных четных элементов в массиве с учетом возможного наличия одинаковых значений — например:  $[20, -40, -10]$  дает ответ  $\{1, 2\}$ , а  $[50, 40, 30, 30, 30]$  — ответ  $\{2, 3\}$  или любой другой, где есть числа 30 и 40;
  - опишите, как найти пересечение двух массивов и проанализировать производительность решения;
  - напечатайте матрицу в диагональном порядке — т. е. от сверху справа до снизу слева.
- ◆ более сложные — на проверку знаний в информатике:
  - опишите, как реализовать кеш LRU (подсказка: используйте хеш-таблицу, где элемент указывает на связанный список);
  - имея марафонскую дорожку с  $n$  бегунами и  $m$  датчиками, которые сообщают о проходящих бегунах, разработайте структуру данных для отслеживания  $k \leq n$  ведущих бегунов. Подсказка: используйте индексированную кучу из  $k$  текущих лучших;
  - опишите, как поддерживать статистику 5-го и 95-го порядка в потоке. Подсказка: используйте две очереди с приоритетом и не забывайте о пограничных случаях.

Многие другие вопросы разбросаны по всей книге и дополняют ту или иную тему.

Интервьюеры обычно работают программистами, поэтому их окончательное впечатление о кандидате зависит от того, хотят ли они видеть его в своей команде. То есть кандидаты должны подходить к задачам, поставленным на собеседовании, так, как если бы это была реальная задача во время работы. В частности, если вы застряли на задаче, приложите усилия, чтобы решить ее, и, если через некоторое время ничего не выйдет, попросите подсказку. Еще одна стратегия заключается в том, чтобы сначала попробовать простое неэффективное решение, а затем улучшить его. По крайней мере, это покажет, что вы в целом понимаете проблему. Представьте себя уверенным, мотивированным специалистом по решению задач, который сотрудничает с компанией исключительно добровольно. От кандидата обычно ожидают, что он решит все простые задачи и хорошо подумает над сложными. Задачи оцениваются с учетом их сложности, и в конечном итоге все они сходятся к некоторой оценке хорошей производительности (подробнее об оценках рассказано в *главе 21. Вычислительная статистика*). Таким образом, не закончить решение сложного вопроса или не получить идеальное решение — вполне нормально, если этого достаточно для достижения желаемой производительности.

Задача может иметь несколько хороших решений, и интервьюер обычно ожидает конкретного решения. Они бывают следующими:

- ◆ инженерное решение — максимально простое с точки зрения использования библиотечной функциональности и достаточно эффективное для заявленного сценария использования. Такие решения создаются в реальной работе;

- ◆ решение из учебника — оптимальное по быстродействию, достаточно простое, действующее один или несколько часто изучаемых алгоритмов и структур данных. Такие решения выдает кандидат, хорошо знающий информатику;
- ◆ умное, оптимизированное решение — обычно наиболее эффективный способ решения задачи, но иногда неочевидный и, возможно, реализованный только в библиотечном коде.

Большинство интервьюеров предпочитают видеть последнее решение, считают приемлемым первое в качестве начальной точки, и соглашаются со вторым. Поскольку от вас может потребоваться найти нечто большее, чем просто решение, не ограничивайте себя определенными рамками мышления. Многие кандидаты совершают эту ошибку, и тогда никакие подсказки не помогают им преодолеть это ограничение.

Один из моих любимых вопросов: найти индексы двух наименьших элементов в массиве. Вопрос взят из более сложной подзадачи оптимизации Нелдера — Мида (см. главу 24. *Численная оптимизация*). Надо отметить, что вопрос этот неполный, потому что не указано, что должно произойти, если в массиве будет менее двух элементов. Большинство кандидатов уточняют это, и я говорю, что ответ не определен, поэтому нужна проверка ввода — например, с помощью исключения.

Сначала определитесь с объявлением функции. Для поиска наименьших индексов (если кандидат задает уточняющий вопрос, я отвечаю, что обобщение задачи на большее количество не требуется) хорошим способом вернуть ответ будет функция `pair`. Некоторых кандидатов смущает необходимость возвращать более одного значения, но большинством языков это разрешено. Другие пытаются возвращать массивы или векторы, и в этом случае я настаиваю на подробностях. Часто память на массив выделяется неправильно, или один фрагмент кода его выделяет, а другой освобождает. При использовании в C++ типа `vector` требуется сравнить потребление памяти с `pair`. Поскольку `vector` использует динамическую память и хранит в стеке служебные переменные, на хранение требуется около пяти слов памяти (см. главу 5. *Фундаментальные структуры данных*). Некоторых кандидатов смущает необходимость разделять память в стеке и в куче.

Теперь решение задачи. Типичный первый ответ: отсортировать массив и получить результат. Это разрушает связь между значениями и индексами, но большинство кандидатов умеет использовать пары значений и индексов. Это хорошее инженерное решение, но не оптимальное, поэтому я прошу решить задачу без лишних затрат памяти и помощи библиотек.

Дальнейшая попытка часто заключается в выполнении поиска минимума дважды, с игнорированием первого минимума на втором проходе (я много раз просил написать код для этого, и второй цикл оказывается не так-то прост в реализации). Следующее предположение состоит в том, чтобы решить задачу за один проход и объяснить, что такой подход более эффективен с точки зрения использования кеша памяти. Те, кто изучал операционные системы, знают, что один проход приводит к меньшему количеству промахов в кеше (при условии, что массив слишком велик, чтобы полностью поместиться в кеше) и предпочтительнее, несмотря на такое же оптимальное время выполнения  $O(n)$ .

В решении с одним циклом нужно хранить две индексные переменные, соответствующие двум наименьшим значениям, обнаруженным до сих пор (в правильном порядке),

и обновлять их при циклическом поиске нового значения в массиве. Распространенной ошибкой является инициализация обоих индексов фиксированным значением — например 0. Обычно это приводит к тому, что второе значение остается равным 0, и обнаруживается при тестировании с возрастающей последовательностью значений, такой как  $[-40, -10, 20]$ . Правильной будет инициализация по значениям двух первых элементов. Остальная часть кода не представляет особых проблем, за исключением того, что нужно правильно сдвигать индексы, если встречается значение, которое меньше, чем оба известных.

На следующем шаге задачи требуется вернуть  $k > 2$  наименьших индексов, где  $1 \leq k \leq n$ . После работы над случаем  $k = 2$  очевидным решением будет использование массива  $k$  наименьших на текущий момент значений и цикла вместо операторов `if`. Но, как быстро заметил бы интервьюер, время выполнения станет равным  $O(nk)$ . Хорошим промежуточным шагом к лучшему подходу является создание массива пар «значение-индекс» и его сортировка. Время выполнения  $O(n \lg(n))$  не слишком далеко от оптимального, а вот использование лишней  $O(n)$  памяти — это плохо. Для оптимизации только времени выполнения оптимальным ожидаемым решением  $O(n + k \lg(k))$  является применение быстрого выбора (см. главу 7. *Сортировка*) для разделения массива пар значений на  $k$ -м элементе, а затем сортировка левой части. Но этот метод требует  $O(n)$  дополнительной памяти и почти никогда не применяется. При ограничении памяти  $O(k)$  оптимальным решением является использование метода грубой силы  $O(nk)$  с максимальной кучей в качестве хранилища. Хитрость заключается в том, чтобы понять, что перебор методом силы извлекает максимум из  $k$  текущих индексов и  $i$ . Многим кандидатам нужно подсказать, что требуется заменить массив динамической отсортированной последовательностью. Некоторые выбирают сбалансированное дерево, которое также дает время выполнения  $O(n \lg(k))$ , но постоянный фактор в этом случае хуже.

Я прошу применить в реализации минимальную кучу из стандартной библиотеки языка, используемого кандидатом. В некоторых случаях кандидаты не знают, как написать компаратор, который ожидает минимальную кучу. В C++ нужно сделать только индексы и написать класс, содержащий:

```
operator<(int a, int b) const { return array[b] < array[a]; },
```

где `array` — ссылка переменной-члена на входной вектор (подробнее о написании компараторов см. в главе 5. *Фундаментальные структуры данных*). Это также можно сделать с помощью лямбда-функций). В других языках может потребоваться явное создание кучи пар «индекс-значение».

После цикла нужно получить  $k$  наименьших индексов, многократно извлекая их из очереди в результирующий массив. Большинство кандидатов не понимают, что в этом случае индексы возвращаются в обратном порядке. После подсказки это быстро исправляется перемещением индексов в конец массива.

Все, что я описал здесь, могло создать ложное впечатление, что процесс собеседования в основном механический, а хорошая подготовка гарантирует удовлетворительное выполнение. Но это не совсем так:

- ◆ вы можете плохо себя чувствовать, не выспавшись из-за прилета накануне вечером, или просто волноваться. Могут быть и другие внезапные проблемы — такие как головная боль;

- ◆ некоторые интервьюеры более снисходительны, чем другие, и задают вопросы с меньшим количеством логических щелчков (я сам такой);
- ◆ вы можете быть хорошим программистом, но не решить проблему за отведенное время, или вам трудно рассуждать вслух, как это обычно требуется.

Так что процесс собеседования несовершенен, но лучше ничего не придумано. По моему опыту решения задач, даже на простую задачу обычно уходит целый день размышлений, поскольку требуется проработать все детали, и первое реализованное решение, даже если оно хорошо аргументировано, скорее всего, потребует ряда улучшений. Оценка в основном зависит от того, считает ли интервьюер, что вы можете стать частью его команды, поэтому сделайте глубокий вдох и постарайтесь решить все как можно лучше. Если вы покажете, что можете с ними работать, они будут работать с вами.

## 3.6. Более сложные вопросы

Обычно собеседования проводятся в несколько этапов — например, первым идет техническое отборочное собеседование, за которым следуют несколько основных собеседований. Мне всегда задавали по крайней мере один простой вопрос, подобный тому, что мы рассмотрели в предыдущем разделе. Компании, которые тщательно отбирают кандидатов, могут еще больше усложнить собеседование:

- ◆ ввести, например, ограничение по времени. Так, однажды у меня было автоматизированное отборочное собеседование, на котором требовалось ответить на четыре вопроса средней сложности за 70 минут. В другом месте мне задали простые вопросы по кодированию, которые нужно было решить за 10 минут.
- ◆ ставить трудные вопросы, подготовка ответа на которые может занять очень много времени. Например, мне задавали вопросы, для оптимального решения которых требовалось знание топологической сортировки и умение модифицировать ее таким образом, что простая версия на основе DFS, которую я рассматриваю в *главе 11. Алгоритмы графов*, уже не годилась (а нужна была более сложная система сортировки с учетом версий).

Подготовка к собеседованию с помощью книги [3,3] или других ресурсов помогает настроиться на общие методы решения. Если в задаче фигурируют:

- ◆ массивы — рассмотрите возможность начать работу с конца, чтобы было проще управлять памятью;
- ◆ связанные списки — используйте метод 2-го указателя (называемого *бегуном*), когда несколько указателей проходят цикл с разными шагами и в конечном итоге встречаются. Я припоминаю только один вопрос, где это было необходимо, потому что интервьюеры, как правило, сами не имеют большого опыта работы со списками;
- ◆ деревья — нужно знать, как реализовать пред-, пост- и порядковую итерацию. Для вопросов, связанных с уровнями (например, вычисления суммы каждого уровня) используется метод поиска в глубину с массивом или хеш-таблицей, в которых хранятся данные для каждого уровня;
- ◆ графы — чаще всего задачи решаются с помощью DFS. Не используйте более сложные примитивы вроде union-find, т. к. они слишком сложны, чтобы правильно реализовать их на собеседовании, — я пытался и знаю, что это такое;

- ◆ динамическое программирование — в некоторых компаниях запрещено его использование, но можно применить рекурсию с запоминанием.

## 3.7. Собеседование по проектированию систем

Обычно это касается распределенных систем (например, разработка распределенных хеш-таблиц), а иногда и продукта (например, разработка системы предложения книг). В книге [3.3] есть краткая глава об проектировании, но этого мало. О распределенных системах можно почитать в [3.2], а о продуктах — в [3.1], но и этого тоже недостаточно.

В рассматриваемом случае преуспеть поможет только опыт: инженеры инфраструктуры получают преимущество в первом случае, а разработчики приложений — во втором. Учитывайте это при планировании своей карьеры.

## 3.8. Обсуждение предложения

Если вы успешно прошли собеседования в нескольких компаниях, то получите несколько предложений. Самая распространенная ошибка — принять лучшее предложение сразу, без переговоров. Люди часто не хотят обсуждать денежные вопросы и думают, что им предлагают не больше, чем они заслуживают. Рекрутеры делают вам предложения в зависимости от ваших результатов на собеседовании и каких-либо пунктов вашего резюме — например, многолетнего опыта, высшего образования или знания необходимых технологий. Рассмотрим несколько советов по переговорам:

- ◆ не бойтесь — читайте онлайн-статьи, смотрите обучающие видео и не беспокойтесь о том, что предложение будет отменено после попытки договориться о зарплате или слишком высоких запросов с вашей стороны. Но будьте вежливы, потому что грубость аннулирует предложение. Компания довольна вашей работой на собеседовании и ожидает переговоров. Найти отличного кандидата сложно, и они нанесут вред своей репутации, если отменят предложение просто по какой-то прихоти;
- ◆ если у вас много предложений, расскажите рекрутеру о них. Скорее всего, он будет готов пересмотреть условия. Но лучше не говорить, что конкретно содержится в других предложениях, если вас об этом спросят. Говорите, что примете предложение, если в нем будет еще вот это и это;
- ◆ посмотрите статистику зарплат на **level.fyi** и покажите ее рекрутеру, если его предложение хуже среднего. В целом предложения другим сотрудникам не имеют никакого отношения к вашим, но зато это простой способ попросить больше;
- ◆ запросите конкретную и реалистичную прибавку. Для начала можно задать определенный жесткий минимум (на основе вашей текущей зарплаты или финансовых потребностей) или на 10–20% больше первоначального предложения, потому что рекрутеры будут готовы пойти вам навстречу хотя бы частично. А даже если нет, вы можете выиграть в чем-то другом — например, удлинить отпуск или увеличить компенсацию за переезд;
- ◆ предложение — это пакет, а не просто сумма заработной платы, премий, бонусов и прочего. Компания может менять цифры в этом списке, но уровень зарплат в ней

часто может быть ограничен штатным расписанием. Поэтому сначала попросите самую большую зарплату, затем больше акций и, наконец, соглашайтесь на бонусы. Это также позволит вам оценить прогресс в переговорах с компанией и даст вам время, если вы ожидаете поступления других предложений.

Помните, что любая экономическая сделка — это вопрос спроса и предложения, поэтому ваша позиция на переговорах зависит от количества хороших предложений, которые у вас есть, включая вашу текущую позицию. Рекрутеры обычно любезны и передают ваши запросы менеджеру, отвечающему за деньги, но иногда они тоже умеют вести переговоры и обязаны использовать любую известную им тактику, чтобы получить ваше согласие по самой низкой цене.

### 3.9. Как красиво уйти с текущей работы?

Если вы принимаете предложение, предупредите свою компанию об этом за две недели (или в иной установленный срок) и на прощание поблагодарите ее за ценный опыт. Они могут захотеть через некоторое время пригласить вас обратно, или вам может понадобиться рекомендация. Однажды я видел, как кто-то прислал прощальное электронное письмо, полное негатива. Письмо было забавным, но оно не содержало конструктивной обратной связи, и, вероятно, никто не захочет работать с этим человеком в будущем.

Не расстраивайтесь из-за того, что покидаете свою нынешнюю команду. Все понимают, что двигаться дальше — это личное дело каждого, и часто бывшие коллеги остаются на связи и время от времени встречаются. В последние несколько дней сделайте все возможное, чтобы обучить себе замену, но не слишком продлевайте свое пребывание в компании ради этого, т. к. вас могут отпустить раньше, в том числе и в день уведомления. Поэтому выберите период времени, который устроит и вас, и компанию.

### 3.10. Боритесь с самуспокоенностью

Желательно проходить собеседования примерно раз в 1–2 года, даже если вам нравится ваша нынешняя должность (а особенно, если не нравится!). Вам не обязательно принимать предложения, которые вы получаете, но сам процесс позволяет поддерживать в хорошей форме навыки задач решения, подсказывает, в каком направлении развиваться, и устраняет самуспокоенность, связанную с комфортом пребывания на текущей должности. А еще можно начать самому проводить собеседования — этот опыт бесценен.

Неофициальные данные гласят, что многих разработчиков вполне устраивают ограниченные возможности обучения и роста. Это плохо сказывается на карьерном росте, поэтому в такой ситуации следует как можно скорее начать проходить собеседования с другими командами (если работаете в крупной компании) или компаниями, чтобы не усугублять проблему. Даже когда вы откликаетесь на вакансию в другой компании, человек с 10-летним опытом работы в разных командах или компаниях выглядит лучше, чем человек, работающий все эти годы на одном и том же месте (если, конечно, кандидат не меняет место работы слишком часто — нормальным считается 2 года в одной компании и 1 год в одной команде). К тому же наличие разнообразного опыта помогает получать повышение.



Если вы чувствуете, что еще растете и учитесь, продолжайте искать больше возможностей для роста. Всегда отдавайте предпочтение изучению фундаментальных идей, а не конкретных технологий, т. к. при наличии понимания легко освоить инструменты, а наоборот не получится. Я не раз видел, что происходит, когда человеку, плохо знакомому с машинным обучением, попадает в руки популярный набор инструментов языка Python.

## 3.11. Советы по дополнительной подготовке

- ◆ Для отработки вопросов на собеседовании вы можете воспользоваться популярным сайтом **leetcode.com**. Многие программисты заходят туда и решают по задачке в день. Я считаю, что это пустая трата времени, если вы сами проводите собеседования, но в противном случае это необходимо. Согласны?
- ◆ Решите несколько задач из интервью на других языках программирования — таких как Python или JavaScript. Помогает ли это лучше понять сам вопрос и его возможные решения и их систему оценки?

## 3.12. Список рекомендуемой литературы

- 3.1. Cervantes H., & Kazman R. (2016). Designing Software Architectures: A Practical Approach. Addison-Wesley Professional.
- 3.2. Kleppmann M. (2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly.
- 3.3. McDowell G. L. (2019). Cracking the Coding Interview: 189 Programming Questions and Solutions. CareerCup.
- 3.4. Sonmez J. (2017). The Complete Developer's Career Guide. Simple Programmer.

## 4. Основы компьютерного права

### 4.1. Введение

Вопросы права всегда обширны и сложны, а законы в разных странах различаются, несмотря на серьезную работу по их стандартизации посредством различных договоров. В этой главе приведена некоторая общая полезная информация о постоянно меняющемся законодательстве США.

#### **СОВЕТ**

Чтобы получить точную информацию по тому или иному вопросу, лучше всего проконсультироваться с юристом.

Главная цель разработчиков — не попасть под суд из-за невинной ошибки. Чтобы этого не случилось, лучше заранее знать, где поджидает опасность. Обычно, даже если у противной стороны есть повод, требующий судебного разбирательства, иск возникает либо если по его результатам сторона планирует отсудить сумму, достаточную хотя бы для оплаты судебных издержек, либо если имеет место эмоциональный фактор. Но это слабое утешение.

### 4.2. Интеллектуальная собственность

Создание знаний обходится дорого, и наиболее важные из них обычно хранятся в секрете. Чтобы стимулировать обмен знаниями, закон признает некоторые полученные знания частной собственностью. С юридической точки зрения *собственность* — это набор прав ее владельцев, и ее несанкционированное использование любым способом, нарушающим права собственности, является нарушением закона. Разрешение на использование интеллектуальной собственности регулируется *лицензией* (табл. 4.1).

*Таблица 4.1. Типы интеллектуальной собственности*

Собственность	Чем регулируется	Пример программного обеспечения
Идея	Патент или коммерческая тайна	Алгоритм
Выражение	Авторские права	Исходный код
Ассоциация	Товарный знак	Название программы

### 4.3. Патенты

*Патент:*

- ♦ описывает *новые и полезные* процессы, машины, производства и композиции, являющие собой последовательность шагов по преобразованию некоторого физического объекта, устройства, рукотворного предмета, устройства или вещества;
- ♦ дает исключительное право производить, использовать или продавать само изобретение или что-либо, содержащее его, в течение ограниченного периода времени (обычно 20 лет);
- ♦ по истечении срока действия становится достоянием общественности;
- ♦ может быть случайно нарушен. Если вы вызываете чей-либо API, защищенный авторским правом, вы нарушаете закон, сами того не зная.

Почти все можно считать *полезным*, если средний человек может без особых усилий воссоздать это. *Новым* считается изобретение (открытие), которое ранее не было опубликовано и не было очевидным до момента публикации. Как правило, публикация изобретения означает его раскрытие, если раскрывающая сторона не подписывала соглашение о неразглашении. Например, когда ваш ребенок не может уснуть, вы качаете коляску и поете колыбельную. Чтобы облегчить себе этот процесс, вы изобретаете роботизированную руку — и ее можно запатентовать (насколько я знаю, подобного еще никто не сделал), по крайней мере, до тех пор, пока ваше изобретение не будет опубликовано.

Нельзя запатентовать вещи, которые:

- ♦ возникают естественным путем — т. е. открытие не является новой сущностью;
- ♦ абстрактны — если идея не является осязаемой сущностью. Решение вопросов по части идей, которые сейчас могут считаться чисто абстрактными, является предметом многих недавних дел Верховного суда. Алгоритмы в настоящее время патентоспособны. Хотя в ряде случаев решения суда отвергали патентоспособность алгоритмов, все больше устройств полагаются на нетривиальные программные компоненты. Будущие законы и дела покажут, как все будет происходить далее;
- ♦ не несут функциональности — например, картины или другие произведения искусства.

Подать заявку на патент в принципе просто, но сложно в реализации, и процесс этот требует больших затрат (по состоянию на 2010 год — порядка 10 тыс. долларов США на подачу заявки и 20 тыс. долларов США в течение срока действия патента). Эксперт изучает заявку и принимает ее или возвращает на доработку, если некоторые заявленные новшества таковыми не признаются. Поначалу большинство патентов охватывают слишком широкий спектр обстоятельств и несколько раз перерабатываются, пока не будут приняты менее широкие формулировки изобретения.

Выданный патент содержит:

- ♦ список изобретений и перечень изобретателей каждого из них. Каждый изобретатель независимо владеет своими изобретениями, если не передал свои права кому-то другому;
- ♦ даты: *дату выдачи* патента, начиная с которой он вступает в силу, *дату подачи заявки* и *дату приоритета*, с которой изобретение считается сделанным. С даты подачи

заявления отсчитывается двадцать лет. В прошлом дата начала определялась сложными правилами, согласно которым она всегда находилась между датой приоритета и датой подачи заявки;

- ◆ текстовые описания и иллюстрации с рисунками изобретений. Их должно быть достаточно, чтобы изобретение можно было воспроизвести, иначе патент окажется недействительным. Скрывать важные детали запрещено;
- ◆ *подробное описание инновационных аспектов изобретения* — чтобы предупредить возможность внесения в предмет изобретения незначительных изменений, приводящих к логическим нарушениям. Создание кем-либо другим устройства с «по существу эквивалентной» функциональностью является нарушением, но это бывает трудно доказать.

Ущерб за нарушение патентных прав может заключаться:

- ◆ в упущенной выгоде;
- ◆ в потере разумных авторских отчислений;
- ◆ в запрете на дальнейшее использование изобретения, если это не противоречит общественным интересам.

#### **СОВЕТ**

Если алгоритм запатентован, даже если он очевиден, на всякий случай используйте другой алгоритм.

## **4.4. Коммерческие тайны**

*Коммерческая тайна:*

- ◆ распространяется на информацию, которая не является общеизвестной, не является незаконной, часть ценности которой состоит в ее секретности, защищенной системами безопасности, и не получена мошенническим путем. Компании усиливают это определение в трудовых договорах, считая всю внутреннюю информацию секретом, если нет необходимости делиться этим с посторонними;
- ◆ защищает информацию от неправомерного разглашения или разглашения путем нарушения конфиденциальных отношений. *Неправомерным* действием является, например, взятка работнику. Вы не можете раскрыть случайно полученную информацию, если у вас есть веские основания подозревать, что она секретная, и если раскрытие делает эту информацию общественным достоянием. Использование коммерческой тайны требует явного или подразумеваемого разрешения;
- ◆ *конфиденциальные отношения* возникают между сторонами, ведущими друг с другом бизнес, и нарушаются, если одна раскрывает секреты другой. Многие компании не рассматривают предложения, в которых не указано, что их содержимое не является секретным;
- ◆ не защищает от независимого изобретения и обратного инжиниринга;
- ◆ перестает действовать, если информация становится общеизвестной — т. е. известной достаточно большому количеству людей. Например, если сотрудник публикует секрет в Интернете, он перестает быть секретом, но компания может подать на этого сотрудника в суд;

- ◆ не обязательно имеет уникального владельца. Несколько компаний могут владеть секретом одной и той же идеи, но если какая-то одна из них раскрывает секрет, теряют его все. В случае нарушения коммерческой тайны право на возмещение ущерба имеет только тот ее владелец, у которого секрет был украден.

## 4.5. Авторские права

*Авторское право* запрещает точное воспроизведение оригинальной работы, зафиксированной на любом материальном носителе, а также изготовление копий, их распространение и создание производных работ.

- ◆ Любой способ записи обладает силой. Стихотворения, написанные на салфетке или на компьютере, равноценны.
- ◆ Идеи и факты не защищаются авторским правом. Выражение не защищено, если оно излагает идею. Коллекция фактов не защищена авторским правом, если их выбор не является творческим. Защищена может быть только конкретная творческая компоновка или презентация. Вполне вероятно, что в будущем наборы данных тоже будут попадать под защиту.
- ◆ Оригинальности должно быть немного. Комбинация нескольких элементов рассматривается как комбинация, а не как целое, если она является функциональной, а не творческой. Например, разделение GUI на компоненты и индивидуальная защита компонентов авторским правом определяют защиту всего интерфейса.
- ◆ Авторское право принадлежит нескольким сторонам, если каждая из них создала выражение без копирования. Например, в процессе обратного инжиниринга команда разработчиков получает полные функциональные спецификации продукта, не видя их.
- ◆ Работает автоматически и не требует регистрации. Приписка «Этот материал защищен авторским правом» в США не обязательна, но на международном уровне ее достаточно, чтобы владеть авторским правом.
- ◆ Обычно действует при жизни автора и еще 70 лет после его смерти. Для произведений, написанных до 1978 года или созданных в рамках *работы по найму* срок действия авторского права обычно составляет 95 лет после публикации. Все, что выпущено до 1923 года, является общественным достоянием, но на более поздние работы авторские права могут быть продлены.
- ◆ Владелец права является художник, а не устройство, используемое для творчества, и не его владелец. Если турист просит вас сфотографировать его своим фотоаппаратом, вы владеете авторскими правами на фото. Но на использование объекта права, созданного устройством, требуется разрешение. Например, известное случайное селфи обезьяны признается общественным достоянием, потому что производитель камеры не имел намерений создавать этот объект.
- ◆ Если вы создаете защищенный авторским правом контент в рамках наемной работы, право владения обычно остается за вашим нанимателем. Как правило, работник подписывает соглашение о передаче всех таких прав.
- ◆ Совместная работа нескольких авторов полностью принадлежит каждому автору, если планировался индивидуальный вклад каждого. Производная работа не является совместной, даже если это разрешено.

- ◆ Права владельцев достаточно обширны. Например, загрузка нелегальной копии произведения нарушает право на распространение, т. к. загрузка — это тоже копия.
- ◆ *Доктрина добросовестного использования* в некоторых случаях позволяет полностью или частично копировать произведение, особенно в небольших количествах, с целью обучения или критики. Например, удаление нескольких пикселей из изображения или копирование легально приобретенного компакт-диска для личного использования не является нарушением авторских прав. Но определение добросовестного использования довольно размыто и часто непредсказуемо — например, сервис Google Books был признан Верховным судом добросовестным использованием, в числе прочего, за счет существенной пользы для общества. Но лучше получить явное разрешение.
- ◆ *Доктрина первой продажи* позволяет покупателю работы, защищенной авторским правом, перепродавать ее, за некоторыми исключениями. Почти все программное обеспечение запрещает перепродажу лицензии, которая обычно имеет обязательную силу.
- ◆ Существенное содействие нарушению также влечет за собой ответственность, равно как и знание о нарушении, получение от него финансовой выгоды и управление процессом нарушения. Вот почему определенные типы файлообменных сервисов в конечном итоге выходят из бизнеса. Незаконно также содействовать нарушению авторских прав, создавая или продавая инструменты для нарушения. Поставщики контента, такие как YouTube, не несут ответственности за нарушения прав пользователей, если после получения соответствующего уведомления быстро удаляют материалы, нарушающие авторские права.

## 4.6. Товарные знаки

*Товарный знак* — это любой символ, используемый в коммерческих целях для идентификации товаров. *Сервисный знак* — это то же самое, но для услуг. Существуют также *знаки сертификации* и *коллективные знаки*. Чтобы подать в суд за нарушение права использования товарного знака, необходимо его зарегистрировать, и это позволит избежать споров, если несколько сторон предъявляют требования. Надпись «<Товарный знак>®» означает зарегистрированный товарный знак, а «<Товарный знак>™» или «<Сервисный знак><sup>SM</sup>» — незарегистрированный.

Только владелец может использовать товарный или сервисный знак для обозначения товаров или услуг, т. к. потребители ассоциируют этот знак с ним и его репутацией. Намеренное создание путаницы или нарушение авторских прав запрещено (например, поисковая система Y!haa или нижнее белье micro-soft). Исключение составляют художественные произведения — особенно пародийные, если очевидно, что владелец товарного знака их не производил.

Символ товарного знака не может быть:

- ◆ функциональным. Полезная информация — например, указатель направления, не может быть товарным знаком;
- ◆ оскорбительным, чрезмерно обобщенным или вводящим в заблуждение. Самые лучшие товарные знаки — это случайные символы. Например, «ПарСАН» — хоро-

ший знак, а «Парикмахерский салон Анастасии» — слишком для него общее название. Но даже и в этом случае, чтобы создать уникальный товарный знак, можно внести отличительное изменение в правописание — например, так: «Парикмахерский салон Анаста\$ии».

- ♦ вариацией уже существующего товарного знака — во избежание путаницы у потребителей. Но у разных отраслей и регионов существуют исключения.

Товарный знак имеет силу пожизненно, но может быть отозван, если становится слишком распространенным или не используется в коммерческих целях в течение трех лет, а попыток возобновить использование не предпринимается. Например, слова «застежка-молния» и «эскалатор» стали слишком распространенными, и с google скоро произойдет то же самое. Под использованием товарного знака понимается его демонстрация, чтобы он ассоциировался с товарами или услугами, а также активная защита с привлечением нарушителей к ответственности. Регистрация доменного имени под это определение не подходит, поскольку сама по себе такая регистрация не является коммерческим использованием.

Если право собственности на товарный знак оспаривают несколько сторон, побеждает та, которая раньше всех использовала товарный знак, если только одна из сторон не зарегистрировала этот товарный знак пять лет назад и не использовала его в течение всего времени, т. е. в этом случае ее право *неоспоримо*.

## 4.7. Управление интеллектуальной собственностью

Патенты и коммерческая тайна исключают друг друга. Но некоторые коммерческие тайны могут стать патентами. И то и другое может сочетаться с товарными знаками, потому что после истечения срока действия патента или раскрытия тайны потребители, привыкшие к продукту, обычно продолжают его использовать.

Для объединения нескольких объектов интеллектуальной собственности с целью развития торговли может быть создан *стандарт*. В этом случае гарантируется отсутствие претензий на интеллектуальную собственность, но органу по стандартизации нужно платить лицензионные отчисления.

Важно вести надлежащий учет контрактов — особенно разрешений других лиц на использование их интеллектуальной собственности. Это поможет застраховаться от недоразумений, которые время от времени случаются.

## 4.8. Контракты

*Контракт* — это соглашение между несколькими сторонами о том, что каждая из них должна сделать что-то конкретное. Чтобы контракт имел законную силу:

- ♦ должно быть сделано *предложение* и — пока оно не будет отклонено или не истечет срок его действия — оно должно быть передано с помощью надлежащих средств связи, таких как почта, электронная почта или иной указанный способ. Переговорами считается встречное предложение и неявный отказ. Если вы задаете вопросы о предложении — это не переговоры;

- ◆ он не должен подразумевать незаконную деятельность. Если контракт был заключен в мошеннических целях или сторона была принуждена к его подписанию, то он считается недействительным;
- ◆ все стороны должны получить *взамен* что-то, чем они еще не обладают. Например, если вы платите кому-то часть денег по контракту, обещание не требовать остатка суммы недействительно;
- ◆ стороны должны быть *дееспособны*. Контракт, заключенный с несовершеннолетним или недееспособным лицом, как правило, недействителен. Для заключения контракта с компанией физическое лицо, заключающее контракт от ее имени, должно иметь достаточные для этого полномочия;
- ◆ стороны контракта должны полностью понимать все существенные условия. Например, условие «настоящим вы отдаете мне свою квартиру, машину и собаку», напечатанное мелким шрифтом, не имеет обязательной силы, если вы прямо не признаете это (и даже в этом случае квартиру у вас не отберут). Общие договоры, такие как лизинг, как правило, стандартизированы, и предполагается, что все стороны знают общие условия, а особые условия должны быть прописаны отдельно. В подписании документов без прочтения мелкого шрифта нет ничего страшного — даже юристы делают это постоянно;
- ◆ в случае, например, покупки недвижимости он должен быть оформлен письменно. Чтобы проще было доказать факты, большинство контрактов, даже с незначительной стоимостью, заключаются в письменной форме, но большинство устных контрактов тоже действительны. Электронный контракт, принятый нажатием на кнопку, действителен так же, как и письменный;

Исполнение контракта некоторой стороной является *полным*, если все в нем прописанное выполнено с несущественными изменениями, — т. е. другие стороны не считают их существенными, и договор не запрещает такие изменения. Если сторона не выполняет свои обязательства, другие стороны могут подать на нее в суд, но не могут принуждать виновника к их исполнению. Только суд может присудить возмещение убытков или принудить виновника к исполнению, если убыток нельзя возместить (например, если речь о продаже уникальных вещей), и если это не наносит слишком большой ущерб исполняющей стороне.

В некоторых случаях контракт заключается автоматически, чтобы не возникало несправедливости или неосновательного обогащения. Например, вы не можете отказаться заплатить разумную цену за свой обед в ресторане, даже если не давали на это явного согласия перед заказом.

## 4.9. Лицензии

С точки зрения программного обеспечения *лицензия* — это договор между вами и владельцем. Без лицензии вы, скорее всего, вообще не сможете пользоваться некоторой собственностью.

Владелец лицензии может получить многое. Это может быть гордость автора тем, что кто-то использует его программное обеспечение. Это может быть лицензия на шесть упаковок пива.



Лицензия на открытый исходный код требует, чтобы код был доступен по запросу, если вы используете что-либо, подпадающее под ее действие, в качестве компонента. Бывают лицензии, для использования которых необходимо некоторое уведомление. Большинство лицензий требует оплаты, и в них говорится, что вы не можете перепродавать программное обеспечение или передавать его другим лицам. Существуют комбинированные лицензии, разрешающие использование в личных и образовательных целях по одним правилам, а коммерческое использование — по другим.

Лицензирование — это не продажа, и принцип первой продажи в этом случае не применяется. Если на товаре есть примечание «лицензия внутри», подразумевается принятие некоторых условий при покупке, а не при нажатии кнопки «Я согласен» во время установки. Обратный инжиниринг законен, даже если лицензия запрещает это, но только если вы не обходите защиту авторских прав. Лицензия также содержит:

- ◆ условия оплаты;
- ◆ гарантии качества и описание потенциального судебного процесса. Обычно компания требует проводить судебный процесс в благоприятной для нее юрисдикции и отказывается от любой ответственности в других юрисдикциях;
- ◆ инструкции по обновлению и контакты техподдержки. Обновления новой версии могут быть обязательными;
- ◆ условия использования. *Исключительная лицензия* означает, что владелец не может лицензировать собственность кому-либо еще. Если плата за лицензию небольшая, это может сделать интеллектуальную собственность практически бесполезной. Лицензии с открытым исходным кодом могут потребовать публикации любых производных продуктов.

Чтобы лицензировать свою собственность, лучше использовать существующие популярные лицензии, а не свои собственные, чтобы не напортачить везде, где только можно.

У проектов с открытым исходным кодом есть свои проблемы:

- ◆ код обычно исходит от нескольких участников, и если проект с самого начала не принуждает согласиться на общую лицензию, каждый вклад потенциально может лицензироваться по-разному. Каждая из этих лицензий является обязательной;
- ◆ атрибуция обычно требуется даже для самых разрешительных лицензий — даже для бинарных файлов. Таким образом, оптимизация, которая удаляет комментарии, может нарушать авторские права, потому что некоторые комментарии могут быть атрибуцией;
- ◆ случайное объединение проприетарного стороннего кода с кодом, защищенным авторским копирифтом, может привести к существенной ответственности, поскольку этот код не может использоваться совместно с требованиями авторского копирифта.

## 4.10. Трудовые соглашения

Устраиваясь на работу программистом, вы обычно подписываете контракт, в котором прописаны условия найма — например, ваши обязанности и зарплата, а также *соглашение о неразглашении* (nondisclosure agreement, NDA). Контракт гарантирует, что вы не

станете разглашать служебную информацию даже после увольнения и не будете считать что-либо, произведенное вами для компании, своей собственностью. Обычно любая интеллектуальная собственность производится в рамках работы по найму, но бывают и исключения. В частности, вам может быть запрещено выполнять какую-либо работу не для компании, но связанную с ее бизнесом, независимо от того, производите ли вы ее в рабочее время или используете ресурсы компании.

Если вы в нерабочее время занимаетесь проектом, охватываемым соглашением, перед началом работы вам нужно получить письменное разрешение юридического отдела компании. Также вы должны периодически показывать им результаты своей работы и получать письменное подтверждение отсутствия с их стороны претензий. У многих компаний на этот случай разработаны специальные протоколы.

Иногда в NDA также предусматривается *соглашение о неконкуренции*, которое запрещает бывшим сотрудникам какое-то время работать на компанию-конкурента или в конкретной отрасли. В большинстве случаев это реализуемо лишь в некоторых пределах, а часто и вовсе не возможно — зависит от региона и конкретных указанных в соглашении ограничений. Например, на неконкурентный период могут сохраняться выплаты заработной платы, но этот период не может длиться годами.

Иногда принимают *соглашение о неприглашении*, которое запрещает ушедшему сотруднику предлагать клиентам компании и другим ее сотрудникам перейти в новую компанию. То, что касается клиента, как правило, подлежит принудительному исполнению, а сотрудников заставить остаться труднее. В последнем случае законы не вполне ясны, поэтому безопаснее соблюдать условия.

## 4.11. Конфиденциальность

Люди имеют право на неприкосновенность частной жизни и, в частности, на электронную конфиденциальность. Законы о *клевете/диффамации* запрещают раскрытие информации, даже правдивой, если она:

- ◆ слишком оскорбительна (в разумных пределах);
- ◆ не представляет интереса с точки зрения закона.

В противном случае защититься от клеветы поможет правда. В реальной жизни клевета часто проявляется в отзывах на продукт. Иногда мнения вроде «никогда не имейте с этой компанией дела» можно толковать как клевету. Лучше не обобщать и писать, например: «Я больше никогда не буду иметь с ними дело». Еще одним распространенным моментом, которого следует остерегаться, является предоставление кому-либо рекомендации: многие компании специально проверяют только даты приема на работу и запрещают сотрудникам давать личные рекомендации из-за возможности потенциального распространения клеветы.

Если украденная информация не была должным образом защищена, за это предусмотрена юридическая ответственность. Компании, подвергшиеся взлому, могут заплатить серьезный штраф, если причиной взлома стала ненадлежащая защита.

Специальные органы издают немало законов о контроле данных (например, Европейский союз и штат Калифорния), которые дают жителям право удалять свои данные — такие как учетная запись или история поиска, и любое юридическое лицо в юрисдикции, где такие законы приняты, должно их соблюдать.

Конфиденциальность существует там, где это разумно. Если вы поймаете коллегу на просмотре материалов для взрослых, и его за это уволят, он не может жаловаться. Но обыскивать его сумку — незаконно.

Программа не может следить за вашими действиями, если вы не дадите на это разрешение. Программное обеспечение должно учитывать предпочтения пользователей в вопросах отслеживания и сбора данных. Сбор данных о детях в возрасте 13 лет и младше без специального разрешения запрещен.

## 4.12. Киберпреступления

В рассылках по электронной почте должна быть предусмотрена возможность отказа от подписки, а регистрационная форма должна позволять отказаться от получения рекламы.

Несанкционированное использование системы и кража личных данных преследуются по закону. Сюда же входит использование чужого пароля для входа. При оценке несанкционированного использования учитываются намерения и нанесенный ущерб.

## 4.13. Выступления в качестве свидетеля-эксперта

Сторонам разрешено привлекать в суд экспертов для дачи показаний по соответствующим вопросам. Вопросы в судах обсуждаются серьезные, поэтому возникает много проблем.

Свидетели-эксперты несут ответственность перед судом, и нанимающая сторона оплачивает их время (а в некоторых странах суд нанимает их напрямую). Поскольку разногласия для людей — обычное дело, противоборствующие стороны без труда находят свидетелей, искренне верящих в нужную им точку зрения. Например, однажды я услышал от врача, что интерпретация изображений МРТ не подкреплена доказательствами. И это притом, что, адвокаты не позволили никому с медицинским образованием войти в состав присяжных. Бывают и смешанные стратегии: обе стороны нанимают медицинского эксперта для дачи показаний о необходимом лечении и специалиста по финансовому планированию для расчета и суммирования затрат, получая в итоге разные цифры.

Судья до суда определяет, кому разрешено давать показания в качестве свидетеля-эксперта, и что им разрешено говорить. Дело в том, что эксперт должен иметь достаточную квалификацию и использовать для заключения выводов *разумно принятую методологию*. Методология может быть описана в рецензируемой публикации, а может просто быть полезной для суда. Первое требование может быть слишком обременительным для стороны процесса, но было много случаев, когда анализ заключения эксперта спустя годы признавался научным сообществом недействительным. Обе стороны пытаются воспрепятствовать выдаче экспертных показаний друг друга.

Потенциальный свидетель-эксперт должен быть готов к тщательному перекрестному допросу другой стороной. Главная цель такого допроса — заставить эксперта признать определенные удобные для той стороны моменты и, возможно, сбить с его толку. Это

похоже на устный экзамен, за исключением того, что другая сторона задает трудные вопросы только для того, чтобы спровоцировать ошибки или посеять сомнения.

## 4.14. Советы по дополнительной подготовке

Исследуйте проекты по интеллектуальной собственности шрифтов. Отличайте собственно *шрифт* и его вид на печати. Вид шрифта на печати авторским правом в США не защищается. Но для электронной публикации используются шрифты, а не их отображения на печати. Будет ли использование растрового изображения шрифта обходным путем? Почитайте в этих источниках:

- ◆ <http://www.oddmoxie.com/blog/2016/2/8/everything-you-want-to-know-about-using-fonts-legally-but-didnt-know-to-ask>;
- ◆ [https://en.wikipedia.org/wiki/Intellectual\\_property\\_protection\\_of\\_typefaces](https://en.wikipedia.org/wiki/Intellectual_property_protection_of_typefaces).

## 4.15. Список рекомендуемой литературы

- 4.1. Kadane J. B. (2008). Statistics in the Law: A Practitioner's Guide, Cases, and Materials. Oxford University Press.
- 4.2. Landy G. K. (2008). The IT/Digital Legal Companion: A Comprehensive Business Guide to Software, Internet, and IP Law. Syngress.
- 4.3. Mallor J. P., Barnes A. J., Bowers T., & Langvardt A. W. (2006). Business Law: The Ethical, Global, and E-commerce Environment. McGraw.
- 4.4. McJohn S. M., & Graham L. (2016). Fundamentals of Intellectual Property Law. American Bar Association.
- 4.5. Stim R. (202\*). Patent, Copyright & Trademark: an Intellectual Property Desk Reference. Nolo. (A new edition comes out every year).

## 5. Фундаментальные структуры данных

После тяжелых лекций по квантовой механике страшно хочется выпить чего-нибудь крепкого.  
*Джеймс Джинс*

### 5.1. Введение

Массивы интуитивно понятны и универсальны (даже машина Тьюринга представляет собой большой массив), но для программиста, который только массивы и знает, существование других структур данных становится откровением. Большая часть материала этой главы проходится на занятиях по структурам данных. Поэтому я ожидаю, что читатель знаком с динамическими массивами, связанными списками, стеками, очередями и т. п. Акцент здесь делается на их хорошей реализации на C++.

### 5.2. Вспомогательные функции

В C++, к сожалению, нет многих базовых и часто используемых функций, поэтому я разработал их сам. Например, вам не помешал бы макрос, который печатает имя и значение переменной. В моей реализации в файле `Debug.h` показан хорошо спроектированный файл (хотя здесь `DEBUG` — это макрос, на который не влияют пространства имен):

```
#ifndef IGMDK_DEBUG_H
#define IGMDK_DEBUG_H
#include <iostream>
using namespace std;
namespace igmdk{
// вывод выражения, пробела и символа новой строки
#define DEBUG(var) cout << #var " "<< (var) << endl;

} // конец пространства имен
#endif
```

В частности, для эффективного *управления большим объемом кода*:

- ♦ код разбивается на пакеты функций — по одной на файл. Пакет может содержать несколько классов и функций. Обычно структура данных представляет собой пакет с одним классом. Но алгоритмы и их вспомогательные функции, выполняющие одну задачу, также объединяются в пакет, хотя на эту тему часто спорят. В случае сомнений я предпочитаю иметь меньше пакетов, но большего размера;
- ♦ в заголовках используются *операторы* `include`, которые добавляют все необходимое, не определяют переменные и содержат только повторно используемый код;
- ♦ вся функциональность принадлежит пространству имен. Это важно не только для организации кода, но и для корректности его работы. Это связано с тем, что *правило*

*единственного определения* C++ позволяет компоновщику выбирать любую подходящую функцию из отдельно скомпилированных модулей, и результат получается непредсказуемый. Например, во многих файлах может быть определен оператор потока вывода для строки векторов, несмотря на то, что определять операторы для посторонних типов — плохая идея;

- ◆ не рекомендуется применять оператор `using` к пространству имен или реализовывать функциональность в заголовке, хотя оба метода удобны, а последний еще и совместим с шаблонами. В файлах `*.cpp` заголовок включается в файл первым, чтобы обеспечить его самодостаточность. Также неплохо сначала включить больше локальных файлов.

Еще пара полезных функций:

- ◆ частное от целочисленного деления:

```
long long ceiling(unsigned long long n, long long divisor)
{return n/divisor + bool(n % divisor);}
```

- ◆ удобные обертки вокруг `new` и `delete`. Помните, что они не вызывают конструкторы и деструкторы и должны использоваться парами:

```
template<typename ITEM> ITEM* rawMemory(int n)
{return (ITEM*)::operator new(sizeof(ITEM) * n);}
void rawDelete(void* array){::operator delete(array);}
template<typename ITEM> void rawDestruct(ITEM* array, int size)
{
    for(int i = 0; i < size; ++i) array[i].~ITEM();
    rawDelete(array);
}
```

- ◆ универсальный оператор присваивания — уничтожает цель и копирует в нее значение с помощью размещения `new`. Несмотря на популярность, этот метод дает технически неопределенное поведение, потому что разрушенный объект не должен использоваться в соответствии со стандартом:

```
template<typename TYPE> TYPE& genericAssign(TYPE& to, TYPE const& rhs)
{ // в начале выполняется самопроверка
    if(&to != &rhs)
    {
        to.~TYPE();
        new(&to)TYPE(rhs);
    }
    return to;
}
```

- ◆ пара ключ-значение — чтобы избежать использования универсальных операторов `first` и `second` вместо более понятных значений, особенно в словарях (подробнее об этом — в последующих главах):

```
template<typename KEY, typename VALUE> struct KVPair
{
    KEY key;
    VALUE value;
```

```
KVPair(KEY const& theKey = KEY(), VALUE const& theValue = VALUE()):
    key(theKey), value(theValue) {}
};
```

- ◆ пустая структура — на случай, если шаблону нужен тип, но ничего значимого в нем не хранится:

```
struct EMPTY{};
```

Многие алгоритмы — например, алгоритмы сортировки и поиска выполняют *сравнения*. Алгоритмы в библиотеках вроде STL и ей подобных предполагают, что пользователь предоставит для задачи компараторы. Для использования универсальных алгоритмов в STL технически необходимо определить только «<». Например, можно реализовать «==» в его терминах, потому что

$$x = y \leftrightarrow !(x < y) \&\&!(y < x).$$

Поэтому определите универсальные операторы, которые используются компилятором, если для типа они не определены:

```
template<typename ITEM> bool operator==(ITEM const& lhs, ITEM const& rhs)
{return lhs <= rhs && lhs >= rhs;}
template<typename ITEM> bool operator!=(ITEM const& lhs, ITEM const& rhs)
{return !(lhs == rhs);}
```

Большинство алгоритмов ради универсальности использует вместо операторов более общие *компараторы*. Например, попробуйте найти минимум в массиве указателей на объекты. Оператор сравнивает указатели, а компаратор, который разыменовывает элементы перед их сравнением, может быть указан вызывающим. В соглашении C++ STL для сравнения используется оператор(), а «==» не поддерживается. Для эффективности мои компараторы поддерживают «==», чтобы избежать двойного вызова «<», но в остальном они совместимы. По умолчанию выполняется приведение к операторам:

```
template<typename ITEM> struct DefaultComparator
{
    bool operator()(ITEM const& lhs, ITEM const& rhs) const {return lhs < rhs;}
    bool isEqual(ITEM const& lhs, ITEM const& rhs) const {return lhs == rhs;}
};
template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM> >
```

Часто требуется сравнение в обратном порядке, т. е. « $a < b$ » возвращает « $b < a$ »:

```
struct ReverseComparator
{
    COMPARATOR c;
    ReverseComparator(COMPARATOR const& theC = COMPARATOR()): c(theC) {}
    bool operator()(ITEM const& lhs, ITEM const& rhs) const {return c(rhs, lhs);}
    bool isEqual(ITEM const& lhs, ITEM const& rhs) const
    {return c.isEqual(rhs, lhs);}
};
```

Многие компараторы, такие как указатели и индексы массивов, являются частными случаями преобразования и сравнения. Поэтому реализуйте их с соответствующими преобразованиями:

```
template<typename ITEM, typename TRANSFORM, typename COMPARATOR>
struct TransformComparator
{
    TRANSFORM t;
    COMPARATOR c;
    TransformComparator(TRANSFORM const& theT = TRANSFORM(),
        COMPARATOR const& theC = COMPARATOR()): t(theT), c(theC) {}
    // требуется, если преобразование конструируется по умолчанию,
    // а компаратор - нет
    TransformComparator(COMPARATOR const& theC): c(theC) {}
    bool operator()(ITEM const& lhs, ITEM const& rhs) const
    {return c(t(lhs), t(rhs));}
    bool isEqual(ITEM const& lhs, ITEM const& rhs) const
    {return c.isEqual(t(lhs), t(rhs));}
};

template<typename ITEM> struct PointerTransform
{
    ITEM const& operator()(ITEM const* item) const {assert(item); return *item;}
};

template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM> >
using PointerComparator =
    TransformComparator<ITEM const*, PointerTransform<ITEM>, COMPARATOR>;

template<typename ITEM> struct IndexTransform
{
    ITEM const* array;
    IndexTransform(ITEM* theArray): array(theArray) {}
    ITEM const& operator()(int i) const {return array[i];}
};

template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM> >
using IndexComparator =
    TransformComparator<int, IndexTransform<ITEM>, COMPARATOR>;
```

Тот же подход используется для сравнения пар по первому значению:

```
template<typename FIRST, typename SECOND> struct PairFirstTransform
{
    FIRST const& operator()(pair<FIRST, SECOND> const& p) const {return p.first;}
};

template<typename FIRST, typename SECOND, typename COMPARATOR =
    DefaultComparator<FIRST> > using PairFirstComparator = TransformComparator<
    pair<FIRST, SECOND>, PairFirstTransform<FIRST, SECOND>, COMPARATOR>;
```

Массивы переменной длины обычно сравниваются *лексикографически* — т. е. по словарному порядку (например, «кот» < «мышь»). `LexicographicComparator` работает с типами, поддерживающими `operator[]` и `getSize`. Элементы после последнего неявно приводятся к `null`, где `null = null`. Для массивов длины  $k$  сравнение занимает  $O(k)$  времени, но в среднем меньше — в зависимости от длины общего префикса (о векторах мы поговорим в этой главе далее). Для простоты элементы сравниваются с использованием операторов, а не переданного компаратора:

```
template<typename VECTOR> struct LexicographicComparator
{ // первые два функтора относятся к i-му элементу
    bool operator()(VECTOR const& lhs, VECTOR const& rhs, int i) const
```



```

{
    return i < lhs.getSize() ? i < rhs.getSize() && lhs[i] < rhs[i] :
        i < rhs.getSize();
}
bool isEqual(VECTOR const& lhs, VECTOR const& rhs, int i) const
{
    return i < lhs.getSize() ? i < rhs.getSize() && lhs[i] == rhs[i] :
        i >= rhs.getSize();
}
bool isEqual(VECTOR const& lhs, VECTOR const& rhs) const
{
    for(int i = 0; i < min(lhs.getSize(), rhs.getSize()); ++i)
        if(lhs[i] != rhs[i]) return false;
    return lhs.getSize() == rhs.getSize();
}
bool operator()(VECTOR const& lhs, VECTOR const& rhs) const
{
    for(int i = 0; i < min(lhs.getSize(), rhs.getSize()); ++i)
    {
        if(lhs[i] < rhs[i]) return true;
        if(rhs[i] < lhs[i]) return false;
    }
    return lhs.getSize() < rhs.getSize();
}
int getSize(VECTOR const& value) const {return value.getSize();}
};

```

Необходимо позаботиться о правильной реализации операции «<». Например, часто необходимо выполнять сравнение по комбинации приоритетов. Логика типа «если условие  $x$ , то сравнить переменную 1, иначе переменную 2» не нуждается в транзитивном сравнении, т. к. ложное значение  $x$  может давать сравнение между  $a$ ,  $b$  и  $a < b$ , ложное  $x$  — между  $b$ ,  $c$  и  $b < c$  и истинное  $x$  — между  $a$ ,  $c$  и  $a > c$ .

Часто требуется найти минимум или максимум в массиве по некоторому компаратору или функции, которая сравнивает элементы. Наиболее полезные из них реализованы далее:

```

template<typename ITEM, typename COMPARATOR> int argMin(ITEM* array,
    int size, COMPARATOR const& c)
{ // минимум массива с компаратором элементов
    assert(size > 0);
    int best = 0;
    for(int i = 1; i < size; ++i) if(c(array[i], array[best])) best = i;
    return best;
}
template<typename ITEM> int argMin(ITEM* array, int size)
{return argMin(array, size, DefaultComparator<ITEM>());}
template<typename ITEM> int argMax(ITEM* array, int size)
{return argMin(array, size, ReverseComparator<ITEM>());}
template<typename ITEM> ITEM& valMin(ITEM* array, int size)
{
    int index = argMin(array, size);

```

```

    assert(index > -1);
    return array[index];
}
template<typename ITEM> ITEM& valMax(ITEM* array, int size)
{
    int index = argMax(array, size);
    assert(index > -1);
    return array[index];
}
template<typename ITEM, typename FUNCTION> int argMinFunc(ITEM* array,
    int size, FUNCTION const& f)
{ // максимум массива с функцией преобразования
    assert(size > 0);
    int bestIndex = -1;
    double bestScore;
    for(int i = 0; i < size; ++i)
    {
        double score = f(array[i]);
        if(bestIndex == -1 || score < bestScore)
        {
            bestIndex = i;
            bestScore = score;
        }
    }
    return bestIndex;
}
template<typename ITEM, typename FUNCTION> ITEM& valMinFunc(ITEM* array,
    int size, FUNCTION const& f)
{
    int index = argMinFunc(array, size, f);
    assert(index > -1);
    return array[index];
}

```

Еще одна полезная функция — использование некоторых распространенных арифметических операторов. Во многих случаях стандартом является реализация `operator+` в терминах `operator+=`. Общее правило состоит в том, чтобы сделать оператор функцией, не являющейся членом, или дружественной функцией, если это возможно или удобно, в противном случае — функцией-членом. С шаблонами дружественная функция работает лучше, чем не член. В приведенном далее коде создается тип, который может быть унаследован с типом `T` = наследник. Это позволяет последнему автоматически использовать приведенные здесь операторы, которые реализованы в его пользовательских версиях `*=`. Помните, что код шаблона создается только при использовании, поэтому этот класс можно безопасно наследовать, даже если в нем нет всех операторов `*=`:

```

template<typename T> struct ArithmeticType
{
    friend T operator+(T const& a, T const& b)
    {
        T result(a);
        return result += b;
    }
}

```

```

friend T operator-(T const& a, T const& b)
{
    T result(a);
    return result -= b;
}
friend T operator*(T const& a, T const& b)
{
    T result(a);
    return result *= b;
}
friend T operator<<(T const& a, int shift)
{
    T result(a);
    return result <<= shift;
}
friend T operator>>(T const& a, int shift)
{
    T result(a);
    return result >>= shift;
}
friend T operator%(T const& a, T const& b)
{
    T result(a);
    return result %= b;
}
friend T operator/(T const& a, T const& b)
{
    T result(a);
    return result /= b;
}
};

```

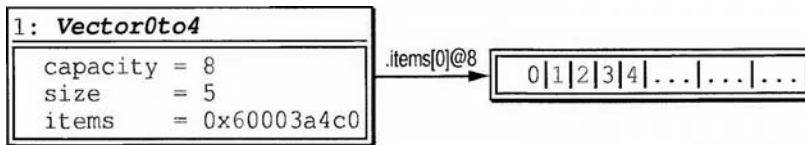
Для нахождения числа  $\pi$  можно вычислить выражение  $4 * \text{atan}(1)$ . Но можно просто использовать константу с точностью чуть большей, чем требуемая:

```
double PI() {return 3.1415926535897932384626433832795;}
```

## 5.3. Вектор

Массив динамического размера — это самая простая и самая полезная структура данных, даже если она неэффективна, но не является узким местом. Такие массивы позволяют моделировать коллекции вроде списков покупок, но без каких-либо дополнительных свойств, которые могут быть полезны для эффективной работы. Например, прежде чем я узнал о других структурах данных, я реализовал очень хорошую игру-лабиринт, используя в качестве единственной структуры данных массив размерности 7 (!). Даже в случае применимости более сложных структур данных, которых в библиотеке нет и которые необходимо реализовывать и тестировать с нуля, удобно начинать с менее эффективного вектора.

Начальный и минимальный размеры массива должны быть малыми степенями двойки, чтобы сократить количество вызовов диспетчера памяти и не использовать лишнюю



**Рис. 5.1.** Структура памяти вектора, содержащего целые элементы 0–4.  
(Этот и другие рисунки структур данных созданы инструментом ddd, у которого есть свой синтаксис для маркировки ссылок-указателей)

память. В динамически распределяемом массиве размера `capacity` первые `size` элементов создаются сразу (рис. 5.1).

Базовый код объявляет элементы данных и разрешает доступ к ним. Помните, что в C++ функция `const` предназначена для чтения, а остальные — для записи, а конструктор перемещения и оператор присваивания удаляются, если они не определены:

```
template<typename ITEM> class Vector: public ArithmeticType<Vector<ITEM> >
{
    enum{MIN_CAPACITY = 8};
    int capacity, size;
    ITEM* items;
public:
    ITEM* getArray(){return items;}
    ITEM* const getArray()const{return items;}
    int getSize()const{return size;}
    ITEM& operator[](int i)
    {
        assert(i >= 0 && i < size);
        return items[i];
    }
    ITEM const& operator[](int i)const
    {
        assert(i >= 0 && i < size);
        return items[i];
    }
};
```

Конструкторы создают вектор некоторого размера из заданного элемента или не нужны в конструкторе элемента по умолчанию. C++ STL дополнительно использует метод `reserve()`, чтобы избежать ненужных перераспределений памяти, но такой метод довольно неуклюж и используется редко:

```
explicit Vector(): capacity(MIN_CAPACITY), size(0),
    items(rawMemory<ITEM>(capacity)) {} // дефолтный конструктор
                                     // ITEM не требуется
explicit Vector(int initialSize, ITEM const& value = ITEM()): size(0),
    capacity(max(initialSize, int(MIN_CAPACITY))),
    items(rawMemory<ITEM>(capacity))
{for(int i = 0; i < initialSize; ++i) append(value);}
Vector(Vector const& rhs): capacity(max(rhs.size, int(MIN_CAPACITY))),
    size(rhs.size), items(rawMemory<ITEM>(capacity))
{for(int i = 0; i < size; ++i) new(&items[i]) ITEM(rhs.items[i]);}
Vector& operator=(Vector const& rhs){return genericAssign(*this, rhs);}
~Vector(){rawDestruct(items, size);}
```

Основная операция модификации — *добавление* элементов в конец. В векторах используется *удвоение массива* (подробнее см. в главе 12. *Разные алгоритмы и методы*), чтобы при необходимости освободить место для дополнительных элементов (рис. 5.2):

1. Выделить новый массив емкостью  $= 2 \times$  размер.
2. Скопировать все элементы в него.
3. Высвободить старый массив.

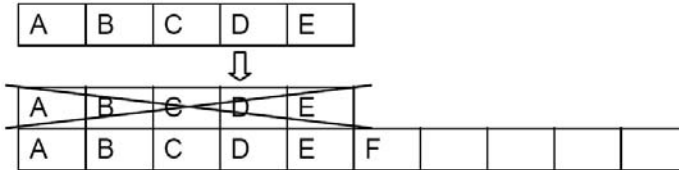


Рис. 5.2. Изменение структуры памяти во время удвоения с добавлением элемента F

В результате для добавления  $n$  элементов выполняется  $O(\lg(n))$  вызовов менеджера памяти. Добавление в наихудшем случае дает сложность  $O(n)$  и амортизированную  $O(1)$ , потому что с момента последнего изменения размера можно вставить  $n/2$  элементов.:

```
void resize()
{ // выделение
    ITEM* oldItems = items;
    capacity = max(2 * size, int(MIN_CAPACITY));
    items = rawMemory<ITEM>(capacity);
    for(int i = 0; i < size; ++i) new(&items[i])ITEM(oldItems[i]); // копирование
    rawDestruct(oldItems, size); // освобождение
}

void append(ITEM const& item)
{
    if(size >= capacity) resize();
    new(&items[size++])ITEM(item);
}
```

Удаление последнего элемента изменяет емкость массива до его удвоенного размера, если размер  $< 1/4$  от емкости (рис. 5.3).

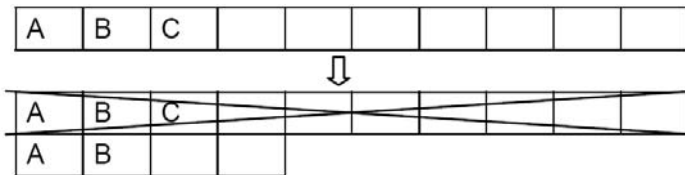


Рис. 5.3. Структура памяти изменяется при удалении элемента C

При использовании  $1/4$  емкости или других ее значений  $< 1/2$  вставка и удаление дают амортизированную сложность  $O(1)$ :

```
void removeLast()
{
    assert(size > 0);
```

```

    items[--size].~ITEM();
    if(capacity > MIN_CAPACITY && size * 4 < capacity) resize();
}

```

Преимущества векторов по сравнению с другими структурами динамического массива:

- ◆ максимально быстрый произвольный доступ;
- ◆ эффективный с точки зрения кеша перебор;
- ◆ может передаваться как непрерывный массив в C API.

Недостатки:

- ◆ вставка сложностью  $O(n)$  (не часть API), наиболее ощутимая для дорогостоящих элементов;
- ◆ после удвоения половина нового массива не используется;
- ◆ слишком большой массив выделить не удастся из-за ограничений ОС;
- ◆ перераспределение делает ссылки на элементы недействительными.

Вектор STL не сжимается при удалении. Чтобы сжать его размер, нужно заменить его временным пустым вектором и присвоить ему значение. Для перехода к C API используйте синтаксис `&vector[0]`. Задавать `size = 0` технически неправильно (потому что STL не сдвигает границы), но моя функция `getArray()` всегда работает правильно.

Обмен сложностью  $O(1)$ :

```

void swapWith(Vector& other)
{
    swap(items, other.items);
    swap(size, other.size);
    swap(capacity, other.capacity);
}

```

Для удобства поддерживается работа с последним элементом, реверсирование, арифметические операции (обратите внимание на отсутствие «+» и «-», которые используют общие операторы, представленные ранее), 2-норма и добавление значения в конец векторов:

```

ITEM const& lastItem()const{return items[size - 1];}
ITEM& lastItem(){return items[size - 1];}
void reverse(int left, int right)
    {while(left < right) swap(items[left++], items[right--]);}
void reverse(){reverse(0, size - 1);}
void appendVector(Vector const& rhs)
    {for(int i = 0; i < rhs.getSize(); ++i) append(rhs[i]);}
Vector& operator+=(Vector const& rhs)
{
    assert(size == rhs.size);
    for(int i = 0; i < size; ++i) items[i] += rhs.items[i];
    return *this;
}
Vector& operator-=(Vector const& rhs)
{
    assert(size == rhs.size);

```

```

    for(int i = 0; i < size; ++i) items[i] -= rhs.items[i];
    return *this;
}
template<typename SCALAR> Vector& operator*=(SCALAR const& scalar)
{
    for(int i = 0; i < size; ++i) items[i] *= scalar;
    return *this;
}
friend Vector operator*(Vector const& a, ITEM const& scalar)
{
    Vector result(a);
    return result *= scalar;
}
friend ITEM dotProduct(Vector const& a, Vector const& b)
{
    assert(a.size == b.size);
    ITEM result(0);
    for(int i = 0; i < a.size; ++i) result += a[i] * b[i];
    return result;
}
Vector operator-()const{return *this * -1;}
bool operator==(Vector const& rhs)const
{
    if(size == rhs.size)
    {
        for(int i = 0; i < size; ++i)
            if(items[i] != rhs[i]) return false;
        return true;
    }
    return false;
}
double norm(Vector<double> const& x){return sqrt(dotProduct(x, x));}

```

Здесь 2-норма реализована не самым стабильным образом (см. главу 22. *Численные алгоритмы: введение и матричная алгебра*), но вполне сгодится.

На собеседованиях часто дают задачу реализовать функцию C++ STL `remove_if`, которая при наличии массива и критерия фильтра перемещает соответствующие элементы в конец массива. Она возвращает новый конечный указатель и сохраняет порядок сохраненных элементов. Лучше всего обращаться к числам с фильтром «keep\_if». Например, если есть массив [1, 2, 3, 4, 5] и  $f(x) = x \% 2$ , результатом будет [1, 3, 5, |4, 2|]. Решение состоит в том, чтобы сохранить скользящую группу элементов «нет», которая обновляется циклом слева направо, помещающим первый элемент группы после нового элемента «да». Например:

$$[|1, 2, 3, 4, 5| \rightarrow [1, |2|, 3, 4, 5| \rightarrow [1, 3, |2|, 4, 5| \rightarrow [1, 3, |2, 4|, 5| \rightarrow [1, 3, 5, |4, 2|].$$

## 5.4. Блочный массив

Можно использовать множество массивов фиксированного размера  $k$ , проиндексированных вектором указателей (рис. 5.4).

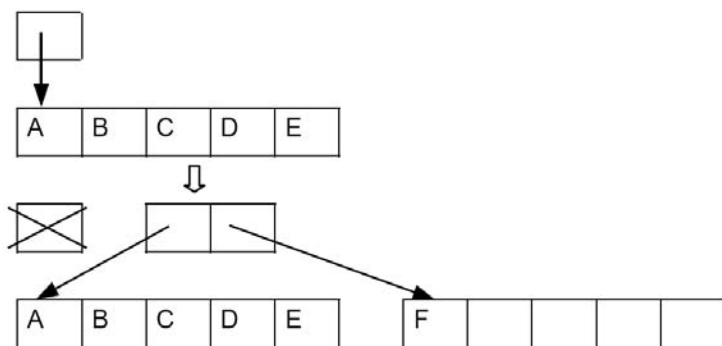


Рис. 5.4. Вставка элемента в блочный массив изменяет размер вектора указателя только при необходимости

Добавление выполняется за  $O(n/k)$  и амортизируется к  $O(1)$ , потому что удвоение происходит после вставки  $n*k$  элементов и копирования  $n/k$  указателей. Здесь  $i$ -й элемент — это  $vector[i/k][i \% k]$ , а  $k$  может равняться 64 или другому не слишком большому и не слишком малому значению.

Преимущества подхода:

- ◆ неиспользуемое пространство после изменения размера составляет  $O(n/k + k)$ ;
- ◆ не требуются большие массивы;
- ◆ элементы не копируются;
- ◆ итерация достаточно эффективна с точки зрения кэширования;
- ◆ ссылки на позиции сохраняются после перераспределения.

Недостатки:

- ◆ более медленный произвольный доступ;
- ◆ не удастся перейти к C API;
- ◆ вставка и удаление выполняются за  $O(n/k)$ .

Рекомендую использовать векторы, т. к. они обеспечивают более быстрый произвольный доступ и возможность перехода к C API. Блочный массив можно использовать для сжатого представления, где массивы, содержащие 0, не выделены, или когда нужен очень большой вектор. Но такие варианты использования специфичны и редки, поэтому их реализация не предусмотрена.

Более сложные методы, использующие блоки переменного размера, имеют оптимальный  $O(1)$  произвольный доступ, добавление и удаление последнего, а также  $O(\sqrt{n})$  неиспользуемого пространства. Эксперименты показывают, что по сравнению с другими структурами динамического массива вектор работает лучше всего, а блочный массив — на втором месте. Блочный массив лучше всего работает, когда вектор приближается к пределу памяти (см. [5.2]).

## 5.5. Связанный список

Для представления последовательности элементов вместо массива можно использовать список элементов, связанных указателями (рис. 5.5).



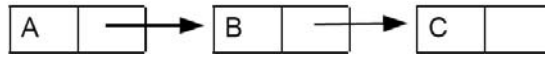


Рис. 5.5. Простой односвязный список

Такой список может быть *двусвязным* — с двунаправленными указателями или *циклическим*, когда последний элемент указывает на первый. В связных списках поддерживается сдвиг узла за  $O(1)$ , но:

- ◆ нет произвольного доступа, если не хранить указатели снаружи. Иначе для доступа к  $i$ -му элементу требуется время  $O(i)$ ;
- ◆ впустую тратится место на навигационные указатели и учет распределителя памяти узла;
- ◆ итерация не является эффективной с точки зрения кеша из-за скачков указателя.

Если элементов немного, а произвольный доступ не требуется, список может работать лучше, чем вектор. В таких реализациях для повышения эффективности обычно используются собственные связанные списки, а не общие. Для примера можете взглянуть на реализацию цепочки хеш-таблиц (см. главу 9. *Хеширование*). Интересным приемом является реализация структур данных, состоящих из узлов, связанных указателями, поверх массива, где каждый элемент массива является узлом. Прием позволяет избежать динамическое выделение памяти, но устраняет большинство преимуществ списка, поэтому он полезен только в особых случаях.

Приведенная далее реализация двусвязного списка позволяет *добавлять* (в начало и в конец), удалять и перемещать узлы (рис. 5.6). Она также допускает двунаправленную итерацию, подобную STL, и инкапсулирует использование итераторов почти для всех операций. Такие операции полезны для некоторых других структур данных. Основная трудность заключается в правильном повторном связывании указателей для затронутого узла, предыдущего узла и следующего узла. Конструктор является шаблонным и принимает один аргумент, который является либо `ITEM`, либо чем-то, что позволяет неявно создать его на месте. Это полезная общая идиома. Также для сборки мусора используются инструкции `new/delete` из C++, а не список свободных мест (об этом позже).

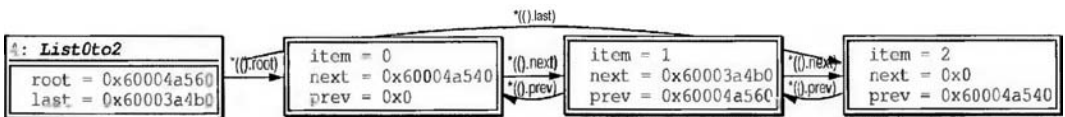


Рис. 5.6. Структура памяти двусвязного списка с целыми элементами 0–2

```
template<typename ITEM> class SimpleDoublyLinkedList
{
    struct Node
    { // обратите внимание на шаблонный конструктор
        ITEM item;
        Node *next, *prev;
        template<typename ARGUMENT>
        Node(ARGUMENT const& a): item(a), next(0), prev(0) {}
    } *root, *last;
```

```

void cut(Node* n) // отвязка узла из списка
{ // соединение предыдущего и следующего
    assert(n);
    (n == last ? last : n->next->prev) = n->prev;
    (n == root ? root : n->prev->next) = n->next;
}

public:
SimpleDoublyLinkedList(): root(0), last(0){}
template<typename ARGUMENT> void append(ARGUMENT const& a)
{ // обратите внимание на шаблонный конструктор
    Node* n = new Node(a);
    n->prev = last;
    if(last) last->next = n;
    last = n;
    if(!root) root = n;
}

class Iterator
{
    Node* current;
public:
    Iterator(Node* n): current(n){}
    typedef Node* Handle;
    Handle getHandle() return current;}
    Iterator& operator++()
    { // к следующему элементу
        assert(current);
        current = current->next;
        return *this;
    };
    Iterator& operator--()
    { // к предыдущему элементу
        assert(current);
        current = current->prev;
        return *this;
    };
    ITEM& operator*() const{assert(current); return current->item;}
    ITEM* operator->() const{assert(current); return &current->item;}
    bool operator==(Iterator const& rhs) const
        {return current == rhs.current;}
};

Iterator begin() return Iterator(root);}
Iterator rBegin() return Iterator(last);}
Iterator end() return Iterator(0);}
Iterator rEnd() return Iterator(0);}

void moveBefore(Iterator what, Iterator where)
{ // where может равняться 0, что означает переход в конец
    assert(what != end());
    if(what != where) // сначала проверяем ссылку на себя
    {
        Node *n = what.getHandle(), *w = where.getHandle();
        cut(n);
        n->next = w;
    }
}

```

```

        if (w)
        {
            n->prev = w->prev;
            w->prev = n;
        }
        else
        {
            n->prev = last;
            last = n;
        }
        if (n->prev) n->prev->next = n;
        if (w == root) root = n;
    }
}

template<typename ARGUMENT> void prepend(ARGUMENT const& a)
{ // добавление и перемещение в начало
    append(a);
    moveBefore(rBegin(), begin());
}

void remove(Iterator what)
{ // отключение и высвобождение памяти
    assert(what != end());
    cut(what.getHandle());
    delete what.getHandle();
}

SimpleDoublyLinkedList(SimpleDoublyLinkedList const& rhs)
{for(Node* n = rhs.root; n; n = n->next){append(n->item);}}
SimpleDoublyLinkedList& operator=(SimpleDoublyLinkedList const&rhs)
{return genericAssign(*this, rhs);}
~SimpleDoublyLinkedList()
{
    while(root)
    {
        Node* toBeDeleted = root;
        root = root->next;
        delete toBeDeleted;
    }
}
};

```

Связанные списки редко используются в повседневном программировании, т. к. векторы предпочтительнее. Но зато о них спрашивают на собеседованиях. Типичный вопрос: два неправильных списка сливаются в каком-то узле и продолжают как один список. Зная головные узлы, найдите узел слияния. Простое решение за  $O(n^2)$  проверяет каждый узел на предмет того, является ли он точкой слияния. Помещение узлов одного списка в хеш-таблицу и проверка их на соответствие другому ожидаемо выполняется за  $O(n)$ , но при этом тратится больше памяти. Умное решение состоит в том, чтобы просмотреть оба списка, найти их длину, затем просмотреть более длинный список, после чего смещаться по каждому списку, пока не найдется узел слияния.

## 5.6. Свободный список со сборкой мусора

*Свободный список* — это набор блоков памяти, выделенных однажды и разделенных на более мелкие объекты. Его использование позволяет не выделять множество мелких объектов, но не обеспечивает сколь-нибудь значительный выигрыш по сравнению с современными менеджерами памяти. Свободный список также может собирать мусор для структуры данных, а это уже полезно. Поскольку освобожденная память недоступна для других частей программы и не может объединять или менять местами структуры данных быстрее, чем при создании полной копии, свободные списки используются редко.

Свободный список возвращает `ITEM*` при выделении и берет его при освобождении. `ITEM*` — это приведенный `Item*`, где `Item` — структура, содержащая `ITEM` необработанной памяти в качестве первого члена и вспомогательные данные (помните, что в C++ адрес структуры равен адресу ее первого члена). Строительный блок — это одноблочный *статический свободный список* без сборки мусора. Он состоит из массива длиной  $k$ . В нем есть указатели на следующий элемент в блоке (индекс `maxSize`) и заголовок возвращаемого списка. Последний определяется неявно возвращаемыми указателями. На рис. 5.7 и в следующем коде приведен пример двух списков, хранящихся в одном массиве:

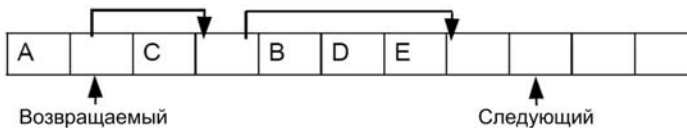


Рис. 5.7. Структура статического списка свободных мест

```
template<typename ITEM> struct StaticFreelist
{ // size - это текущий размер, а maxSize - это самое большое выделение
  int capacity, size, maxSize; // всегда выполняется size <= maxSize <= capacity
  struct Item
  {
    ITEM item;
    union
    {
      Item* next; // используется, если ячейка пуста
      void* userData; // используется при распределении
    };
  } *nodes, *returned;
  StaticFreelist(int fixedSize): capacity(fixedSize), size(0), maxSize(0),
    returned(0), nodes(rawMemory<Item>(fixedSize)){}
  bool isFull(){return size == capacity;}
  bool isEmpty(){return size <= 0;}
};
```

### Распределение:

1. Если возвращенный список не пуст, используется его первый узел.
2. В противном случае, если блок заполнен, вызывающая сторона должна создать еще один.

### 3. Используется следующий элемент:

```
Item* allocate()
{ // заполненные блоки обрабатываются снаружи
  // с помощью арифметики указателей
  assert(!isFull());
  Item* result = returned;
  if(result) returned = returned->next;
  else result = &nodes[maxSize++];
  ++size;
  return result;
}
```

Стоимость равна  $O(\text{вызовы конструктора} + \text{вызовы менеджера памяти})$ . Последнее амортизируется за счет размера блока.

#### Высвобождение:

1. Уничтожить элемент.
2. Пометить его как уничтоженный.
3. Добавить ячейку в возвращаемый список (это делается в пункте 2).

```
void remove(Item* item)
{ // узлы берутся из списка
  assert(item - nodes >= 0 && item - nodes < maxSize);
  item->item.~ITEM();
  item->next = returned;
  returned = item;
  --size;
}
```

Использование пространства равно  $O(\text{максимальное количество выделяемых элементов})$ . Деструктор собирает мусор, уничтожая все невозвращенные элементы, а это выполняется за  $O(1)$ , если возвращены все элементы, и  $O(\text{maxSize})$  в противном случае:

```
~StaticFreelist()
{
  if(!isEmpty())
  { // отмечаются выделенные узлы, снимается отметка
    // с возвращенных, отмеченные уничтожаются
    Vector<bool> toDestruct(maxSize, true);
    while(returned)
    { // перебор списка возврата, чтобы снять отметку
      toDestruct[returned - nodes] = false;
      returned = returned->next;
    }
    for(int i = 0; i < maxSize; ++i)
      if(toDestruct[i]) nodes[i].item.~ITEM();
  }
  rawDelete(nodes);
}
```

Обычно для реализации свободного списка нужен двусвязный список статических свободных списков, а элементы должны знать, из какого статического списка они пришли.

Каждый статический список образует блок. Структура получается похожей на блочный массив. Список разделен так, что полные блоки находятся сзади, а неполные — впереди.

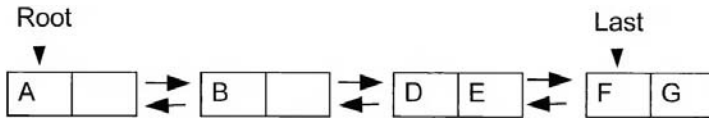


Рис. 5.8. Структура динамического свободного списка

Размеры блоков увеличиваются в 2 раза, с 32 до 8192, что позволяет достичь полезности и эффективности при небольших размерах. Использование памяти в маловероятном худшем случае будет равно  $O(\min(\max \text{ количество элементов, выделенных в любой момент времени} \times (\text{sizeof}(\text{ITEM}) + \text{sizeof}(\text{указатель})), \text{ количество выделенных элементов} \times \text{размер блока}))$ , что происходит, если в каждом блоке хранится по одному элементу. Для сборки мусора общая стоимость равна  $O(\text{количество живых элементов} \times \text{стоимость деструктора} + \text{общий размер оставшихся блоков})$ , как показано на рис. 5.9 и в следующем коде.

```
template<typename ITEM> class Freelist
{
    enum{MAX_BLOCK_SIZE = 8192, MIN_BLOCK_SIZE = 8, DEFAULT_SIZE = 32};
    int blockSize;
    typedef SimpleDoublyLinkedList<StaticFreelist<ITEM> > ListType;
    typedef typename StaticFreelist<ITEM>::Item Item;
    typedef typename ListType::Iterator I;
    ListType blocks;
    // запрет копирования
    Freelist(Freelist const&);
    Freelist& operator=(Freelist const&);
public:
    Freelist(int initialSize = DEFAULT_SIZE): blockSize(max<int>(
        MIN_BLOCK_SIZE, min<int>(initialSize, MAX_BLOCK_SIZE))) {}
};
```

### Выделение:

1. Используется первый блок, который при необходимости сначала создается.
2. Элементы помещаются в него.
3. Если блок заполнится, он перемещается в конец.

```
ITEM* allocate()
{
    I first = blocks.begin();
    if(first == blocks.end() || first->isFull())
    { // проверка, нужен ли первый блок
        blocks.prepend(blockSize);
        first = blocks.begin();
        blockSize = min<int>(blockSize * 2, MAX_BLOCK_SIZE);
    } // выделение элемента
    Item* result = first->allocate();
}
```

```

result->userData = (void*)first.getHandle(); // указатель списка блока
// перемещение блока в конец
if(first->isFull()) blocks.moveBefore(first, blocks.end());
return (ITEM*)result; // приведение работает по правилу первого члена
}

```

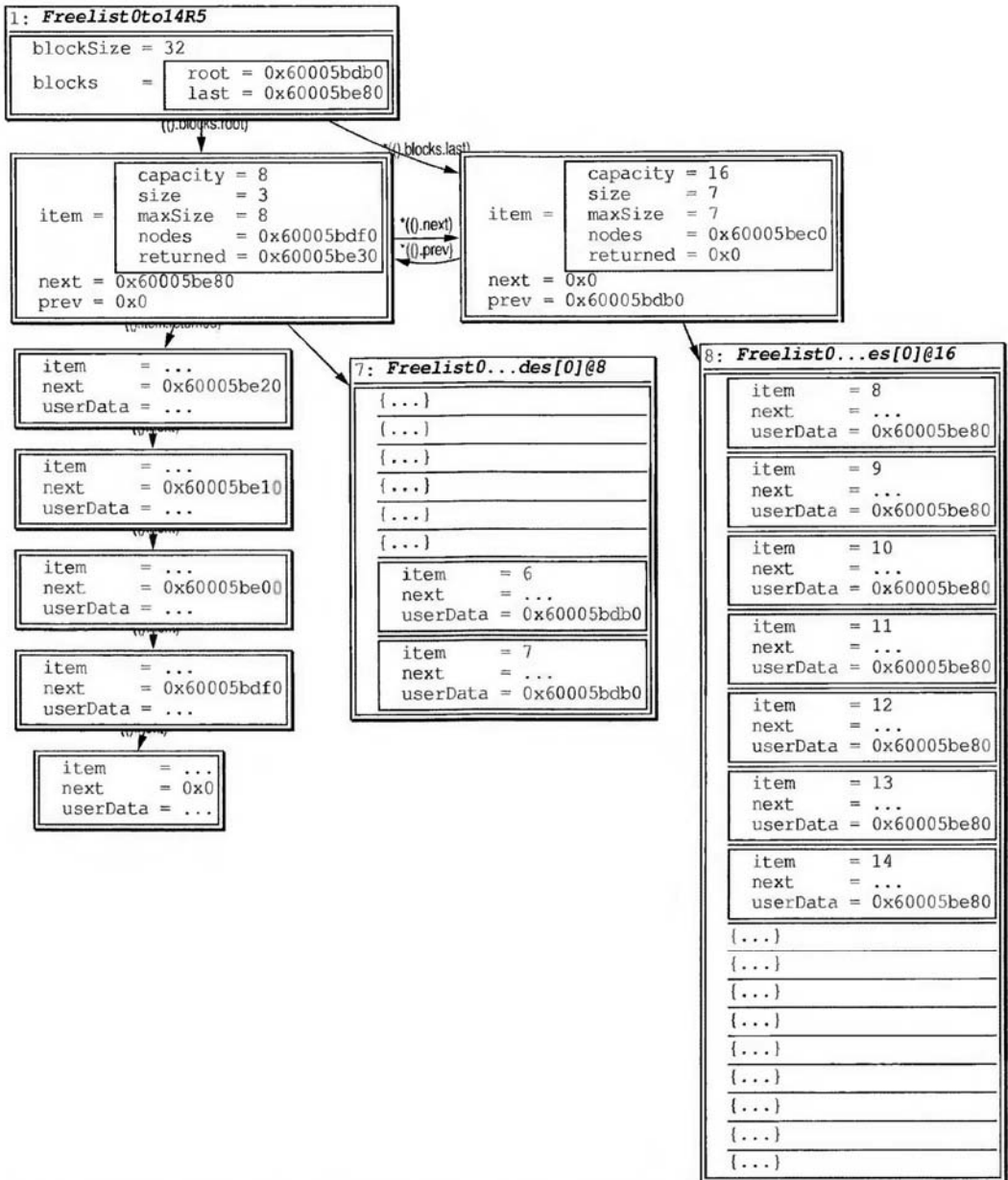


Рис. 5.9. Структура памяти свободного списка с выделенными целочисленными элементами 0–14 и освобожденными элементами 0–5

## Высвобождение:

1. Элемент возвращается в блок, из которого он возник.
2. Если блок стал пустым, он высвобождается.
3. Иначе перемещается в начало.

```
void remove(ITEM* item)
{ // если элемент не из списка, возникает неопределенное поведение
  if(!item) return; // работа с указателем null
  Item* node = (Item*)(item); // приведение из первого члена
  I cameFrom((typename I::Handle)node->userData);
  cameFrom->remove(node);
  if(cameFrom->isEmpty())
  { // удаление блока, если он пустой, либо уменьшение его размера
    // обратите внимание, что границы блока удаляют мусорные
    // данные, но это маловероятно
    blockSize = max<int>(MIN_BLOCK_SIZE,
      blockSize - cameFrom->capacity);
    blocks.remove(cameFrom);
  } // доступные блоки перемещаются в начало
  else blocks.moveBefore(cameFrom, blocks.begin());
}
```

## 5.7. Стек

Реализуем последовательность вида *первым пришел — последним вышел*, в которой элементы можно только *добавлять* (push) *наверх* и *извлекать* (pop) *сверху* (рис. 5.10).

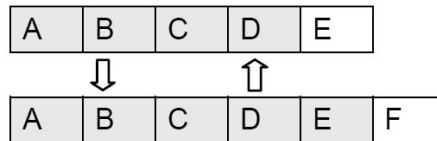


Рис. 5.10. Добавление в стек и извлечение из него

Чтобы создать такую структуру, используем последовательность, в которой вся работа происходит с последним элементом. Это может быть:

- ◆ массив фиксированного размера со счетчиком, если его максимальный размер известен и мал, а у элемента есть конструктор по умолчанию. Это быстрый и простой способ, который позволяет избежать динамического выделения памяти;
- ◆ вектор — способ быстрый, пишется малым объемом кода и приводит к небольшому выделению памяти;
- ◆ блочный массив — тот же код, что и для вектора. Такая структура эффективна для очень больших размеров или элементов, но повторяющаяся последовательность push/pop на границе блока вызывает многократное выделение/освобождение памяти.

Связанный список для этого не подходит, потому что выделение отдельных узлов неэффективно.



Вектор и блочный массив дают амортизированную сложность операций push/pop  $O(1)$  из-за изменения размера, а также поддерживают произвольный доступ, что полезно в некоторых случаях (но здесь не реализовано). Выберем первый вариант (как показано на рис. 5.11 и в следующем коде).

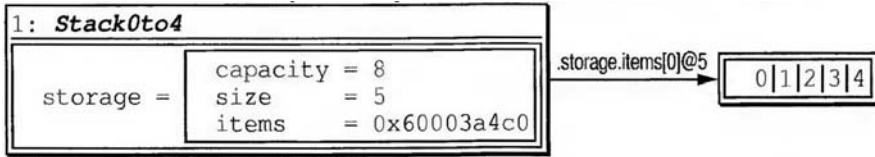


Рис. 5.11. Структура памяти векторной реализации стека с числами 0–4

```
template<typename ITEM, typename VECTOR = Vector<ITEM> > struct Stack
{
    VECTOR storage;
    void push(ITEM const& item){storage.append(item);}
    ITEM pop()
    {
        assert(!isEmpty());
        ITEM result = storage.lastItem();
        storage.removeLast();
        return result;
    }
    ITEM& getTop()
    {
        assert(!isEmpty());
        return storage.lastItem();
    }
    bool isEmpty(){return !storage.getSize();}
};
```

Маловероятно, что операции над стеком будут в программе узким местом, поэтому разработчикам следует относиться к ним как к абстрактной концепции, которая в любой библиотеке реализована достаточно хорошо. Стек обычно используется для реализации разбора простых выражений, — например, в калькуляторе, но в своем коде вы будете их применять, скорее всего, только чтобы избавиться от рекурсии.

## 5.8. Очередь

*Очередь* — это последовательность типа *первый пришел — первым вышел*, позволяющая *добавлять* элементы в конец очереди (enqueueing), а *извлекать* (dequeuing) из ее начала (рис. 5.12).

Для реализации очереди будет нужна последовательность, эффективно работающая с первым и последним элементами. Вот варианты того, что можно использовать:

- ◆ связанный список;
- ◆ *круговая очередь* с удвоением (об этом подробнее позже) — самый простой и достаточно эффективный выбор;

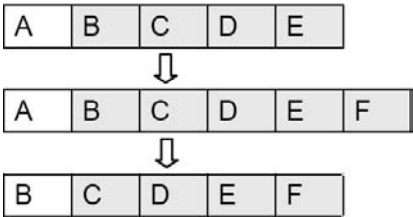


Рис. 5.12. Постановка в очередь и удаление из очереди

- ◆ циклическая очередь по блочному массиву — слегка неуклюжий, но, вероятно, лучший вариант для размеров, близких к пределу памяти;
- ◆ связанный блочный список, похожий на блочный массив, — еще более неуклюжий вариант, чем связанный список и, по всей видимости, худший из всех прочих вариантов;
- ◆ массив с дополнительным местом на обоих концах, который удваивается, когда место заканчивается — амортизированная сложность  $\text{push/pop } O(1)$ , но структура все равно неуклюжая.

В целом здесь применимы те же соображения, что и для стека. Структуру можно расширить до *двусторонней очереди*, где разрешается помещать и забирать элементы с обоих концов, но здесь мы этого делать не будем.

Циклическая очередь с удвоением реализуется через динамический массив, но элементы начинаются в начале индекса, чтобы обеспечить эффективную работу в начале. Таким образом,  $\text{индекс}(i) = (\text{начало} + i) \% \text{емкость}$ , а  $\text{конец} = \text{индекс}(\text{размер} - 1)$ . Добавление и выталкивание добавляется с обоих концов, время выполнения амортизируется  $O(1)$ . Произвольный доступ также поддерживается (рис. 5.13).

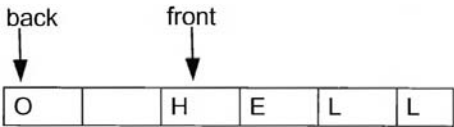


Рис. 5.13. Указатели очереди

Для простоты реализация поддерживает только обычную очередь, как показано на рис. 5.14 и в следующем коде.

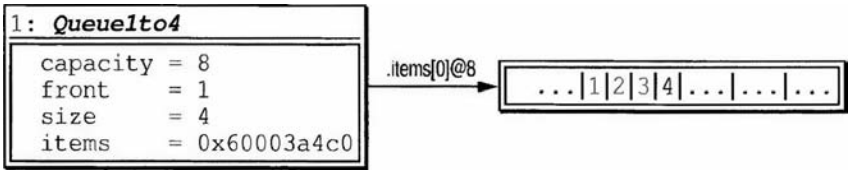


Рис. 5.14. Структура памяти очереди с вставленными целочисленными элементами 0–4 и удаленным 0

```
template<typename ITEM> class Queue
{
    enum{MIN_CAPACITY = 8}; // то же, что и для вектора
    int capacity, front, size;
```

```

ITEM* items; // объявляется после емкости
int offset(int i) const { return (front + i) % capacity; }
void resize()
{
    ITEM* oldArray = items;
    int newCapacity = max(int(MIN_CAPACITY), size * 2);
    items = rawMemory<ITEM>(newCapacity);
    for(int i = 0; i < size; ++i) new(&items[i]) ITEM(oldArray[offset(i)]);
    deleteArray(oldArray);
    front = 0;
    capacity = newCapacity;
}
void deleteArray(ITEM* array)
{ // удаляются только выделенные элементы
    for(int i = 0; i < size; ++i) array[offset(i)].~ITEM();
    rawDelete(array);
}
public:
bool isEmpty() const { return size == 0; }
int getSize() const { return size; }
ITEM& operator[](int i)
{
    assert(i >= 0 && i < size);
    return items[offset(i)];
}
ITEM const& operator[](int i) const
{
    assert(i >= 0 && i < size);
    return items[offset(i)];
}
Queue(int theCapacity = MIN_CAPACITY): capacity(max(int(MIN_CAPACITY),
    theCapacity)), front(0), size(0), items(rawMemory<ITEM>(capacity)) {}
Queue(Queue const& rhs): capacity(max(int(MIN_CAPACITY), rhs.size)),
    size(rhs.size), front(0), items(rawMemory<ITEM>(capacity))
    {for(int i = 0; i < size; ++i) push(rhs[i]);}
Queue& operator=(Queue const& rhs) { return genericAssign(*this, rhs); }
~Queue() { deleteArray(items); }
void push(ITEM const& item)
{
    if(size == capacity) resize();
    new(&items[offset(size++)]) ITEM(item);
}
ITEM pop()
{
    assert(!isEmpty());
    ITEM result = items[front];
    items[front].~ITEM();
    front = offset(1);
    if(capacity > 4 * --size && capacity > MIN_CAPACITY) resize();
    return result;
}

```

```
ITEM& top() const
{
    assert(!isEmpty());
    return items[front];
}
};
```

В специальных, в основном аппаратных, приложениях можно упростить циклическую очередь до циклического буфера фиксированного размера, но в этом особой нужды нет, поскольку операции с очередью обычно не являются узким местом.

Очереди — это важный строительный блок некоторых алгоритмов, таких как сортировка на диске (см. главу 13. *Алгоритмы для работы с внешней памятью*). Но в основном они используются, когда этого требует логическая модель чего-либо — например, для имитации времени ожидания в очереди. Во многих задачах лучше подходит очередь с более общим приоритетом (см. главу 10. *Приоритетные очереди*).

Интересный вопрос с собеседования — как реализовать API очередей с двумя стеками. Лучшее решение — это помещать попавшие в очередь элементы в стек<sub>1</sub>. А чтобы удалить элемент из очереди, возьмите элемент из стека<sub>2</sub>, если он есть. В противном же случае переместите все элементы стека<sub>1</sub> в стек<sub>2</sub>. Операция выполняется за амортизированное время  $O(1)$ .

## 5.9. Деревья

*Дерево* похоже на связанный список, но в его узле хранятся указатели на все дочерние элементы, и в нем не должно быть циклических соединений. Узел, у которого нет дочерних элементов, является *внешним/листовым*, а если они есть — то *внутренним*. У дерева, в котором есть  $n$  узлов, имеется  $n - 1$  дочерний указатель. Дерево является *бинарным*, если каждый его узел имеет не более двух дочерних элементов. Бинарные деревья легко представить в коде, поскольку у них ограниченное число дочерних элементов, из которых образуется множество структур данных. Для *упорядоченного дерева* (рис. 5.15) выполняется соотношение «данные в левом узле  $<$  данных в родительском узле  $\leq$  данным в правом узле».

Обычное дерево изоморфно бинарному дереву, если используется отображение (первый дочерний элемент, следующий соседний элемент)  $\leftrightarrow$  (левый дочерний элемент,

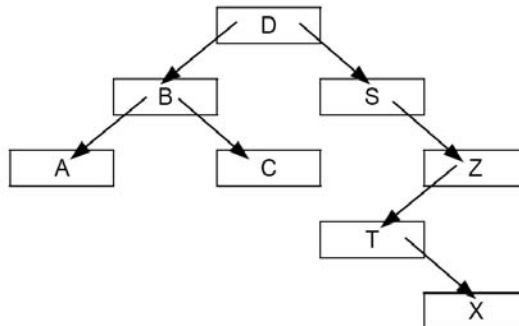


Рис. 5.15. Упорядоченное бинарное дерево

правый дочерний элемент). Чтобы представить его напрямую, нужен список дочерних указателей переменной длины, работать с которым неудобно. Структура полезная, если логика предметной области для нее подходит.

Узел может иметь указатель на своего родителя. В отличие от связанных списков, деревья полезны только с дополнительной логикой, а в последующих главах мы обсудим конкретные реализации.

## 5.10. Битовые алгоритмы

В C++ нет команд для вычисления  $\lg$  (двоичный логарифм), *pop count* (вычисления количества установленных битов в слове) и множества других операций, которые на ассемблере выполняются за время  $O(1)$ . Запомните:

- ◆ сдвиги на количество битов меньше нуля или больше длины слова в языке не определены;
- ◆ при сдвиге вправо слово со знаком заполняется единицами, а не нулями;
- ◆ функция `numeric_limits<WORD>::digits` возвращает результат на 1 меньше для слов со знаком;
- ◆  $1 = \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long}) \leq \text{sizeof}(\text{long long})$ , а `unsigned long long` имеет длину  $\geq 64$  бита. Например, в зависимости от архитектуры компьютера, `int` может иметь 32 или 64 бита, `long long` — 64 или 128 битов, а указатель или `int` имеют больше битов, чем другие;
- ◆ `<stdint>` определяет `intX_t` и `uintX_t` для  $x = 8, 16, 32$  и  $64$ , но в реализации может не учитываться  $x$ , для которого типа слова не существует. Таким образом, в общем случае эти типы могут быть переносимыми, хотя на практике это случается редко, а приведенный далее вариант неуклюж;
- ◆ `int_leastX_t` и `uint_leastX_t` для  $x = 8, 16, 32$  и  $64$  имеют по крайней мере указанное количество битов. Используйте их с масками `0xFF...ull` для имитации любых недоступных типов с фиксированной шириной.

Если вы полагаетесь на размеры слов, используйте переносимые типы с фиксированной шириной. При использовании общего типа слова при проверке нужного количества битов возможны ошибки. С любой необычной архитектурой приходится работать разными способами.

Битовые операции позволяют эффективно работать со степенями двойки для положительных целых чисел:

- ◆  $[\lg(x)]$  = позиция старшего установленного бита;
- ◆ если  $x$  является степенью двойки, то у числа  $x - 1$  все младшие биты равны единице. В обоих случаях используется *битовый параллелизм* — т. е. работа с несколькими битами одновременно с помощью операции на словах.

```
unsigned long long twoPower(int x){return 1ull << x;}
// например, для числа 8 имеем !(0100 и 0011).
// Осторожно: возвращает true для 0
bool isPowerOfTwo(unsigned long long x){return !(x & (x - 1));}
```

```
int lgFloor(unsigned long long x)
{ // сдвиг до 0 и подсчет. Например, lgFloor(2) = lgFloor(3) = 1
  assert(x); // логарифм 0 не определен
  int result = 0;
  while(x >= 1) ++result;
  return result;
}
```

Вдобавок к этому можно эффективно реализовать  $\lceil \lg(x) \rceil$  и следующую степень двойки  $x$ :

```
// Например, lgCeiling(3) = 2
int lgCeiling(unsigned long long x){return lgFloor(x) + !isPowerOfTwo(x);}
// Например, nextPowerOfTwo(7) = nextPowerOfTwo(8) = 8
unsigned long long nextPowerOfTwo(unsigned long long x)
{return isPowerOfTwo(x) ? x : twoPower(lgFloor(x) + 1);}
```

**Битовая маска** — это слово, применение к которому битовой операции и  $x$  дает желаемый результат. Чтобы получить, установить или перевернуть один бит в позиции  $i$ , маска для  $i$ -го бита должна равняться  $2^i$ :

```
namespace Bits
{ // этот код полезен, т. к. работает со значениями int
  unsigned long long const ZERO = 0, FULL = ~ZERO;
  bool get(unsigned long long x, int i){return x & twoPower(i);} // and
  bool flip(unsigned long long x, int i){return x ^ twoPower(i);} // xor
  template<typename WORD> void set(WORD& x, int i, bool value)
  {
    assert(!numeric_limits<WORD>::is_signed);
    if(value) x |= twoPower(i);
    else x &= ~twoPower(i);
  }
} // конец пространства имен
```

Для управления битами в слове  $x$ :

- ◆ `getValue` создает слово, состоящее из  $n$  битов, начиная с позиции  $i$  числа  $x$ ;
- ◆ `setValue` устанавливает  $n$  битов числа  $x$ , начиная с позиции  $i$ , до первых  $n$  битов значения слова;
- ◆ есть возможность использовать маски для диапазонов.

Основная задача — вычисление масок. Приведенные далее операции работают с любым беззнаковым типом, но по возможности предпочитают приведение к типу `unsigned long long` вместо шаблонов. Функции входят в пространство имен `Bits`:

```
unsigned long long upperMask(int n){return FULL << n;} // 11110000
unsigned long long lowerMask(int n){return ~upperMask(n);} // 00001111
unsigned long long middleMask(int i, int n){return lowerMask(n)<<i;} // 00111000
unsigned long long getValue(unsigned long long x, int i, int n)
{return (x >> i) & lowerMask(n);} // сдвиг, применение маски
template<typename WORD>
void setValue(WORD& x, unsigned long long value, int i, int n)
{
  assert(!numeric_limits<WORD>::is_signed);
```

```

WORD mask = middleMask(i, n); // обратите внимание на приведение
x &= ~mask;    // удаление
x |= mask & (value << i);    // установка
}

```

Чтобы вычислить число извлечений — т. е. количество установленных битов в слове, разбейте его на байты и используйте таблицу предварительно вычисленных битовых счетчиков для каждого байта:

```

class PopCount8
{
    char table[256];
public:
    static int popCountBruteForce(unsigned long long x)
    { // для вычисления таблицы
        int count = 0;
        while(x)
        { // считываем один за другим правые биты
            count += x & 1;
            x >>= 1;
        }
        return count;
    }
    PopCount8(){for(int i = 0; i < 256; ++i) table[i] = popCountBruteForce(i);}
    int operator()(unsigned char x) const{return table[x];}
};

int popCountWord(unsigned long long x) // при работе с шаблоном
                                     // эффективность сохраняется
{ // инициализация при первом вызове
    static PopCount8 p8;
    int result = 0;
    for(; x; x >>= 8) result += p8(x); // одинаковая эффективность
                                     // для всех типов слов
    return result;
}

```

Сочетание арифметических и битовых операций позволяет эффективно манипулировать правильными битами. Например, чтобы подсчитать число нулей перед первой единицей в слове, измените слово так, чтобы все биты инвертировались, и подсчитайте единицы:

```

int rightmost0Count(unsigned long long x){return popCount(~x & (x - 1));}

```

Использование таблицы — это еще один быстрый и простой способ инвертировать крайние правые биты. Этот метод используется, например, в FFT (см. главу 23. *Численные алгоритмы: работа с функциями*). Составляйте результат по одному байту за раз, начиная с байта MSD, который является байтом LSD с обратным битом исходного слова:

```

class ReverseBits8
{
    unsigned char table[256];
public:
    template<typename WORD> static WORD reverseBitsBruteForce(WORD x)

```

```

{
    assert(!numeric_limits<WORD>::is_signed);
    WORD result = 0;
    for(int i = 0; i < numeric_limits<WORD>::digits; ++i)
    {
        result = (result << 1) + (x & 1);
        x >>= 1;
    }
    return result;
}
ReverseBits8()
{
    for(int i = 0; i < 256; ++i)
        table[i] = reverseBitsBruteForce<unsigned char>(i);
}
unsigned char operator()(unsigned char x) const {return table[x];}
};
template<typename WORD> WORD reverseBits(WORD x)
{
    assert(!numeric_limits<WORD>::is_signed);
    static ReverseBits8 r8;
    WORD result = 0;
    for(int i = 0; i < sizeof(x); ++i, x >>= 8) result = (result << 8) + r8(x);
    return result;
}

```

Инвертирование младших  $n$  битов выполняется как инвертирование всего слова с соответствующими сдвигами. Оптимизация заключается в уменьшении числа байтовых сдвигов путем обнуления старших битов, при этом маскировать результат не требуется:

```

template<typename WORD> WORD reverseBits(WORD x, int n)
{
    int shift = sizeof(x) * 8 - n;
    assert(!numeric_limits<WORD>::is_signed && n > 0 && shift >= 0);
    return reverseBits<WORD>(x & Bits::lowerMask(n)) >> shift;
}

```

## 5.11. Набор битов

*Набор битов* ведет (рис. 5.16) себя как вектор логических значений, для каждого из которых `operator[]` возвращает значение, а не ссылку, поскольку каждый представлен битом в векторе слов.

Набор битов поддерживает операторы «&», «|», «^» и «~», которые ведут себя так, как если бы это было целое число с младшим значащим битом 0. Для эффективности применяйте словесные операции и слова максимально возможного размера. При этом по умолчанию задействуется `unsigned long long`, хотя на 32-разрядных машинах `unsigned int` может работать быстрее. В переносимых хранилищах `unsigned char` позволяет избежать несовместимости порядка следования байтов, поэтому при необходимости используйте этот тип.



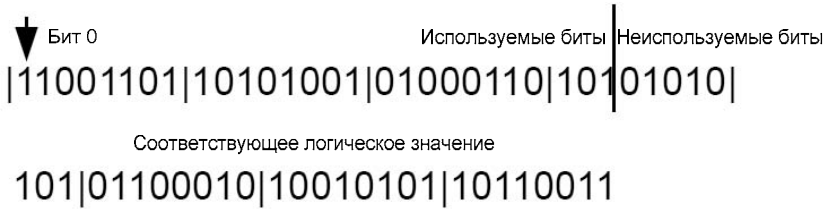


Рис. 5.16. Расположение битов в битовом наборе

Лишние биты в последнем слове — это мусор, поэтому их можно просто обнулить. Такой подход дает много преимуществ — например, возможность сравнения на равенство путем прямого сравнения векторов памяти. Пусть  $B$  — количество битов в слове. Логический бит  $i$  соответствует биту  $i \% B$  в слове  $i/B$ . Таким образом, набор битов из одного слова сопоставляется с соответствующим словом, а порядок слов с прямым порядком байтов упрощает добавление. Операции get/set находят и слово, и бит и применяют соответствующие битовые операции. Из-за использования вектора амортизированная стоимость добавления равняется  $O(1)$ , как показано на рис. 5.17 и в следующем коде.

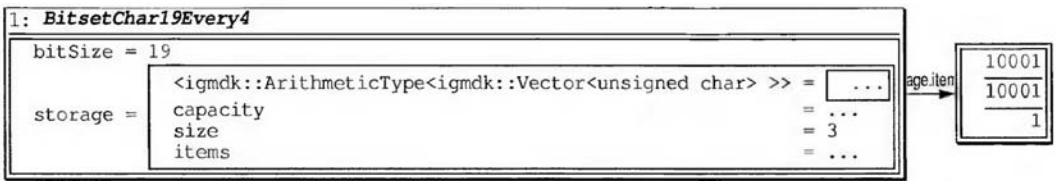


Рис. 5.17. Структура памяти набора битов с 19 битами, в котором каждый четвертый бит равен единице

```
template<typename WORD = unsigned long long> class Bitset
{
    enum{B = numeric_limits<WORD>::digits};
    unsigned long long bitSize; // объявляется до storage
    Vector<WORD> storage;
    void zeroOutRemainder()
    {if(bitSize > 0)storage.lastItem() &= Bits::lowerMask(lastWordBits());}
    bool get(int i)const
    {
        assert(i >= 0 && i < bitSize);
        return Bits::get(storage[i/B], i % B);
    }
    unsigned long long wordsNeeded()const{return ceiling(bitSize, B);}
public:
    Bitset(unsigned long long initialSize = 0): bitSize(initialSize),
        storage(wordsNeeded(), 0){} // устанавливаем все биты на 0
    Bitset(Vector<WORD> const& vector): bitSize(B * vector.getSize()),
        storage(vector) {} // полезно прямое построение из вектора storage
    int lastWordBits()const
    { // 1 в B, если > 1 бита
        assert(bitSize > 0);
        int result = bitSize % B;
```

```

    if(result == 0) result = B;
    return result;
}
int garbageBits()const{return bitSize > 0 ? B - lastWordBits() : 0;}
Vector<WORD> const& getStorage()const{return storage;}
unsigned long long getSize()const{return bitSize;}
unsigned long long wordSize()const{return storage.getSize();}
bool operator[](int i)const{return get(i);}
void set(int i, bool value = true)
{
    assert(i >= 0 && i < bitSize);
    Bits::set(storage[i/B], i % B, value);
}
void append(bool value)
{ // увеличение размера, если необходимо, и получение последнего бита
    ++bitSize;
    if(wordSize() < wordsNeeded()) storage.append(0);
    set(bitSize - 1, value);
}
void removeLast()
{
    assert(bitSize > 0);
    if(lastWordBits() == 1) storage.removeLast(); // сжатие
                                                // по возможности
    --bitSize;
    zeroOutRemainder(); // удаленный бит мог быть равен 1
}
bool operator==(Bitset const& rhs)const{return storage == rhs.storage;}
Bitset& operator&=(Bitset const& rhs)
{ // имеет смысл только для одинаковых размеров
    assert(bitSize == rhs.bitSize);
    for(int i = 0; i < wordSize(); ++i) storage[i] &= rhs.storage[i];
    return *this;
}
void flip()
{
    for(int i = 0; i < wordSize(); ++i) storage[i] = ~storage[i];
    zeroOutRemainder();
}
};

```

Путем замены `&=` на `|=` или `^=` можно получить коды для этих операций. Вы можете эффективно реализовывать и другие операции с множествами в дополнение к указанным.

Сдвиги гарантируют, что величина сдвига  $> 0$  и  $\text{mod } \text{bitSize}$ , разбивают значение на сдвиги слов и битов и применяют каждый отдельно. Часть сдвига слова перемещает слова и заполняет конец нулями. Часть сдвига битов отслеживает переносы для правильного разделения и рекомбинации слов. Например, для сдвига влево алгоритм начинает со старших слов и переносит их младшие биты на младшие слова. Время выполнения  $O(n/B)$ :

```

Bitset& operator>>=(int shift)
{ // сдвиг на 0
  if(shift < 0) return operator<<=(-shift);
  int normalShift = shift % bitSize, wordShift = normalShift/B,
      bitShift = normalShift % B;
  if(wordShift > 0) // сдвиг слов
    for(int i = 0; i + wordShift < wordSize(); ++i)
    {
      storage[i] = storage[i + wordShift];
      storage[i + wordShift] = 0;
    }
  if(bitShift > 0) // сдвиг битов
  { // для раскладки слова 00000101|00111000 >>= 4 ->
    // 10000000|00000011
    WORD carry = 0;
    for(int i = wordSize() - 1 - wordShift; i >= 0; --i)
    {
      WORD tempCarry = storage[i] << (B - bitShift);
      storage[i] >>= bitShift;
      storage[i] |= carry;
      carry = tempCarry;
    }
  }
  return *this;
}

Bitset& operator<<=(int shift)
{
  if(shift < 0) return operator>>=(-shift);
  int normalShift = shift % bitSize, wordShift = normalShift/B,
      bitShift = normalShift % B;
  if(wordShift > 0) // сдвиг слов
    for(int i = wordSize() - 1; i - wordShift >= 0; --i)
    {
      storage[i] = storage[i - wordShift];
      storage[i - wordShift] = 0;
    }
  if(bitShift > 0) // сдвиг битов
  { // для раскладки слов 10000000|00000011 <<= 4 ->
    // 00000000|00111000
    WORD carry = 0;
    for(int i = wordShift; i < wordSize(); ++i)
    {
      WORD tempCarry = storage[i] >> (B - bitShift);
      storage[i] <<= bitShift;
      storage[i] |= carry;
      carry = tempCarry;
    }
  }
  // некоторые равные единице биты могли сдвинуться в остаток
  zeroOutRemainder();
  return *this;
}

```

Чтобы проверить, равны ли все биты 0, можно работать с целыми словами:

```
void setAll(bool value = true)
{
    for(int i = 0; i < wordSize(); ++i)
        storage[i] = value ? Bits::FULL : Bits::ZERO;
    zeroOutRemainder();
}

bool isZero() const
{
    for(int i = 0; i < wordSize(); ++i) if(storage[i]) return false;
    return true;
}
```

Чтобы манипулировать битовой последовательностью размера  $n$  в позиции  $i$ , нужно найти затронутые слова и работать с их затронутыми битами. Если последовательность охватывает несколько слов, затронутые биты будут левыми для первого слова, полностью охватывать средние слова и правыми для последнего слова. Например, пусть есть макет слова `uuuuxxxx|uuuuuuuu|xxxxxuu`, тогда `u` — затронутые биты. Для получения начального бита в первом слове вызовите функцию  $\min(n, B - \text{bit})$  и поместите результат из  $\text{shift} = 0$ . Для более поздних слов задается  $\text{bit} = 0$ , а  $\text{shift}$  увеличивается на количество прочитанных битов. То же самое делается для наборов битов. Сложность становится равна  $O(\text{количество затронутых слов})$ :

```
unsigned long long getValue(int i, int n) const
{
    assert(n <= numeric_limits<unsigned long long>::digits && i >= 0 &&
           i + n <= bitSize && n > 0);
    unsigned long long result = 0;
    for(int word = i/B, bit = i % B, shift = 0; n > 0; bit = 0)
    { // получение сначала младших битов
        int m = min(n, B - bit); // либо все биты, либо столько,
                                // сколько поместилось в слове
        result |= Bits::getValue(storage[word++], bit, m) << shift;
        shift += m;
        n -= m;
    }
    return result;
}

void setValue(unsigned long long value, int i, int n)
{
    assert(n <= numeric_limits<unsigned long long>::digits && i >= 0 &&
           i + n <= bitSize && n > 0);
    for(int word = i/B, bit = i % B, shift = 0; n > 0; bit = 0)
    { // задаются сначала младшие биты
        int m = min(n, B - bit); // либо все биты, либо столько,
                                // сколько поместилось в слове
        Bits::setValue(storage[word++], value >> shift, bit, m);
        shift += m;
        n -= m;
    }
}
```

Операции добавления реализуются функцией `setValue` путем увеличения емкости, если это необходимо, и установки последних битов в желаемое значение:

```
void appendValue(unsigned long long value, int n)
{
    int start = bitSize;
    bitSize += n;
    int k = wordsNeeded() - wordSize();
    for(int i = 0; i < k; ++i) storage.append(0);
    setValue(value, start, n);
}
```

Также полезно иметь возможность добавлять в битовый набор напрямую:

```
void appendBitset(Bitset const& rhs)
{ // добавление слов в storage и удаление лишних слов, если они есть
    if(rhs.getSize() > 0)
    {
        for(int i = 0; i < rhs.wordSize(); ++i)
            appendValue(rhs.storage[i], B);
        bitSize -= B - rhs.lastWordBits();
        if(wordSize() > wordsNeeded()) storage.removeLast();
    }
}
```

При реверсировании используются операции со словами, которые обычно более эффективны, чем векторный обмен битами. Сначала сдвиньте, чтобы скрыть мусорные биты, а затем переверните слова хранения. Нули из сдвига заканчиваются как новые биты мусора:

```
void reverse()
{ // наполнение мусорных битов
    int nFill = garbageBits();
    bitSize += nFill;
    (*this) <<= (nFill);
    // инвертирование слов на хранении
    storage.reverse();
    for(int i = 0; i < wordSize(); ++i)
        storage[i] = reverseBits(storage[i]);
    // удаление мусора
    bitSize -= nFill;
    zeroOutRemainder();
}
```

Количество выданных элементов равно сумме их количеств для отдельных слов:

```
int popCount() const
{
    int sum = 0;
    for(int i = 0; i < wordSize(); ++i) sum += popCountWord(storage[i]);
    return sum;
}
```

## 5.12. Поиск объединения

Попробуем решить *проблему непересекающихся множеств* — о поддержании отношения эквивалентности на подмножествах элементов. Поддерживаемые операции:

- ◆ добавить подмножество;
- ◆ *объединить* два подмножества;
- ◆ проверить, находятся ли два элемента в одном и том же подмножестве;
- ◆ получить размер подмножества, содержащего элемент.

Индексы массива представляют все элементы. Структура данных — это вектор целых чисел, изначально равных  $-1$ , где вектор  $[i]$  соответствует узлу  $i$ . Целые числа обозначают отрицательные размеры подмножества для корневых узлов и родительских индексов в противном случае. Два элемента находятся в одном и том же подмножестве, если их корневые индексы равны.

Чтобы соединить два корня (рис. 5.18):

1. Сделайте меньшее подмножество дочерним для большего.
2. Измените его корень на корень другого.
3. Установите корень равным общему размеру.

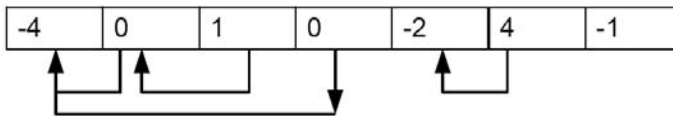


Рис. 5.18. Структура указателя поиска объединения (union-find) с объединенными элементами

Поиск корня следует по цепочке родительских указателей на него, сжимает путь, устанавливая родителя каждого посещаемого узла в корень, и возвращает его индекс. Получение размера подмножества возвращает значение в `findRoot`:

```
class UnionFind
{
    mutable Vector<int> parent; // родительский или отрицательный размер дерева
public:
    UnionFind(int size): parent(size, -1){}
    bool isRoot(int n) const { return parent[n] < 0; }
    int find(int n) const
    { return isRoot(n) ? n : (parent[n] = find(parent[n])); }
    void join(int i, int j)
    {
        int parentI = find(i), parentJ = find(j);
        if (parentI != parentJ)
        { // parent[parentI] и parent[parentJ] имеют
          // отрицательные размеры
            if (parent[parentI] > parent[parentJ]) swap(parentI, parentJ);
            parent[parentI] += parent[parentJ];
            parent[parentJ] = parentI;
        }
    }
}
```

```
bool areEquivalent(int i, int j) const { return find(i) == find(j); }  
int subsetSize(int i) const { return -parent[find(i)]; }  
void addSubset() { parent.append(-1); }  
};
```

Поиск и объединение выполняются за  $O(\lg(n))$  и амортизируются к  $O(1)$  для любых практически применимых значений  $n$ . Эта реализация называется *объединением по размеру со сжатием*.

## 5.13. Примечания по реализации

Моя реализация вектора существенно отличается по API от STL. Например, я включил в нее общие функции работы с векторами, что, возможно, будет лишним, но мне кажется полезным.

Идея реализации в виде свободного списка тоже оригинальна и послужила источником вдохновения для всей книги. Я был в восторге от идеи структуры данных, основанной на последовательности блоков удвоенного размера, и несколько лет спустя взял эту стратегию на вооружение (см. главу 12. *Разные алгоритмы и методы*). Я рассматривал одинаковые структуры для реализаций стека и очереди, но сокращенные вызовы диспетчера памяти не оправдывают неуклюжести такого подхода. Функция сборки мусора позволила мне не писать деструкторы для многих структур на основе указателей.

Битовые алгоритмы было выбрать труднее всего, потому что обычно для решения простых задач имеется много разных способов. К таким задачам относятся, например, подсчет всплывающих окон или обращение битов. Представленные здесь алгоритмы на основе таблиц помещают таблицы в статическую память, чего лучше избегать, поскольку могут возникать промахи в кеше, но более умные алгоритмы грубой силы работают хуже. Другой вариант — вычислять таблицы вместо их жесткого кодирования, как это делается в других источниках.

Для набора битов нужно решать, какую функциональность грубой силы заменить операциями, работающими с целыми словами. В качестве примера возьмем операцию инвертирования: перестановка битов грубой силой при небольшом размере слова, вероятно, выполняется быстрее.

Для поиска объединения размер подмножества обычно не вычисляется, но я включил эту операцию из-за ее полезности и простоты реализации.

## 5.14. Комментарии

Насколько мне известно, свободный список со сборкой мусора никто не реализовывал. Различные интеллектуальные указатели C++ использовать удобнее, но они влекут за собой дополнительные накладные расходы и не повышают эффективность использования памяти.

Что касается подсчета выданных элементов, следует помнить, что на некоторых аппаратных системах, если таблица не находится в кеше, сдвиг и подсчет могут выполняться быстрее, несмотря на использование гораздо большего количества инструкций. Но мои тесты показали, что табличная версия на порядок быстрее, чем версия грубой

силы. Альтернативные переносимые алгоритмы с использованием магических чисел здесь не рассматриваются. Об этих и многих других битовых алгоритмах рассказано в [5.1] и [5.4].

Для поиска по объединению классический альтернативный вариант *объединения по высоте со сжатием* дает ту же асимптотическую и практическую производительность, но тратит больше памяти, т. к. нужно хранить значения высот. Существует экспериментальный чуть более быстрый алгоритм (см. [5.3]), но он сложнее, асимптотически медленнее и не позволяет проверять размеры подмножества.

## 5.15. Советы по дополнительной подготовке

- ◆ Есть ли смысл заводить вспомогательную функцию для возведения в квадрат? Встроенная функция `pow` очень медленная из-за работы с аргументами с плавающей точкой, и ей нужно много битов точности (см. главу 23. *Численные алгоритмы: работа с функциями*). Даже стандарт для операций с плавающей запятой рекомендует, чтобы в языковых библиотеках была готовая функция `pow` для целочисленных показателей степени (см. главу 17. *Большие числа*, чтобы узнать, как эффективно ее реализовать). Но более простой и эффективный вариант для очень маленьких показателей, таких как 2 или 3, — это использовать шаблонное метапрограммирование, расширяя операцию умножения. Напишите такую функцию. Преимущество ее в том, что вам не придется создавать временные переменные, как это делается обычно.
- ◆ Имеет ли смысл для вектора добавлять функции для вставки/удаления элемента не в конец? Такую функцию можно реализовать поверх минимальной вычислительной базы. То же самое приемлемо для добавления массива или вектора элементов вместо одного.
- ◆ Сделайте функции подсчета выданных значений и инвертирования битов переносимыми — т. е. уберите предположение о 8 битах в байте.
- ◆ Для функции инвертирования битов разберитесь, насколько эффективны алгоритмы грубой силы, и как их преимущество зависит от размера слов.

## 5.16. Список рекомендуемой литературы

- 5.1. Arndt J. (2010). *Matters Computational: Ideas, Algorithms, Source Code*. Springer.
- 5.2. Joannou S., & Raman R. (2011). An empirical evaluation of extendible arrays. In *Experimental Algorithms* (pp. 447–458). Springer.
- 5.3. Patwary M. M. A., Blair J., & Manne F. (2010). Experiments on union-find algorithms for the disjoint-set data structure. In *Experimental Algorithms* (pp. 411–423). Springer.
- 5.4. Warren H. S. (2012). *Hacker's Delight*. Addison-Wesley.



## 6. Генерация случайных чисел

Лотерея — это налог на незнание математики.  
*Амброуз Бирс*

Любой, кто изобретает арифметические методы генерации случайных чисел, несомненно, грешник.  
*Джон фон Нейман*

### 6.1. Введение

Рандомизация — это важная часть алгоритмического инструментария, но на занятиях по алгоритмам о ней говорят редко. Для изучения этой главы вам понадобятся некоторые знания теории вероятностей. Эта глава более сложная, чем следующие, вплоть до главы 11. *Алгоритмы графов*, которой мы завершим обзор стандартных тем, касающихся алгоритмов.

### 6.2. Краткий обзор теории вероятностей

*Вероятность* — это функция, измеряющая неопределенность множеств  $E_i$  событий, являющихся подмножествами некоторого *выборочного пространства*  $\Omega$ . Обычно  $\Omega$  — это действительные или целые числа, а множества — непрерывные или дискретные диапазоны. Это положение не распространяется на многомерные  $\Omega$ .  $E_i$  может составляться с помощью некоторых операций над множествами — таких как пересечения, объединения и дополнения. Вероятность удовлетворяет *аксиомам Колмогорова*:

- ♦  $P(\Omega) = 1$ ;
- ♦ для непересекающихся  $E_i$  —  $P(\cup E_i) = \sum P(E_i)$ ;
- ♦  $P(E^c) = 1 - P(E)$ .

*Распределение вероятности* с помощью *функции плотности вероятности*, ФПВ (Probability Density Function, PDF),  $f$  присваивает значения  $x \in \Omega$  таким образом, что:

$$\forall E \Pr(E) = \int_{x \in E} f(x).$$

Чаше бывает удобно работать с *совокупной функцией распределения*, СФР (Cumulative Distribution Function, CDF):

$$F(x) = \int_{-\infty < t \leq x} f(t).$$

С помощью этих функций можно воспринимать диапазоны как элементарные события и вычислять вероятности более сложных событий, используя приведенные ранее аксиомы.

Часто применяемые распределения:

- ♦ *нормальное* ( $\mu, \sigma$ ), где  $\mu$  — среднее значение,  $\sigma$  — стандартное отклонение (рис. 6.1). Оно моделирует многие события и весьма часто используется;
- ♦ *геометрическое* ( $p$ ) —  $f(x) = (1 - p)^{x-1} p$  — для дискретного  $x$  (рис. 6.2).

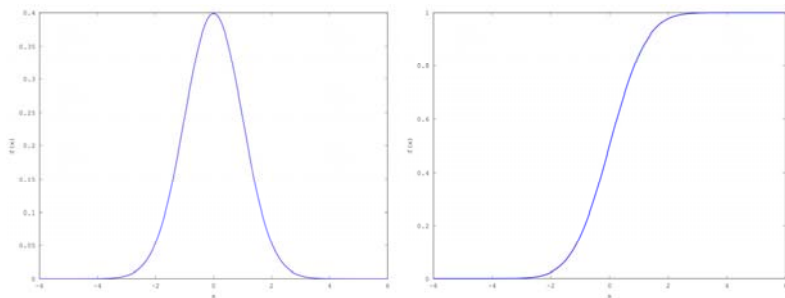


Рис. 6.1. Нормальное (0, 1): ФПР (слева) и СФР (справа)

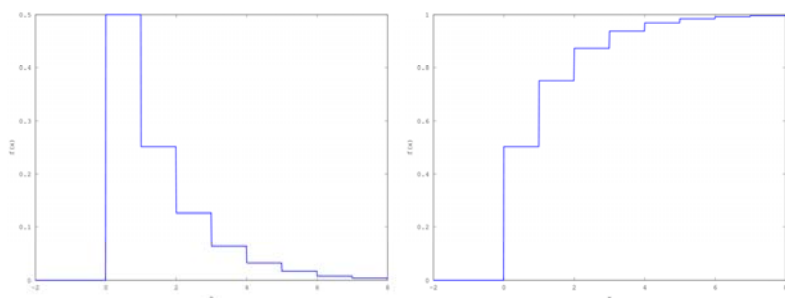


Рис. 6.2. Геометрическое (0,5): ФПР (слева) и СФР (справа)

Некоторые события в природе не имеют известных вероятностей, поэтому их моделируют приблизительно, используя распределения. Для различных случаев определено множество различных распределений, и они хорошо работают в решении задач, но иногда предположения о распределении на практике не подтверждаются. Например, для непрерывного распределения  $Pr(\text{конкретный } x) = 0$ , но если  $x = 0$  или другому логическому значению, непрерывное распределение технически становится неправильной моделью, поскольку  $x = 0$  может встречаться достаточно часто. В результате приходится исправлять многие статистические процедуры, такие как обработка противоречащих предположениям связей в непараметрических тестах (см. главу 21. *Вычислительная статистика*).

Многие события следуют нормальному распределению из-за статистической механики. Например, случайные взаимодействия многих частиц, согласно CLT (см. далее в этой главе), стремятся к нормальному распределению. Бывают явления, которые считаются нормальными, хотя на самом деле таковыми не являются. Каждое распределение хорошо моделирует одни события и плохо — другие. Например, нормальное распределение не подходит для моделирования полета птицы, которая огибает гору, потому что птица будет отклоняться влево или вправо, создавая распределение с двумя пиками (рис. 6.3).



Рис. 6.3. Умные птицы обычно не летят на препятствия

Мультимодальные или асимметричные распределения часто иллюстрируют некорректность допущений конкретных статистических процедур. Во многих случаях для анализа данных нужно не полное распределение вероятности, а *математическое ожидание*:

$$\mu = \int_{x \in S} xf(x)$$

или *дисперсия*:

$$\sigma^2 = E[(x - \mu)^2].$$

### 6.3. Генерация псевдослучайных чисел

В играх, рандомизированных алгоритмах и симуляциях используют случайные числа, из которых можно создавать более сложные случайные объекты. Числовая последовательность является *случайной*, если наблюдение за ее прошлыми значениями не помогает предсказать следующее значение (рис. 6.4).



Рис. 6.4. Орел или решка?

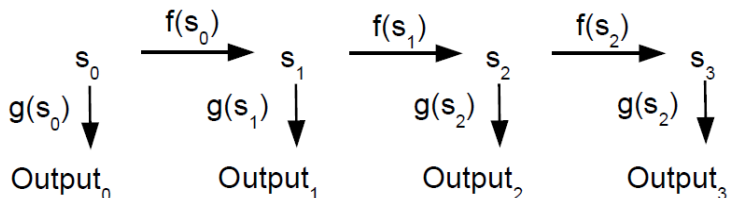


Рис. 6.5. Типовые переходы состояний генератора псевдослучайных чисел и выходные данные

*Равномерно распределенные физические явления* — такие как атмосферный шум или радиоактивный распад — генерируют случайные числа, и их можно использовать в лотереях и в отборе случайных присяжных для суда. Пример генератора случайных чисел приведен на сайте **www.random.org**. Существуют специальные устройства-генераторы, а операционные системы генерируют случайные числа на основе активности ЦП или жесткого диска. Они используются для криптографических алгоритмов, но медленны и непереносимы.

Быстрые *генераторы псевдослучайных чисел* генерируют случайную последовательность, начиная с некоторого *начального состояния* (рис. 6.5). Выходное значение и следующее состояние являются соответственно *выходной* ( $g$ ) и *переходной* ( $f$ ) функциями текущего состояния. Обычно на эти операции требуется  $O(1)$  времени и  $O(1)$  пространства.

Получается ли результат достаточно случайным, даже если в начальном состоянии есть много случайных битов? Технически нет, потому что комбинация аргументов делает не

все выходные последовательности возможными. Но почти всегда предполагается, что псевдослучайность для того или иного приложения достаточно случайна. Попытка как-то преднамеренно использовать механику конкретного генератора, как правило, будет успешной только в случае, если такая возможность заранее предусмотрена. На практике информационный поток генератора и домена независимы.

Для простоты и переносимости начальное состояние может быть функцией системного времени и пароля. Можно использовать и другие, недоступные для переноса источники, если нужно больше случайных битов. Для полной независимости запуска генератора можно сохранять его состояние в файл и восстанавливать для следующей генерации, но это редко стоит затраченных усилий. Простейшие генераторы используют состояние из одного слова и функцию. Выход обычно состоит из 32 или 64 битов. Вы можете объединять последовательные 32-битные выходные данные для получения 64-битных выходных данных, если необходимо.

Хорошие генераторы:

- ◆ проходят большинство тестов крупных статистических наборов, таких как TestU01 (см. [6.4]), которые пытаются отвергнуть гипотезу (о гипотезах см. в *главе 21. Вычислительная статистика*) о том, что последовательность является случайной и имеет *период* (число переходов, после которых состояние возвращается к начальному)  $\geq 264$  и высокое *равнораспределение* (максимальное  $k$ , для которого последовательность из  $k$  выходов может быть любой последовательностью). Так как тестирование многократное (см. *главу 21. Вычислительная статистика*), проходить все тесты необязательно;
- ◆ эффективны, просты, переносимы и легко инициализируются;
- ◆ возвращают случайное число типа `double`  $u \in (0, 1)$ , а не  $[0, 1]$ , чтобы избежать неопределенных результатов для таких преобразований, как  $\log(u)$ . Ни один генератор не выдает  $2^n$ , поэтому константа нормализации  $\geq 2^n$  будет соответствовать единице. Константы двойной нормализации для 32- и 64-битных значений полного диапазона — это  $2,32830643653869629E-10$  и  $5,42101086242752217E-20$  (см. [6.8]). Некоторые генераторы также не выдают значение 0. В этом случае используйте функцию  $\max(1, u)$  или генерируйте до тех пор, пока  $u \neq 0$ . В первом случае результат имеет смещение, а во втором нет границ худшего случая, но на практике и то и другое не страшно.

При желании создавайте *независимые потоки* для параллельных вычислений. Для этого нужно большое значение периода и возможность эффективно пропускать шаги.

Исторически популярный *линейный конгруэнтный генератор* (linear congruential generator, LCG) с состоянием из одного слова  $s$  использует переход  $s = (as + c) \% m$  (например, при  $a = 69069$ ,  $c = 1234567$  и  $m = 2^{32}$ ). Но у него плохое качество:

- ◆ период младших  $k$  битов  $= 2^k$ ;
- ◆ он не проходит многие тесты;
- ◆ чтобы избежать переполнения, умножение требует точности двойного слова или разбивается на части.

Функция C++ `rand()`, скорее всего, основана на этом или другом низкачественном генераторе, поэтому лучше ее не использовать.

## 6.4. Класс генераторов псевдослучайных чисел Xorshift

*Переход* умножает битовый вектор  $r$ , представленный *состоянием* слова, на последовательность *сдвигов* (shift) и *исключающих операций* (xor). Используемая здесь последовательность проходит тесты с хорошими результатами. Математическое обоснование свойств Xorshift довольно сложно (см. [6.8]). Для 64-битного состояния период равен  $2^{64} - 1$  (0 никогда не генерируется), что достаточно много. Далее приведена 32-битная функция перехода с периодом  $2^{32} - 1$  и различными коэффициентами, полезная для реализации универсального хеширования (см. главу 9. Хеширование):

```
uint32_t xorshiftTransform(uint32_t x)
{
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    return x;
}
```

Мы рассмотрим только 64-разрядную версию псевдослучайной генерации из-за превосходных результатов теста (см. далее в этой главе). Тем не менее биты последовательно идущих друг за другом чисел имеют некоторую корреляцию из-за линейности матричного умножения. Таким образом, QualityXorshift64 выводит следующую генерацию LCG результата Xorshift:

```
class QualityXorshift64
{
    uint64_t state;
    enum{PASSWORD = 19870804};
public:
    QualityXorshift64(uint64_t seed = time(0) ^ PASSWORD)
        {state = seed ? seed : PASSWORD;}
    static uint64_t transform(uint64_t x)
    {
        x ^= x << 21;
        x ^= x >> 35;
        x ^= x << 4;
        return x * 2685821657736338717ull;
    }
    uint64_t next(){return state = transform(state);}
    unsigned long long maxNextValue(){return numeric_limits<uint64_t>::max();}
    double uniform01(){return 5.42101086242752217E-20 * next();}
};
```

Алгоритм не генерирует 0, потому что константа умножения — простое число, а для простого числа  $p$ ,  $0 = xp \% 2^{64} \rightarrow xp = c2^{64}$ . Поскольку  $c > p$ ,  $c$  делится на  $p$ , что подразумевает противоречие с тем, что  $x = (p/c) \cdot 2^{64}$ , а  $x$  меньше.

Переходы Xorshift являются однозначными, а переходы LCG — нет.

## 6.5. Вихрь Мерсенна

*Вихрь Мерсенна* — это популярный и быстрый алгоритм с хорошим качеством как в 32-битном, так и в 64-битном варианте. В книге [6.7] представлен краткий алгоритм, а в статье <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html> приведены сложные реализации на C, в которых, впрочем, присутствуют новые параметры и процедуры инициализации. В 64-битной версии выполняется дополнительная операция & с константой, но в остальном она отличается только параметрами. Генератор работает так же быстро, как *QualityXorshift64*, но намного сложнее, и использует в качестве состояния массив из 623 целых чисел.

Я упростил код и убедился, что эта версия на моей машине работает так же быстро:

```
class MersenneTwister64
{
    enum{N = 312, PASSWORD = 19870804};
    uint64_t state[N];
    int i;
public:
    MersenneTwister64(uint64_t seed = time(0) ^ PASSWORD)
    {
        state[0] = seed;
        for(i = 1; i < N; ++i) state[i] = 6364136223846793005ULL *
            (state[i - 1] ^ (state[i - 1] >> 62)) + i;
        i = 0;
    }
    uint64_t next()
    {
        int j = (i + 1) % N, k = (i + 126) % N;
        uint64_t y = (state[i] & 0xFFFFFFFF80000000ULL) |
            (state[j] & 0x7fffffffULL);
        y = state[i] =
            state[k] ^ (y >> 1) ^ (y & 1 ? 0xB5026F5AA96619E9ULL : 0);
        i = j;
        y ^= (y >> 29) & 0x5555555555555555ULL;
        y ^= (y << 17) & 0x71D67FFFE6A60000ULL;
        y ^= (y << 37) & 0xFFF7EEE000000000ULL;
        y ^= (y >> 43);
        return y;
    }
    double uniform01(){return 5.42101086242752217E-20 * next();}
};
```

## 6.6. Генератор MRG32k3a

*Множественный рекурсивный генератор* (multiple recursive generator, MRG) — это обобщение LCG, где задействована линейная комбинация нескольких последних состояний по модулю  $m$ . MRG32k3a объединяет выходные данные двух генераторов с тремя состояниями, которые используют переход  $s_{i,next} = c_i + s_i T_i \pmod{m_i}$  с соответствующим образом выбранными матрицами перехода  $T_i$  и модулями  $m_i$ . Оба параметра

были выбраны в результате исчерпывающего поиска, а предлагаемые значения  $T_i$  невелики из соображений эффективности (см. [6.3]). Выход представляет собой комбинацию  $c_i$ :

```
struct MRG32k3a
{
    enum{PASSWORD = 19870804};
    static long long const m1 = 429496708711, m2 = 429494444311;
    long long s10, s11, s12, s20, s21, s22;
    void reduceAndUpdate(long long c1, long long c2)
    { // вспомогательная функция для выполнения перехода
        if(c1 < 0) c1 = m1 - (-c1 % m1);
        else c1 %= m1;
        if(c2 < 0) c2 = m2 - (-c2 % m2);
        else c2 %= m2;
        s10 = s11; s11 = s12; s12 = c1;
        s20 = s21; s21 = s22; s22 = c2;
    }
public:
    unsigned long long next()
    { // не возвращает 0
        long long c1 = (1403580 * s11 - 810728 * s10),
            c2 = (527612 * s22 - 1370589 * s20);
        reduceAndUpdate(c1, c2);
        return (c1 <= c2 ? m1 : 0) + c1 - c2;
    }
    unsigned long long maxNextValue(){return m1;}
    // s1(0-2) и s2(0-2) должны быть < m1 и m2 соответственно,
    // но не все равными 0
    MRG32k3a(): s10(max(time(0) ^ PASSWORD, 11) % m2), s11(0), s12(0),
        s20(s10), s21(0), s22(0){}
    double uniform01(){return next()/(m1 + 1.0);} // гарантирует нахождение u в (0, 1)
};
```

Можно прыгнуть вперед, используя эффективное возведение в степень по модулю  $m$  (см. главу 17. Большие числа), на  $T_i$ . Рекомендуемый размер прыжка —  $2^{76}$ , а результирующие матрицы уже рассчитаны (см. [6.5]):

```
void jumpAhead()
{
    const long long A1p76[3][3] = {
        { 82758667u, 1871391091u, 4127413238u}, // для s10
        {3672831523u, 69195019u, 1871391091u}, // для s11
        {3672091415u, 3528743235u, 69195019u}}, // для s12
        A2p76[3][3] = {
            {1511326704u, 3759209742u, 1610795712u}, // для s20
            {4292754251u, 1511326704u, 3889917532u}, // для s21
            {3859662829u, 4292754251u, 3708466080u}}; // для s22
    long long s1[3] = {s10, s11, s12}, s2[3] = {s20, s21, s22};
    for(int i = 0; i < 3; ++i)
    {
        long long c1 = 0, c2 = 0;
```

```

    for(int j = 0; j < 3; ++j)
    {
        c1 += s1[j] * A1p76[i][j];
        c2 += s2[j] * A2p76[i][j];
    }
    reduceAndUpdate(c1, c2);
}
}

```

Незначительная проблема этого генератора заключается в том, что диапазон выходных данных не является полным 64-битным, а ограничен  $m_1$ . Его 64-битная версия не разработана. Так, например:

- ◆ число  $u$ , которое имеет точность  $\approx 16$  цифр, получает только  $\approx 12$ ;
- ◆ все, что зависит от состояния переменной (или  $u$ ), будет слишком ограничено по диапазону ( $u$  не может быть слишком близко к 0 из-за ограничения точности).

## 6.7. Алгоритм RC4

*Криптографически безопасный случайный вывод* нельзя предсказать каким-либо эффективным методом, результат которого был бы значительно лучше простого угадывания. RC4 безопасно генерирует байты из достаточно случайного начального ключа. Чтобы избежать некоторых известных атак, он отбрасывает первые 1024 байта. Собственно «RC4» представляет собой товарный знак, и реализации называют его «ARC4» (где «А» означает «предполагаемый»), но, скорее всего, этот товарный знак уже не защищается. Логика алгоритма сложна, а его безопасность до сих пор недостаточно изучена (подробности см. в [6.10]). Алгоритм пытается генерировать и поддерживать что-то вроде случайной перестановки:

```

struct ARC4
{
    unsigned char sBox[256], i, j;
    enum{PASSWORD = 19870804};
    void construct(unsigned char* seed, int length)
    {
        j = 0;
        for(int k = 0; k < 256; ++k) sBox[k] = k;
        for(int k = 0; k < 256; ++k)
        { // отличается от алгоритма случайной перестановки
            j += sBox[k] + seed[k % length];
            swap(sBox[k], sBox[j]);
        }
        i = j = 0;
        for(int dropN = 1024; dropN > 0; dropN--) nextByte();
    }
    ARC4(unsigned long long seed = time(0) ^ PASSWORD)
        {construct((unsigned char*)&seed, sizeof(seed));}
    // для криптографической инициализации из длинного посева
    ARC4(unsigned char* seed, int length){construct(seed, length);}
    unsigned char nextByte()

```



```

{
    j += sBox[++i];
    swap(sBox[i], sBox[j]);
    return sBox[(sBox[i] + sBox[j]) % 256];
}
unsigned long long next()
{
    unsigned long long result = 0;
    for(int k = 0; k < sizeof(result); ++k)
        result |= ((unsigned long long)nextByte()) << (8 * k);
    return result;
}
unsigned long long maxNextValue()
{return numeric_limits<unsigned long long>::max();}
double uniform01(){return 5.42101086242752217E-20 * max(lull, next());}
};

```

Используемые время и пространство по-прежнему равны  $O(1)$ , т. к.  $2^{56}$  является константой.

## 6.8. Выбор генератора

Для удобства рассмотрения свойства описанных в предыдущих разделах генераторов сведены в табл. 6.1.

**Таблица 6.1.** Свойства генераторов псевдослучайных чисел

Генератор	Период и затраты памяти на хранение состояния	Количество непройденных тестов из TestU01	Время выполнения $2^{30}$ вызовов в сек.	Примечания
Xorshift	$2^{32} - 1$ , 4 байта	> 65	3	Простейшая, самая быстрая, хороша как 32-битная хеш-функция.
Xorshift64	$2^{64} - 1$ , 8 байтов	16, с разными сдвигами; 59 для 32-битной версии	6,3	Полезна как строительный блок и как хеш-функция
QualityXorshift64	$2^{64} - 1$ , 8 байтов	Н/Д; 0 для меньшего набора тестов Diehard	6,6	Простая, хороша как 64-битная хеш-функция
MGR32k3a	$\approx 2^{182}$ , 24 байта	0	25,6	Поддерживает пропуск
Вихрь Мерсенна 64	$2^{19937} - 1$ , 2504 байта	Н/Д (4 для 32-битной версии)	9,7	Популярная, но с трудом выходит из состояний со многими 0
RC4	Огромный, 256 байтов	0	47,8	Криптографическая функция

Выполнение симуляций нацелено на то, чтобы не потерять значимость из-за неслучайности, а рандомизированные алгоритмы должны быть эффективными в целом. Все генераторы достаточно быстрые, чтобы не быть узким местом в приложении. Так что используйте:

- ◆ QualityXorshift64 (мой выбор) или вихрь Мерсенна (популярный выбор) по умолчанию;
- ◆ MGR32k3a — если нужна поддержка независимого потока;
- ◆ RC4 — для задач криптографии.

## 6.9. Использование генератора

Здесь генерацию выполняет глобальный объект. Он также генерирует числа  $\text{mod } n$  и  $\in [a, b]$ , и большинство генераторов случайных объектов и выборок из вероятностных распределений являются его членами:

```
template<typename GENERATOR = QualityXorshift64> struct Random
{
    GENERATOR g;
    enum{PASSWORD = 19870804};
    Random(unsigned long long seed = time(0) ^ PASSWORD): g(seed){}
    unsigned long long next(){return g.next();}
    unsigned long long maxNextValue(){return g.maxNextValue();}
    unsigned long long mod(unsigned long long n)
    {
        assert(n > 0);
        return next() % n;
    }
    int sign(){return mod(2) ? 1 : -1;}
    long long inRange(long long a, long long b)
    {
        assert(b >= a);
        return a + mod(b - a + 1);
    }
    double uniform01(){return g.uniform01();}
};
Random<>& GlobalRNG()
{
    static Random<> r; // запускается лишь раз
    return r;
}
```

## 6.10. Создание выборок из распределений

Большинство алгоритмов генерации предполагают возможность генерации  $u$  — выборки из распределения  $\text{uniform}(0,1)$ . Для непрерывных распределений существует несколько методов:

- ◆ *инверсия* — совокупное распределение вероятностей  $F$  представляет собой функцию, отображающую  $x \rightarrow [0, 1]$ , поэтому  $F^{-1}(u)$  является случайной величиной. На-

пример, далее в этой главе используется генератор экспоненциального распределения. Это хорошо работает, когда  $F^{-1}$  легко вычислить. Численное решение  $F(x) = u$  (см. главу 23. Численные алгоритмы: работа с функциями) тоже работает, но медленно. В нем нужно найти диапазон, содержащий  $x$ , соответствующий  $u$ , используя экспоненциальный поиск, начиная с  $x = 0$ :

```
template<typename CDF> double invertCDF(CDF const& c, double u,
    double guess = 0, double step0 = 1, double prec = 0.0001)
{
    assert(u > 0 && u < 1);
    auto f = [c, u](double x){return u - c(x);};
    pair<double, double> bracket = findInterval0(f, guess, step0, 100);
    return solveFor0(f, bracket.first, bracket.second, prec).first;
}
```

Экспоненциальный поиск гарантированно работает с монотонными функциями, поэтому приведенная здесь реализация надежна. Инверсия также работает для дискретных распределений, но не для многомерных (рис. 6.6);

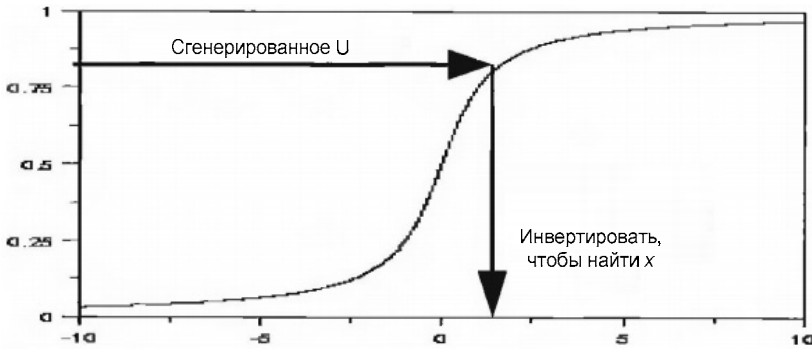


Рис. 6.6. Стратегия метода инверсии

♦ *принятие/отклонение* — если  $x \in A$  равномерно распределен в  $A$ ,  $B \subseteq A$  и  $x \in B$ , то  $x$  равномерно распределен в  $B$ . Пусть  $f, g$  — функции плотности вероятности такие, что для некоторой константы  $c > 0 \forall x \in \mathbb{R}^D f(x) < cg(x)$ . Если  $x$  генерируется из  $g$ ,  $u$  — из  $\text{uniform}(0, cg(x))$  и  $u \leq f(x)$ , то  $x$  распределяется в соответствии с  $f$  (см. [6.1]). Генерируется  $x$  и  $u$  до тех пор, пока  $u \leq f(x)$ , и возвращается  $x$ . Это работает для дискретных и многомерных распределений, но для больших  $D$  неэффективно, потому что  $E$  [количество отклонений] экспоненциально в  $D$ . Требуется внутренний/внешний тестер принадлежности, который может ускорить генерацию, проверяя, находится ли образец внутри *функций сжатия* в  $f$  (рис. 6.7). Так, например, работает гамма-генератор, описанный далее в этой главе. Техническая проблема заключается в том, что число поколений в наихудшем случае  $= \infty$ , хотя экспоненциально маловероятно, что оно будет больше, чем ожидалось.

Отношения между распределениями: можно использовать несколько распределений для создания связанных распределений. Например, генераторы для бета и многих других сводятся к генераторам для гаммы;

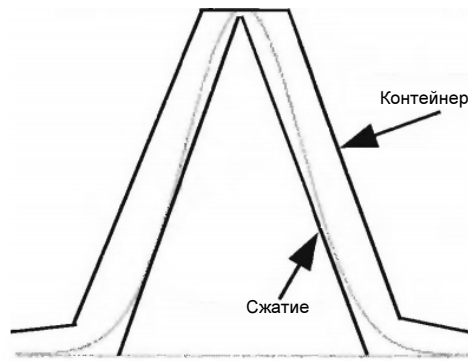


Рис. 6.7. Стратегия метода принятия/отклонения

- ♦ *метод композиции* — вы можете разделить распределение, определенное на  $(a, b)$ , на два распределения, определенных на диапазонах  $(a, c)$  и  $[c, b)$ , сгенерировать переменную выбора, которая определяет, какой из двух используется, и применить затем то или иное распределение. Метод удобен при наличии параметров распределения (где не нужно выбирать область) — например, гамма-генератор использует разные стратегии в зависимости от того, отвечает ли параметр формы условию « $< 1$ ».

## 6.11. Генерация выборок из дискретных распределений с ограниченным диапазоном

Предположим, что желаемое распределение представлено в виде массива  $p$  из  $n$  вероятностей. Если нужно сгенерировать только значения  $O(1)$ , проще всего сначала выполнить генерацию, а затем линейный поиск, чтобы найти первый индекс  $i$  такой, что  $\sum p_i \geq u$ . Но даже в этом случае лучше использовать приведенный далее метод из API.

*Метод-псевдоним* использует память  $O(n)$  и поддерживает генерацию за  $O(1)$ . В случае равновероятности значений при генерации вероятность каждого значения равна  $1/n$ . Некоторые значения будут иметь большую вероятность (*богатые*), а другие — меньшую (*бедные*), но в среднем вероятность каждого равна  $1/n$ . Для усреднения можно сопоставить каждое богатое значение с бедным. Богатый может стать бедным, если отдаст слишком много, и тогда окажется в паре с другим богатым. Для упрощения вместо вероятностей используйте параметр богатства ( $wealth_i$ ), равный  $np_i$ . После завершения объединения для любого  $i$  будет верно равенство:

$$1 = wealth_i + \text{чистая сумма взятого-и-данного.}$$

*Псевдоним* здесь — это индекс значения, которое отдает.

Порядок действий:

1. Сгенерировать равномерное случайное  $i$ .
2. Если у  $i$  нет донора, вернуть  $i$ .
3. В противном случае сгенерировать равномерное распределение  $(0, 1/n)$  и вернуть  $i$  с вероятностью, пропорциональной его первоначальному богатству, иначе — его донора.

Учитываются все вероятности. Алгоритм сопоставления работает жадно:

1. Поместить каждый индекс  $i$  в список бедных, если  $\text{wealth}_i < 1$ , и в список богатых в противном случае.
2. Пока любой из списков не станет пустым:
3.       Выбрать бедного и богатого, взять у богатого  $1 - \text{wealth}_{\text{бедного}}$ .
4.       Переместить богатого в список бедных, если он стал бедным.

Чтобы запомнить богатство, которое было у бедного изначально, не обновляйте его значение. Суммарный размер списка уменьшается, даже если богатый станет бедным, и время выполнения будет равно  $O(n)$ . Можно использовать любой тип списка, но проще всего задействовать стек. Оба списка должны стать пустыми одновременно. Ошибки округления иногда немного влияют на результаты, но незначительно. Поэтому выполняем сопоставление, пока не опустеет хотя бы один список, как показано на рис. 6.8 и в следующем коде.

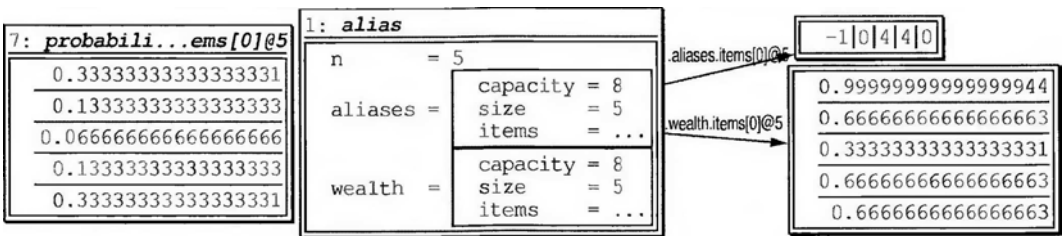


Рис. 6.8. Память метода псевдонима для 5 элементов и неравных вероятностей

```
class AliasMethod
{
    int n;
    Vector<int> aliases; // -1 означает отсутствие донора
    Vector<double> wealth; // исходное богатство или его значение после перераспределения
public:
    AliasMethod(Vector<double> const& probabilities):
        n(probabilities.getSize()), aliases(n, -1), wealth(n, 0)
    {
        Stack<int> smaller, greater;
        for(int i = 0; i < n; ++i)
        { // делим на бедных и богатых
            (wealth[i] = n * probabilities[i]) < 1 ?
                smaller.push(i) : greater.push(i);
        }
        while(!smaller.isEmpty() && !greater.isEmpty())
        { // перераспределяем богатство до тех пор,
            // пока бедные не закончатся
            int rich = greater.getTop(), poor = smaller.pop();
            aliases[poor] = rich;
            wealth[rich] -= 1 - wealth[poor];
            if(wealth[rich] < 1) smaller.push(greater.pop());
        }
    }
}
```

```

int next()
{ // проверка на -1 нужна для обнаружения накопленных
  // ошибок округления
  int x = GlobalRNG().mod(n);
  return aliases[x] == -1 || GlobalRNG().uniform01() < wealth[x] ?
    x : aliases[x];
}
};

```

Куча сумм не использует дополнительное пространство и обеспечивает эффективную генерацию, обновление, поиск и получение совокупной вероятности. Это дерево, в котором значение родителя равно сумме значений его дочерних элементов и самого себя (рис. 6.9).

6 = 1 + 2 + 3, то есть значение корня = 3

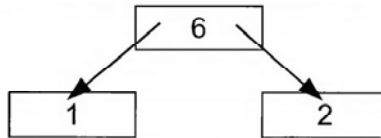


Рис. 6.9. Правила суммирования родительского узла кучи сумм

В задаче распределения вероятностей значение корня равно 1, если распределение не нормализовано, иначе оно равно константе нормализации. Куча содержит значения в диапазоне  $[0, 1]$ , рекурсивно вычисляемые как [левое CDF]  $\cup$  [правое CDF]  $\cup$  [родительское PDF], где значение родительского PDF = родительский – левый – правый. Что касается метода псевдонимов, то узлы индексируются как позиции массива. Большинство операций выполняется за время  $O(\lg(n))$ . Значения устанавливаются с помощью инкрементных обновлений. Обновление заключается в прибавлении суммы приращения к значению узла и каждого его предка. Бинарная куча (см. главу 10. Приоритетные очереди) реализуется через вектор, как показано на рис. 6.10 и в следующем коде.

```

template<typename ITEM> class SumHeap
{
    Vector<ITEM> heap;
    int parent(int i){return (i - 1)/2;}
    int leftChild(int i){return 2 * i + 1;}
    int rightChild(int i){return 2 * i + 2;}
public:
    ITEM total(){return heap[0];}
};

```

Чтобы получить значение узла, вычитите его дочерние значения:

```

ITEM get(int i)
{
    ITEM result = heap[i];
    int c = leftChild(i);
    if(c < heap.getSize())
    {
        result -= heap[c];
        c = rightChild(i);
    }
}

```

```

    if(c < heap.getSize()) result -= heap[c];
}
return result;
}

```

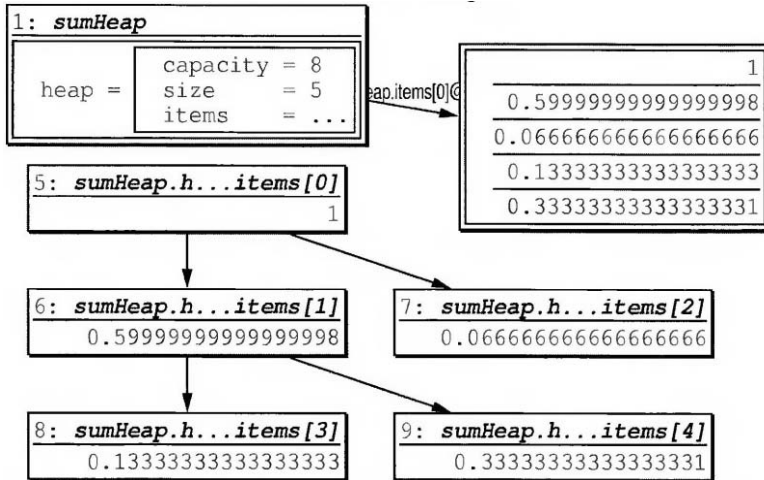


Рис. 6.10. Логика суммирования памяти кучи для одних и тех же данных

При добавлении значения к узлу нужно также добавить его каждому родителю на пути к корню:

```

void add(ITEM value, int i = -1)
{ // -1 означает, что узлов еще нет
    if(i == -1)
    {
        i = heap.getSize();
        heap.append(0);
    }
    for(;; i = parent(i))
    {
        heap[i] += value;
        if(i < 1) break;
    }
}

```

Генерация работает за счет интервальной связи. Для заданного значения в диапазоне от 0 до значения корня нужно найти соответствующий узел путем проверки, где он находится в интервале текущего узла, и либо вернуть его, либо перейти к одному из его дочерних элементов. Дальше левые значения накапливаются, пока вы не получите желаемое значение:

```

int find(ITEM value)
{
    assert(0 <= value && value <= total());
    ITEM totalLeftValue = 0;
    for(int i = 0, c;; i = c)

```

```

{
    c = leftChild(i);
    if(c >= heap.getSize()) return i; // конечный узел
    if(value > totalLeftValue + heap[c])
    {
        totalLeftValue += heap[c];
        c = rightChild(i);
        if(c >= heap.getSize() || // проверяем, есть ли значение в родительском
            value > totalLeftValue + heap[c]) return i;
    }
}
}
int next(){return find(GlobalRNG().uniform01()*total());}

```

CDF узла равно его значению +  $\sum$  значений всех его левых потомков на пути к корню:

```

ITEM cumulative(int i)
{
    ITEM sum = heap[i];
    while(i > 0)
    { // прибавляем значение каждого левого брата, если
        int last = i;
        i = parent(i);
        int l = leftChild(i);
        if(l != last) sum += heap[l];
    }
    return sum;
}

```

## 6.12. Генерация выборок из особых распределений

Некоторые из представленных генераторов просты, а другие математически сложны. Последние более подробно обсуждаются в книге [6.2] или по конкретным ссылкам. Вот некоторые важные непрерывные распределения (пример вывода представленных генераторов приведен на рис. 6.11).

♦ **Uniform( $a, b$ )** — равномерное распределение ( $a, b$ ), где  $a < b$ .

$$f(x) = \begin{cases} 1/(b-a), & a \leq x \leq b \\ 0 & \end{cases}.$$

$$F(x) = 0, \text{ если } x < a, \frac{x-a}{b-a}, \text{ если } a \leq x \leq b \text{ и } 1, \text{ если } x > b.$$

Генератор масштабирует  $u$ :

```
double uniform(double a, double b){return a + (b - a) * uniform01();}
```

♦ **Exponential( $a$ )** — экспоненциальное распределение ( $a$ ): моделирует время между независимыми событиями — например, время между дорожно-транспортными происшествиями для конкретного водителя.

$$f(x) = ae^{-ax} \text{ и } F(x) = 1 - e^{-ax}.$$



Чтобы использовать обратный метод, предположим, что  $F(x) = u$ , и решаем, чтобы получить  $x = -\ln(1 - u)/a$ , что упрощается до  $-\ln(u)/a$ , поскольку  $u$  и  $1 - u$  имеют одинаковое распределение (обратите внимание, что функция `<math>\log</math>` вычисляется по основанию  $e$ ):

```
double exponential(double a){return -log(uniform01())/a;}
```

- ♦ **Cauchy( $\mu$ ,  $\sigma$ )** — распределение Коши ( $\mu$ ,  $\sigma$ ): толстые хвосты и неопределенное математическое ожидание. Последнее связано с тем, что одно наблюдение может составлять  $> 99\%$  дисперсии выборки.

$$f(x) = \frac{1}{\pi + ((x - \mu) / \sigma)^2} \text{ и } F(x) = \frac{1}{\pi} \arctan\left(\frac{x - \mu}{\sigma}\right) + \frac{1}{2}.$$

Генерация использует обратный метод, решая вышеизложенное:

```
double cauchy(double m, double q)
{return (tan((uniform01() - 0.5) * PI()) + m) * q;}
```

- ♦ **Normal( $\mu$ ,  $\sigma$ )** — нормальное распределение ( $\mu$ ,  $\sigma$ ): широко используется и необходимо другим генераторам.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right) \text{ и } F(x) = \frac{1}{2} \left(1 + \operatorname{erf}\left(\frac{x - \mu}{\sigma\sqrt{2}}\right)\right),$$

где *erf* — функция ошибок. Генерация использует *полярный метод*, специально разработанный для нормального распределения (см. [6.8]).  $E[\text{время выполнения}] = O(1)$ :

```
double normal01()
{
    for(;;)
    {
        double a = 2 * uniform01() - 1, b = 2 * uniform01() - 1,
               c = a * a + b * b;
        if(c < 1)
        {
            double temp = sqrt(-2 * log(c)/c);
            return a * temp; // может вернуть b * temp
                           // как второй образец iid
        }
    }
}

double normal(double m, double q){return m + q * normal01();}
```

- ♦ **Gamma(1, b)** — гамма-распределение (1, b): используется другими генераторами. Генератор сложен, но основная его идея состоит в том, чтобы принять/отклонить значение с помощью быстрой полиномиальной функции сжатия (см. [6.6]).  $E[\text{время выполнения}] = O(1)$ :

```
double gammal(double b)
{
    assert(b > 0);
    if(b >= 1)
    {
        for(double third = 1.0/3, d = b - third, x, v, u, xs;;)
```

```

{
    do
    {
        x = normal01();
        v = 1 + x * third/sqrt(d);
    }while(v <= 0);
    v *= v * v; u = uniform01(), xs = x * x;
    if(u > 0.0331 * xs * xs || log(u) < xs/2 +
        d * (1 - v + log(v))) return d * v;
    }
}
else return pow(uniform01(), 1/b) * gammal(b + 1);
}

```

Получено из гаммы с использованием отношений между распределениями:

```

double erlang(double m, int k){return gammal(k) * m/k;}
double chiSquared(int k){return 2 * gammal(k/2.0);}
double t(int v){return sqrt(v/chiSquared(v)) * normal01();}
double beta(double p, double q)
{
    double G1 = gammal(p);
    return G1/(G1 + gammal(q));
}
double F(int v1, int v2)
{return v2 * chiSquared(v1)/(v1 * chiSquared(v2));}

```

- ◆ **Triangular(*middle*)** — треугольное распределение (середина): функция плотности вероятности представляет собой треугольник, образованный точками (0,0), (середина, 1) и (1, 0). Генератор получается путем интегрирования для получения CDF и использования обратного метода [6.11]:

```

double triangular01(double middle)
{
    assert(0 < middle && middle < 1);
    double u = uniform01();
    return sqrt(u <= middle ? middle * u : (1 - middle) * (1 - u));
}
double triangular(double a, double b, double middle)
{return a + (b - a) * triangular01((middle - a)/(b - a));}

```

- ◆ **Levy(*c*)** — распределение Леви (*c*): более толстые хвосты, чем у распределения Коши, асимптотическая сложность  $O(1/x^{1.5})$  (см. [6.12]).

$$f(x) = \sqrt{\frac{c}{2\pi}} \exp\left(-\frac{c}{2x}\right) \frac{1}{x^{1.5}}.$$

Генерация использует тот факт, что если  $N$  — это выборка из  $\text{normal}(0, 1/\sqrt{c})^2$ , тогда  $1/N \sim \text{Леви}(c)$ . Значение  $c = 0,455$  дает медиану  $\approx 1$ . В отличие от Коши, распределение Леви является односторонним, но с ним можно использовать метод композиции для выборки из «симметричного» распределения Леви путем выборки  $-\text{Levy}(c)$  с вероятностью 0,5:

```
double Levy(double scale = 0.455)
{
    double temp = normal(0, 1/sqrt(scale));
    return 1/(temp * temp);
}
double symmetrizedLevy(double scale = 0.455){return sign() * Levy(scale);}
```

Для дискретных распределений использование отношений между распределениями часто дает хорошие генераторы. Вот некоторые из них:

- ◆ **Bernoulli( $p$ )** — Бернулли ( $p$ ): 1 с вероятностью  $p$  и 0 с вероятностью  $1 - p$ :

```
bool bernoulli(double p){return uniform01() <= p;}
```

- ◆ **Binomial( $p, n$ )** — биномиальное ( $p, n$ ): сумма  $n$  от Бернулли ( $p$ ). Время выполнения  $O(n)$ :

```
int binomial(double p, int n)
{
    int result = 0;
    for(int i = 0; i < n; ++i) result += bernoulli(p);
    return result;
}
```

- ◆ **Geometric( $p$ )** — геометрическое ( $p$ ): количество генераций Бернулли, чтобы получить значение 1.  $E[\text{время выполнения}] = O(1/p)$ . При  $p = 0,5$  для эффективности можно использовать случайные целые биты:

```
int geometric(double p)
{
    assert(p > 0);
    int result = 1;
    while(!bernoulli(p)) ++result;
    return result;
}
int geometric05(){return rightmost0Count(next()) + 1;}
```

- ◆ **Poisson( $l$ )** — распределение Пуассона ( $l$ ): количество независимых событий в заданном временном интервале со средним значением  $l$ . Интервалы между событиями ~ экспоненциальные, поэтому необходимо сгенерировать интервалы между событиями и подсчитать, сколько из них вписывается в интервал  $l$ . Чтобы избежать вычисления логарифмов в экспоненциальном генераторе, работайте с экспоненциальными значениями напрямую. Время выполнения  $O(l)$ :

```
int poisson(double l)
{
    assert(l > 0);
    int result = -1;
    for(double p = 1; p > exp(-l); p *= uniform01()) ++result;
    return result;
}
```

```
GlobalRNG().uniform01() 0.69228108650398515
GlobalRNG().uniform(10, 20) 18.376161340514045
GlobalRNG().normal@!() 1.7189974518331508
GlobalRNG().normal(10, 20) 15.259720132769834
GlobalRNG().exponential(1) 0.54213976708769562
GlobalRNG().gammal(0.5) 0.023344422176525669
GlobalRNG().gammal(1.5) 3.0564205547857468
GlobalRNG().weibull1(20) 0.95422285166630827
GlobalRNG().erlang(10, 2) 19.44737361504022
GlobalRNG().chiSquared(10) 13.561992437780951
GlobalRNG().t(10) -0.51654593773277468
GlobalRNG().logftormal(10, 20) 3.9190681459327817e+023
GlobalRNG().beta(9.5, 0.5) 0.058201077636632362
GlobalRWG().F(10, 20) 9.3580019590627147
GlobalRNG().cauchy(0, 1) 4.700499584114346
GlobalRNG().Levy() 8.4811381466439535
GlobalRNG().syrametrizedLevy() -0.11641736404696633
GlobalRNG().binomial(0.7, 20) 14
GlobalRNG().geometric(0.7) 1
GlobalRNG().poisson(0.7) 1
```

Рис. 6.11. Пример вывода рассмотренных генераторов

## 6.13. Генерация случайных объектов

Некоторые объекты технически невозможно сгенерировать правильно, потому что количество битов, необходимых для хранения объекта, больше, чем способен хранить генератор, но для практическических вычислений это не является проблемой.

В математических расчетах бывает полезен случайный единичный вектор. Чтобы решить нормализацию, избегайте генерации нулевых векторов. Поэтому запретите значение 0, генерируя каждый компонент из  $\pm u$ :

```
Vector<double> randomUnitVector(int n)
{
    Vector<double> result(n);
    for(int i = 0; i < n; ++i) result[i] = uniform01() * sign();
    result *= 1/norm(result);
    return result;
}
```

Чтобы случайным образом переставить  $n$  элементов, замените первый случайным элементом, а затем случайным образом переставьте оставшиеся  $n - 1$ , как показано на рис. 6.12 и в следующем коде:

```
template<typename ITEM> void randomPermutation(ITEM* numbers, int size)
{
    for(int i = 0; i < size; ++i)
        swap(numbers[i], numbers[inRange(i, size - 1)]);
}
```

Время выполнения—  $O(n)$ , что быстрее, чем сортировка со случайными приоритетами. Но зато она работает быстрее при использовании внешней памяти (см. главу 13. Алго-

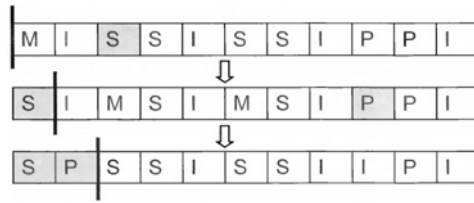


Рис. 6.12. Рекурсивная логика случайной перестановки: выберите S для позиции 0 и P для 1

ритмы для работы с внешней памятью). Чтобы сгенерировать упорядоченную выборку из  $k$  целых чисел в диапазоне  $[0, n - 1]$ , выберите каждое из них с вероятностью:

$$\frac{k - n\text{Selected}}{n - n\text{Considered}}.$$

Тогда значение 0 попадет в выборку с вероятностью  $k/n$ , а 1 — с вероятностью  $(k - 1)/(n - 1)$ , если выбран 0, или с  $k/(n - 1)$ , если нет. Что дает полную вероятность  $\frac{k-1}{n-1} \frac{k}{n} + \frac{k}{n-1} \left(1 - \frac{k}{n}\right) = \frac{k}{n}$  и т. д. Таким образом, каждый элемент выбирается с вероятностью  $k/n$ , и выбирается  $k$  элементов, потому что, если  $n\text{Selected} = k$ , вероятность = 0, и если вероятность = 1, все последующие элементы будут выбраны:

```
Vector<int> sortedSample(int k, int n)
{
    assert(k <= n && k > 0);
    Vector<int> result;
    for(int considered = 0, selected = 0; selected < k; ++considered)
        if(bernoulli(double(k - selected)/(n - considered)))
        { // выбор
            result.append(considered);
            ++selected;
        }
    return result;
}
```

Время выполнения —  $O(n)$ , и алгоритм может создавать выборки в режиме реального времени, хотя здесь это не реализовано.

Он также эффективно производит комбинации целых чисел  $\in [0, n - 1]$  в отсортированном порядке, который легко рандомизировать:

```
Vector<int> randomCombination(int k, int n)
{
    Vector<int> result = sortedSample(k, n);
    randomPermutation(result.getArray(), k);
    return result;
}
```

Создание выборки из резервуара — это поиск выборок размера  $k$  из потока неизвестного размера за один проход с затратами  $O(1)$  на единицу данных:

- ◆ Выбрать первые  $k$  элементов.
- ◆ Для  $n > k$ , когда прибывает  $n$ -й элемент, пусть  $r$  = случайное число  $\% n$ .
- ◆ Если  $r < k$ , заменить  $r$ -й элемент новым.

```
template<typename ITEM> struct ReservoirSampler
{
    int k, nProcessed;
    Vector<ITEM> selected;
    void processItem(ITEM const& item)
    {
        ++nProcessed;
        if(selected.getSize() < k) append(item); // выбор первого элемента k
        else
        { // случайная замена
            int kickedOut = GlobalRNG().mod(nProcessed);
            if(kickedOut < k) selected[kickedOut] = item;
        }
    }
    ReservoirSampler(int wantedSize): k(wantedSize), nProcessed(0){}
};
```

Сортировка (или множественный выбор — см. главу 7. *Сортировка*)  $n$  случайных величин дает упорядоченную статистику. Для статистики только  $i$ -го порядка из  $n$  обычно более эффективно генерировать однородную  $i$ -ю статистику (начиная с 1),  $\sim \text{beta}(i, n - i + 1)$  (см. [6.1]), и применить к ней обратный метод:

```
double uniformOrderStatistic(int i, int n){return beta(i, n - i + 1);}
```

## 6.14. Генерация выборок из многомерных распределений

Принятие/отклонение, преобразование между распределениями и композиция работают для  $D > 1$ , а обратный метод — нет. Принятие/отклонение неэффективны для большого количества измерений, но хороших альтернатив иногда нет вообще. У некоторых распределений есть свои эффективные генераторы. Методы МСМС (они более продвинутые, и о них мы поговорим в главе 21. *Вычислительная статистика*) могут быть более эффективными.

Примером принятия/отклонения является генерация точки на единичной окружности: генерируем точку  $p \in (-1, 1)^2$  до тех пор, пока расстояние  $(p, (0, 0)) \leq 1$ :

```
pair<double, double> pointInUnitCircle()
{
    double x = uniform(-1, 1), y = uniform(-1, 1);
    while(x * x + y * y > 1)
    { // повторяем генерацию, если значение повторяется
        x = uniform(-1, 1);
        y = uniform(-1, 1);
    }
    return make_pair(x, y);
}
```

Для многомерного нормального распределения со средним значением  $\mu$  и ковариационной матрицей  $\Sigma$  сгенерируем вектор  $V$  нормальных распределений  $(0, 1)$  и вернем  $\mu + L \times V$ , где  $L$  вычисляется с помощью факторизации Холецкого по  $\Sigma$  (см. главу 22.

*Численные алгоритмы: введение и матричная алгебра*). Этот метод работает, поскольку результирующая переменная имеет среднее значение  $\mu$  и дисперсию  $LL^T = \Sigma$ ;

```
class MultivariateNormal
{
    Vector<double> means;
    Matrix<double> L;
    static Matrix<double> makeL(Matrix<double> const& covariance)
    {
        Cholesky<double> c(covariance);
        assert(!c.failed); // ковариация может быть неправильной
                           // или создавать числовые проблемы
        return c.l;
    }
public:
    MultivariateNormal(Vector<double> const& theMeans, Matrix<double> const&
        covariance): means(theMeans), L(makeL(covariance)) {}
    Vector<double> next()const
    {
        Vector<double> normals;
        for(int i = 0 ; i < means.getSize(); ++i)
            normals.append(GlobalRNG().normal01());
        return means + L * normals;
    }
};
```

Тестирование результата затруднено еще и тем, что статистическое распределение тестов типа Колмогорова — Смирнова (см. главу 21. *Вычислительная статистика*) работает только для одного измерения.

## 6.15. Метод Монте-Карло

Из расчета средних значений и стандартных отклонений можно получить немало информации. Сначала немного теории:

♦ **Закон больших чисел (LLN)**: для  $n$  выборок  $x_i$  таких, что  $E[x_i] = \mu$ , *среднее значение*

$$\text{равно: } \bar{x} = \frac{\sum x_i}{n} \rightarrow \mu \text{ для } n \rightarrow \infty.$$

♦ **Выборочная дисперсия**  $s^2 = \frac{\sum (x_i - \bar{x})^2}{n-1} \rightarrow$  дисперсия  $\sigma^2$ . Используйте  $n-1$  вместо  $n$ ,

чтобы скорректировать погрешность. Система оценки  $E[(x_j - \bar{x})^2]$  не работает, потому что  $x_j$  влияет на  $\bar{x}$ .

♦ **Центральная предельная теорема (CLT)**: для данных  $n$  выборок  $y_i$  из распределения с конечным  $\sigma^2$  для  $n \rightarrow \infty$ ,  $\bar{x} \sim \text{normal}(\mu, \sigma^2/n)$ .

♦ **LLN и теорема Слуцкого** (см. [6.9]) позволяют использовать  $s^2$  вместо  $\sigma^2$  (для применения теоремы используйте тот факт, что  $s^2/\sigma^2 \rightarrow 1$ ).

♦ Таким образом,  $\bar{x}$  оценивает  $\mu$  с дисперсией  $s^2/n$ . Не путайте это с дисперсией данных — это дисперсия среднего значения в качестве оценки.

- ◆ Для полученной оценки ошибки  $2s/\sqrt{n}$  с доверительной вероятностью 95% необходимо соблюдение некоторых условий:
  - $n \geq 30$ . В противном случае  $t$ -распределение (см. главу 21. *Вычислительная статистика*) дает лучшее приближение;
  - среднее значение сходимости к нормальному для этого  $n$  обычно непредсказуемое.

Например, для конкретной выборки и соответствующего интервала  $\bar{x} \pm \varepsilon$  доверительный интервал 95% означает 5%-ную вероятность ложного охвата, т. е. для некоторых выборок расчетный интервал, который фиксируется после расчета, не будет охватывать  $\mu$ . При каждом моделировании у вас будут получаться интервалы немного разной длины с разными центрами, и изредка могут возникать небольшие отличия. Дополнительные сведения приведены в главе 21. *Вычислительная статистика*.

Метод Монте-Карло заключается в применении CLT для вычисления интересующей величины  $\mu$ , если:

- ◆ существует функция генератора событий  $f$ , создающая iid событий со значением  $y_i$ , таких что  $E[y_i] = \mu$ ;
- ◆ все важные события генерируются достаточно часто, чтобы гарантировать, что  $\mu$  существует, а  $n$  не должно быть слишком большим для срабатывания CLT (это интуитивное, а не формальное требование).

1. Задаемся значениями  $\mu$  и  $f$ .
2. Пока не кончится терпение или ошибка не станет достаточно мала:
3.  $y_i \leftarrow f(x_i)$ .
4. Пошагово обновляем  $\bar{x}$  и  $s^2$  с помощью  $y_i$ .
5. Возвращаем  $\mu \leftarrow \bar{x}$  с вероятностью 95% асимптотической ошибки  $2s/\sqrt{n}$ .

Если предположить, что генерация события занимает  $O(1)$  времени и пространства, то симуляция занимает  $O(n)$  времени и  $O(1)$  пространства, поскольку  $x_i$  хранить не нужно. Чтобы получить модуль ошибки меньший, чем  $< \varepsilon$ , необходимо  $n = (1/\varepsilon)^2$ . Это значение обычно бывает слишком велико, поэтому ориентируйтесь на имеющиеся вычислительные мощности и примите полученную точность как наилучшую доступную.

Для примера рассмотрим вычисление числа  $\pi$ . Площадь круга с радиусом  $r$  равна  $\pi r^2$ , а площади окружающего его квадрата  $4r^2$ , поэтому:

$$\mu = \frac{\pi}{4} = \frac{\text{площадь (круга)}}{\text{площадь (квадрата)}}.$$

Пусть  $f$  генерирует случайную точку  $p \in [-1, 1]^2$  и возвращает  $y_i = (\text{расстояние } (p, (0,0)) \leq 1)$ . Значения  $y_i = 1$  и  $y_i = 0$  должны встречаться часто, поэтому малого  $n$  будет достаточно. После  $10^8$  генераций число  $\pi$  становится равно  $3,14182 \pm 0,000493$  (рис. 6.13). Другие прогоны дадут другие оценки, но с такими же достоверными значениями ошибок.

Некоторые задачи, для которых метод Монте-Карло не подходит:

- ◆ оценка  $\mu$  выборок из распределения Коши, т. к. их не существует. Алгоритм вернет произвольную оценку в зависимости от используемых выборок;



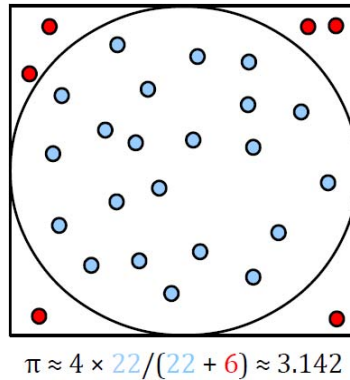


Рис. 6.13. Оценка числа  $\pi$  по количеству точек в окружности / количеству точек в квадрате

- ♦ оценка среднего дохода в городе, где живет миллиардер. Результат получится очень разным в зависимости от того, включен миллиардер в выборку или нет, и в обоих случаях бессмысленный. Это, возможно, лучшая иллюстрация определения «с достоверностью 95%», т. к. наборы данных отличаются, и порой сильно.

Метод Монте-Карло можно расширить несколькими способами:

- ♦ для расчета ошибки с достоверностью 95% на самом деле требуется множитель 1,96, но вы можете использовать множитель 2 и получите 95,45%. 95% — это стандарт в статистике, потому что большее значение достоверности часто бывает обманчивым, а меньшее — недостаточным. Учитывая множитель, можно оценить нормальную CDF, чтобы получить соответствующую достоверность (см. главу 21. *Вычислительная статистика*). Имейте в виду, что значение основано на асимптотической нормальности и в лучшем случае приблизительно точно, поэтому точный процент здесь не важен;
- ♦ смоделированное событие может выдать  $k$  значений. Мы выполняем  $k$  связанных симуляций по цене одной. В случае одновременной оценки ошибок остерегайтесь многократного тестирования (см. главу 21. *Вычислительная статистика*).

В конце симуляции нужно вычислить структуру сводных данных нормального распределения, чтобы убедиться, что в ней содержится среднее значение и дисперсия среднего значения. Нормальное распределение позволяет складывать и вычитать независимые нормальные распределения и масштабировать их по константе. Масштабирование в 2 раза дает не тот же результат, что прибавление значения к самому себе, потому что только сложение относится к независимым распределениям. Также можно периодически вычислять минимум и максимум, но ничего значимого в этих значениях не содержится (см. главу 21. *Вычислительная статистика*. Возможно, вместо этих значений нужен интервал допуска), а вычисляются они лишь для проверки:

```
struct NormalSummary: public ArithmeticType<NormalSummary>
{
    double mean, variance;
    double stddev()const{return sqrt(variance);}
    double error95()const{return 2 * stddev();}
    explicit NormalSummary(double theMean = 0, double theVariance = 0):
        mean(theMean), variance(theVariance){assert(variance >= 0);}
```

```

NormalSummary operator+=(NormalSummary const& b)
{ // для вычисления суммы складываем средние и дисперсии
    mean += b.mean;
    variance += b.variance;
    return *this;
}
NormalSummary operator--(NormalSummary const& b)
{ // для разницы вычитаем средние, но прибавляем дисперсии
    mean -= b.mean;
    variance += b.variance;
    return *this;
}
NormalSummary operator*=(double a)
{ // масштабируем как среднее значение, так и дисперсию
    mean *= a;
    variance *= a * a;
    return *this;
}
};

```

В расчете используется  $s^2 = \frac{\sum y_i^2 - \sum y_i^2 / n}{n - 1}$ , которая менее численно стабильная из-за возможного сокращения при вычитании  $n - 1$  (см. главу 22. Численные алгоритмы: введение и матричная алгебра). Но зато формула эта простая и позволяет постепенно обновлять  $\sum y_i$  и  $\sum y_i^2$ . Отмена не является проблемой с точностью до двойного слова, если ошибка уже не чрезвычайно мала по сравнению с величинами числителя. Поэтому можно с уверенностью предположить, что ошибка в случае отрицательного числителя равна 0:

```

struct IncrementalStatistics
{
    double sum, squaredSum, minimum, maximum;
    long long n;
    IncrementalStatistics(): n(0), sum(0), squaredSum(0),
        minimum(numeric_limits<double>::infinity()), maximum(-minimum){}
    double getMean()const{return sum/n;}
    double getVariance()const{return n < 2 ? 0 :
        max(0.0, (squaredSum - sum * getMean())/(n - 1.0));}
    double stdev()const{return sqrt(getVariance());}
    void addValue(double x)
    { // обновляем инкрементные переменные
        ++n;
        maximum = max(maximum, x);
        minimum = min(minimum, x);
        sum += x;
        squaredSum += x * x;
    }
    NormalSummary getStandardErrorSummary()const
    {return NormalSummary(getMean(), getVariance()/n);}
};

```

```
template<typename FUNCTION> IncrementalStatistics MonteCarloSimulate(
    FUNCTION const& f, long long nSimulations = 1000)
{
    IncrementalStatistics s;
    for(long long i = 0; i < nSimulations; ++i) s.addValue(f());
    return s;
}
```

**Хвостовое неравенство (неравенство Милля)** показывает, что вероятность очень большой ошибки в оценке среднего экспоненциально снижается. Пусть:

$$z = \frac{x - \bar{x}}{s}; \text{ тогда } \Pr(Z > z) < \sqrt{\frac{2}{\pi}} \frac{f(z)}{z},$$

где  $f$  — нормальная плотность вероятности (см. [6.9]). Однако это относится к смоделированным данным только асимптотически.

Метод Монте-Карло может помочь найти  $E[\text{использование ресурсов}]$  рандомизированного алгоритма. Для проверки скорости используйте следующий код:

```
template<typename FUNCTION> struct SpeedTester
{
    FUNCTION f;
    SpeedTester(FUNCTION const& theFunction = FUNCTION()): f(theFunction){}
    double operator() () const
    {
        int now = clock();
        f();
        return (clock() - now) * 1.0/CLOCKS_PER_SEC;
    }
};
```

Можно применять метод Монте-Карло для сравнения средней производительности рандомизированных алгоритмов. Поскольку  $\text{normal}(a, b) - \text{normal}(c, d) = \text{normal}(a - c, b + d)$ , с вероятностью 95% можно заключить, что  $a > c$ , если  $0 \notin [a - c \pm 2 \sqrt{b + d}]$ . Но чтобы избежать многократного тестирования, это должен быть единственный вывод, т. е. нельзя выдавать данные об этом интервале и интервалах для средних значений.

Проблемы метода Монте-Карло:

- ◆ события могут коррелировать или образовывать коррелированные последовательности. Это позволяет усреднить коррелированные подпоследовательности и предположить, что полученные *средние значения всей последовательности* независимы;
- ◆ сходимость метода слишком медленная, потому что ошибка стремится к  $O(1/\sqrt{n})$ . Когда  $x_i$  исходят из многомерного пространства, CLT, по-видимому, преодолевает проблемы размерности, т. к. скорость сходимости от размерности не зависит. А дисперсия  $y_i$  может зависеть от размерности  $x_i$ , поэтому влияет на константу в «Большой O»;
- ◆ мало событий, которым для работы с CLT требуется сколь угодно большое значение  $n$ . Событие, требующее, чтобы монетка выпала решкой 100 раз подряд, весьма маловероятно.

Методы *снижения дисперсии* несколько ускоряют сходимость CLT. Они выполняют такое же моделирование, но с меньшей дисперсией. *Обычные случайные числа* берут на себя все, что не моделируется. Например, при сравнении производительности рандомизированных алгоритмов вы можете запускать все симуляции на фиксированном наборе входных данных вместо создания нового набора для каждого запуска. Для тестирования на различных входных данных каждый набор можно использовать несколько раз. Этот метод кажется полезным только для парных сравнений, а неправильное его применение может породить систематическую ошибку из-за неслучайности, что способно привести к изменению выводов.

Для моделирования системы обработки запросов обычно используют приоритетную очередь (см. главу 10. *Приоритетные очереди*) событий, запланированных на некоторое абсолютное или относительное время. Алгоритм таков: пока в очереди есть событие, мы извлекаем его, выполняем и, может быть, добавляем в очередь что-то еще.

Я придумал и задавал на множестве собеседований интересный вопрос: как найти приблизительную площадь набора возможных перекрывающихся кругов, заданных центрами и радиусами, при наличии неограниченных, но конечных вычислительных ресурсов. Пытаться найти формулы пересечения нескольких окружностей в случае «цветка» бесполезно. Но метод Монте-Карло работает (рис. 6.14):

1. Вычислить ограничивающую рамку кругов.
2. Написать функцию, определяющую, находится ли точка внутри круга.
3. Использовать привязку  $O(1/\sqrt{n})$  для оценки количества выборок  $n$  при некоторой приблизительной целевой точности, такой как 0,001.
4. Создать  $n$  точек внутри поля и определить, находится ли она внутри какого-либо круга.

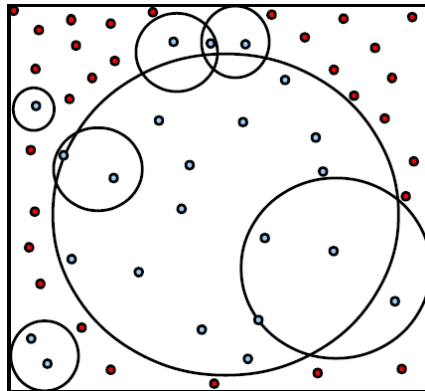


Рис. 6.14. Решение задачи про пересекающиеся круги методом Монте-Карло

Чуть более сложный, но принципиально похожий метод — подсчет пикселей в искусственном изображении. Его я даю в качестве подсказки тем, кто не знаком с методом Монте-Карло. Далее приведено решение для моделирования, написанное на Python:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```

class Circle:
    def __init__(self, p, r):
        self.p = p
        self.r = r
import math
def distance(p1, p2):
    return math.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2)
def isInCircle(p, circles):
    for c in circles:
        if distance(p, c.p) <= c.r:
            return True
    return False
def findBoundingBox(circles):
    inf = float("inf") # В моем python нет math.inf
    left = inf
    right = -inf
    down = inf
    up = -inf
    for c in circles:
        if c.p.x - c.r < left:
            left = c.p.x - c.r
        if c.p.x + c.r > right:
            right = c.p.x + c.r
        if c.p.y - c.r < down:
            down = c.p.y - c.r
        if c.p.y + c.r > up:
            up = c.p.y + c.r
    return [Point(left, up), Point(right, down)]
import random
def findCoveredArea(circles, nSimulations):
    box = findBoundingBox(circles)
    left = box[0].x
    right = box[1].x
    down = box[1].y
    up = box[0].y
    inCount = 0
    for i in range(0, nSimulations):
        if isInCircle(Point(random.uniform(left, right),
            random.uniform(down, up)), circles):
            inCount += 1
    return float(inCount)/nSimulations * (up - down) * (right - left)
print findCoveredArea([Circle(Point(0, 0), 1)], 1000000) # Оценка числа Pi

```

## 6.16. Примечания по реализации

Возможно, мы рассмотрели слишком много генераторов. Но у всех них есть свои недостатки.

При создании распределения встает вопрос: какие использовать? Некоторые из представленных здесь распределений используются нечасто, за исключением особых. Многие другие не используются вообще. Мы отобрали только самые простые алгоритмы.

Реализация кучи сумм занимает больше всего времени из-за всех исследований, направленных на поиск хотя бы одного источника для довольно очевидной идеи.

При моделировании всегда нужно решать, что в него включать. Хорошо бы сосредоточиться на получении оценки с помощью базовой меры точности, основанной на норме. Нормальная оценка предназначена для облегчения манипулирования результатами. Минимальную и максимальную статистику вычислить легко, и они потенциально полезны для отладки, но не для статистических целей. Было бы неправильно удалить этот код.

## 6.17. Комментарии

Если бинарный источник генерирует «решку» с вероятностью  $p \neq 0,5$ , можно получить несмещенные биты:

1. Бросьте монетку дважды, пока не получите «решка-орел» или «орел-решка».
2. Если получили «решка-орел», верните решку.
3. В противном случае верните орла.

Таким образом, орел и решка выпадают с равной вероятностью.  $E[\text{число\_бросков}] = \frac{1}{p(1-p)}$  не имеет конечной границы для наихудшего случая, а большие значения

экспоненциально маловероятны. Но этот метод всего лишь любопытен как разминка для ума. Проще получить правильно случайные или псевдослучайные числа.

Идея кучи сумм с тернарной структурой узлов оригинальна (насколько мне известно). Например, дерево Вонга и Истона (см. [6.13]) имеет другую структуру.

CLT обобщает выходные данные векторной функции, и в этом случае вместо дисперсии используется ограничивающая матрица корреляции, но этот метод менее полезен.

Другие методы уменьшения дисперсии, в том числе *антитетические вариации*, *контрольные вариации* и *выборка по важности*, бесполезны для работы методом «черного ящика», потому что нужно знать многое о системе, а уменьшение ошибки равно всего лишь  $O(1)$ , в то время как моделирование наиболее полезно, когда о системе не известно ничего. Но выборка по важности может быть полезна для выборки редких событий, если какие-то аналитические знания о системе все-таки есть.

## 6.18. Советы по дополнительной подготовке

- ◆ Улучшите 32-битный Xorshift для преобразования вывода с помощью LCG с множителем 1099087573 (см. [6.4]). Хеш-функция должна стать лучше.
- ◆ Расширьте генератор нормалей, чтобы он возвращал пару переменных. Сделайте вокруг него удобную оболочку и выбирайте по одному.
- ◆ Измените геометрический генератор, чтобы он работал за время  $O(1)$ , используя специальную формулу (см. например, [6.2]). Ограничивает ли число битов на выходе генератора полезный диапазон переменных, в отличие от метода грубой силы?
- ◆ Исследуйте численно устойчивые формулы для пошагового вычисления среднего.

- ◆ Имеет ли смысл отключать потенциальные «вечные» генераторы, такие как генератор для точки в круге, чтобы избежать бесконечных циклов? Одна из стратегий заключается в выдаче исключения после неоправданно большого количества попыток.
- ◆ Преобразуйте решение Python из задачи о площади круга в C++. Реализуйте решение для подсчета пикселей и сравните их. Если вы проводите собеседования, задайте этот вопрос несколько раз и посмотрите, как кандидаты будут на него отвечать. Требуется ли им подсказка, чтобы выбрать тот или иной подход? Какой из них правильнее реализуется? Понаблюдайте за другими интересными моментами.

## 6.19. Список рекомендуемой литературы

- 6.1. Devroye L. (1986). *Non-uniform Random Variate Generation*. Springer.
- 6.2. Kroese D. P., Taimre T., & Botev Z. I. (2011). *Handbook of Monte Carlo Methods*. Wiley.
- 6.3. L'Ecuyer P. (1999). Good parameters and implementations for combined multiple recursive random number generators. *Operations Research* 47: 159–164.
- 6.4. L'Ecuyer P. & Simard R. (2007). TestU01: a C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*, 33(4), 22.
- 6.5. L'Ecuyer P., Simard R., Chen E. J., & Kelton W. D. (2002). An object-oriented random number package with many long streams and substreams. *Operations Research*, 50(6), 1073–1075.
- 6.6. Marsaglia G., & Tsang W. W. (2000). A simple method for generating gamma variables. *ACM Transactions on Mathematical Software (TOMS)*, 26(3), 363–372.
- 6.7. Matsumoto M., & Nishimura T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1), 3–30.
- 6.8. Press W. H., Teukolsky S. A., Vetterling W. T., & Flannery B. P. (2007). *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge University Press.
- 6.9. Wasserman L. (2004). *All of Statistics: A Concise Course in Statistical Inference*. Springer.
- 6.10. Wikipedia (2013). RC4. <http://en.wikipedia.org/wiki/RC4>. Accessed May 12, 2013.
- 6.11. Wikipedia (2015). Triangular distribution. [https://en.wikipedia.org/wiki/Triangular\\_distribution](https://en.wikipedia.org/wiki/Triangular_distribution). Accessed November 1, 2015.
- 6.12. Wikipedia (2017). Lévy distribution. [https://en.wikipedia.org/wiki/L%C3%A9vy\\_distribution](https://en.wikipedia.org/wiki/L%C3%A9vy_distribution). Accessed December 10, 2017.
- 6.13. Wong C. K., & Easton M. C. (1980). An efficient method for weighted sampling without replacement. *SIAM Journal on Computing*, 9(1), 111–113.

# 7. Сортировка

## 7.1. Введение

Сортировка часто изучается на занятиях по алгоритмам и почти всегда должна выполняться с помощью функций стандартной библиотеки, даже если применяется специальный, более эффективный алгоритм. В этой главе основное внимание уделяется деталям реализации наиболее полезных алгоритмов.

Немного теории:

- ◆ набор элементов, удовлетворяющих *слабому отношению порядка* « $\leq$ », сортируется. Помните, что отношение « $\leq$ » не является слабым;
- ◆ для  $n$  элементов существует  $n!$  порядков, и  $k$  бинарных сравнений сортируют  $\geq 2k$  из них, поэтому для сортировки с использованием только сравнений необходимо  $k = \Theta(n \lg(n))$  операций;
- ◆ сортировка *стабильна*, если равные элементы сохраняют свой первоначальный относительный порядок. Использование местоположения элемента в качестве *вторичного ключа* обеспечивает стабильность, но требует больших затрат памяти и само по себе неудобно для вызывающей стороны;
- ◆ анализ сортировки элементов обычно предполагает  $O(1)$  сравнений. Для повышения эффективности сортировки дорогостоящих элементов отсортируйте массив указателей на них.

## 7.2. Сортировка вставками

Для небольших массивов сортировка вставками стабильна и является самой быстрой, поэтому она часто используется как вспомогательная в некоторых более сложных алгоритмах. Она похожа на сортировку карт. Имея отсортированный массив первоначально из одного элемента, мы итеративно вставляем следующий элемент в нужное место (рис. 7.1).



Рис. 7.1. Логика следующего несортированного элемента сортировки вставками

```
template<typename ITEM, typename COMPARATOR>
void insertionSort(ITEM* vector, int left, int right, COMPARATOR const& c)
{ // Разрешаем выполнение left != 0
  for(int i = left + 1; i <= right; ++i)
```



```

{
    ITEM e = vector[i];
    int j = i;
    for (; j > left && c(e, vector[j - 1]); --j) vector[j] = vector[j - 1];
    vector[j] = e;
}

```

Время выполнения составляет  $O(n^2)$  с очень низкими постоянными компонентами и  $O(\text{количество перевернутых пар, называемых инверсиями})$ , что для почти отсортированного входного массива дает  $O(n)$ .

## 7.3. Быстрая сортировка

Если вам не нужна стабильность, самый быстрый и почти универсальный алгоритм в любом хорошем API — это быстрая сортировка (рис. 7.2). Его базовый алгоритм:

1. Выбираем опорный элемент.
2. Разделяем массив так, чтобы элементы  $\leq$  опорной точки находились соответственно слева/справа.
3. Сортируем обе половины рекурсивно.

Порядок элементов в каждом подмассиве после раздела не имеет значения.

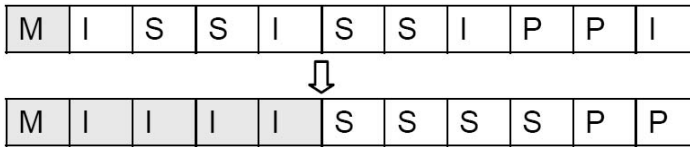


Рис. 7.2. Возможный результат быстрой сортировки

Наиболее практичным правилом выбора опорной точки является *медиана трех* случайных элементов:

- ◆ детерминированный выбор опорной точки может дать время выполнения  $O(n^2)$ ;
- ◆ выбор одиночной случайной опорной точки работает немного медленнее;
- ◆ выбор пяти или более случайных элементов работает быстрее, но реализуется сложнее.

```

template<typename ITEM, typename COMPARATOR>
int pickPivot(ITEM* vector, int left, int right, COMPARATOR c)
{ // хорошо, когда опорные точки случайно совпадают
    int i = GlobalRNG().inRange(left, right), j =
        GlobalRNG().inRange(left, right), k = GlobalRNG().inRange(left, right);
    if(c(vector[j], vector[i])) swap(i, j);
    // неравенство i <= j решает, куда пойдет k
    return c(vector[k], vector[i]) ? i : c(vector[k], vector[j]) ? k : j;
}

```

*Разделение* — это группировка элементов на те, что больше, меньше или равны опорному (см. [7.7]). В отличие от полной сортировки, элементы в разделах «<» и «>» будут

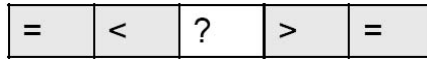


Рис. 7.3. Стратегия разделения при быстрой сортировке

располагаться в произвольном порядке. В качестве промежуточного шага поместите равные элементы по бокам (рис. 7.3).

Левый и правый указатели используются для двунаправленного сканирования массива. Если отсканированный элемент не относится к соответствующему разделу «<» или «>», пометьте его как требующей замены. На каждой итерации указатели останавливаются на таких элементах. Остановитесь, когда стрелки пересекутся. Затем поменяйте местами боковые секции «=» на средние:

```
template<typename ITEM, typename COMPARATOR> void partition3(ITEM* vector,
    int left, int right, int& i, int& j, COMPARATOR const& c)
{ // i/j - текущие указатели влево/вправо
  ITEM p = vector[pickPivot(vector, left, right, c)];
  int lastLeftEqual = i = left - 1, firstRightEqual = j = right + 1;
  for(;;) // после одного обмена замененные элементы
    // действуют как опорные
  { // меняем местами равные элементы
    while(c(vector[++i], p));
    while(c(p, vector[--j]));
    if(i >= j) break; // указатели пересеклись
    swap(vector[i], vector[j]); // оба указателя нашли элементы, требующие замены
    // меняем местами равные элементы по бокам
    if(c.isEqual(vector[i], p)) // i влево
      swap(vector[++lastLeftEqual], vector[i]);
    if(c.isEqual(vector[j], p)) // j вправо
      swap(vector[--firstRightEqual], vector[j]);
  }
  // инвариант: i == j в случае остановки на опорном элементе,
  // и это может произойти как с левым, так и с правым элементом
  // или они пересекаются и i = j + 1
  if(i == j){++i; --j;} // не трогайте центральную точку
  // меняем боковые элементы на середину; слева "<" и справа ">"
  for(int k = left; k <= lastLeftEqual; ++k) swap(vector[k], vector[j--]);
  for(int k = right; k >= firstRightEqual; --k)
    swap(vector[k], vector[i++]);
}
```

Либо  $i$ , либо  $j$  могут оказаться за пределами, и это определяет постусловие (рис. 7.4).

С кодом нужно работать осторожно, ибо в нем легко ошибиться, особенно с крайними значениями для циклов `while`. Чуть менее сложно реализуется разбиение « $\leq/\geq$ », в кото-

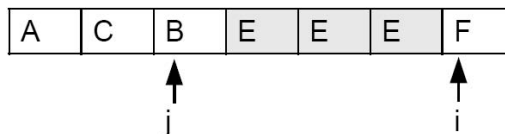


Рис. 7.4. Указатели быстрой сортировки после разделения по трем элементам

ром особого внимания равным элементам не уделяется. Оно получается простым удалением кода для перестановки в стороны и обратно. Инвариант  $i-j$  остается прежним, но он определяет постусловие. Разделение по трем элементам также применяется к векторной сортировке и выполняется быстрее для массивов с большим количеством одинаковых элементов. При наличии небольшого числа дополнительных элементов проверка равенства и перестановок добавляется немного.

Оптимизация основного алгоритма:

- ◆ сортировка меньших подмассивов сначала обеспечивает  $O(\lg(n))$  дополнительной памяти, поэтому стек рекурсии почти гарантированно не закончится;
- ◆ используйте сортировку вставками для небольших подмассивов размером 16. Из-за кеширования рекурсия в сортировке вставками в конце концов выполняется быстрее, чем одиночная сортировка вставками по всему массиву, несмотря на использование большего количества инструкций (см. [7.5]);
- ◆ удалите хвостовую рекурсию. Удаление другой рекурсии усложняет алгоритм, а выгоды дает мало.

```
template<typename ITEM, typename COMPARATOR>
void quickSort(ITEM* vector, int left, int right, COMPARATOR const& c)
{
    while(right - left > 16)
    {
        int i, j;
        partition3(vector, left, right, i, j, c);
        if(j - left < right - i) // выбираем меньшую
        {
            quickSort(vector, left, j, c);
            left = i;
        }
        else
        {
            quickSort(vector, i, right, c);
            right = j;
        }
    }
    insertionSort(vector, left, right, c);
}

template<typename ITEM> void quickSort(ITEM* vector, int n)
{quickSort(vector, 0, n - 1, DefaultComparator<ITEM>());}
```

При базовом разделении  $E[\text{время выполнения}]$  равняется  $O(n \lg(n))$ . Доказательство приведено в работе [7.1]: предположим, что точка опоры выбрана случайно, а все элементы уникальны. Пусть  $X_{ij}$  — количество раз, когда элемент в  $i$  сравнивался с элементом в  $j$  в отсортированном массиве с  $j > i$ .  $E[X_{ij}] = \Pr(i \text{ или } j \text{ является опорным})$ , поскольку  $i$  и  $j$  сравнивались не более одного раза и только в том случае, если одно из них было опорным в подмассиве, содержащем другое. В противном случае, если элемент в диапазоне  $> j$  или  $< i$  был опорным,  $i$  и  $j$  попадают в один и тот же подмассив, иначе — в отдельные. Поскольку существует  $j - i + 1$  точек опоры:

$$\Pr(i \text{ или } j \text{ является опорным}) = \frac{2}{j - i + 1}.$$

Таким образом,

$$E[\text{общее количество сравнений}] = E\left(\sum_{0 \leq i < n} \sum_{i+1 \leq j < n} X_{ij}\right) < 2 \sum_{0 \leq i < n} \sum_{i+1 \leq k < n} \frac{1}{k} < 2n \lg(n).$$

Маловероятно, что наихудший случай будет иметь сложность  $O(n^2)$ , если каждый раз будут выбираться плохие опорные точки. Тот же анализ распространяется и на тройное разделение, потому что единственная разница заключается в сравнениях на равенство, которые на скорость не влияют.

## 7.4. Сортировка слиянием

Сортировка слиянием — самая эффективная стабильная сортировка (рис. 7.5):

1. Разделить массив пополам.
2. Выполнить сортировку слиянием рекурсивно.
3. Объединить половинки за время  $O(n)$ .

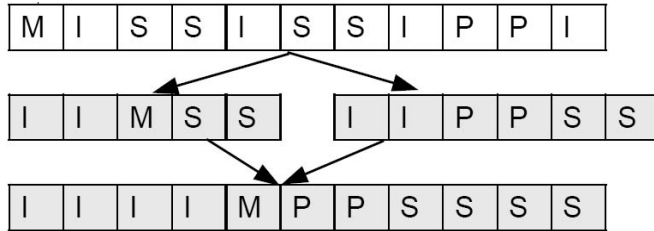


Рис. 7.5. Сортировка слиянием

Время выполнения удовлетворяет условию  $R(n) = O(n) + 2R(n/2)$ , поэтому по основной теореме  $R(n) = O(n \lg(n))$ . Оптимизации основного алгоритма:

- ♦ чередуйте массивы данных и временного хранения, чтобы избежать ненужных копий;
- ♦ используйте сортировку вставками для массивов размером  $\leq 16$ , как для быстрой сортировки.

Слияние занимает большую часть работы. Мы итеративно перемещаем наименьший крайний левый элемент обоих массивов в результирующее множество. Самый правый индекс левого массива — `middle`. Рекурсивный вызов и слияние предполагают, что для диапазона `[left, right]` элементы находятся во временном массиве хранения, так что исходный срез `[left, right]` перезаписывается отсортированным результатом:

```
template<typename ITEM, typename COMPARATOR> void merge(ITEM* vector,
    int left, int middle, int right, COMPARATOR const& c, ITEM* storage)
{ // i для левой половины, j для правой, объединяем,
  // пока не заполним вектор
  for(int i = left, j = middle + 1; left <= right; ++left)
  { // либо i, либо j могут выйти за пределы
    bool useRight = i > middle || (j <= right &&
      c(storage[j], storage[i]));
```

```

        vector[left] = storage[(useRight ? j : i)++];
    }
}

template<typename ITEM, typename COMPARATOR> void mergeSortHelper(
    ITEM* vector, int left, int right, COMPARATOR const& c, ITEM* storage)
{
    if(right - left > 16)
    { // сортируем хранилище, используя вектор в качестве хранилища,
      // затем объединяем в вектор
        int middle = (right + left)/2;
        mergeSortHelper(storage, left, middle, c, vector);
        mergeSortHelper(storage, middle + 1, right, c, vector);
        merge(vector, left, middle, right, c, storage);
    }
    else insertionSort(vector, left, right, c);
}

template<typename ITEM, typename COMPARATOR>
void mergeSort(ITEM* vector, int n, COMPARATOR const& c)
{ // сначала копируем вектор в хранилище
    if(n <= 1) return;
    Vector<ITEM> storage(n, vector[0]); // резервируем пространство
                                         // для n с первым элементом
    for(int i = 1; i < n; ++i) storage[i] = vector[i];
    mergeSortHelper(vector, 0, n - 1, c, storage.getArray());
}

```

Используемый алгоритм зависит от стабильности, потому что в противном случае будет использоваться быстрая сортировка, например:

- ◆ при сортировке таблицы по столбцу может неявно выполняться стабильная сортировка по нескольким столбцам, при этом другие столбцы являются непрерывным ключом;
- ◆ в некоторых алгоритмах может потребоваться избавиться от случайности быстрой сортировки, если есть вторичные ключи, влияющие на общие вычисления.

Возможно, сортировка слиянием в основном полезна для иллюстрации схемы ее работы, которая представляет собой общий принцип «разделяй-и-властвуй».

## 7.5. Целочисленная сортировка

Целые числа можно сортировать за время  $O(n)$ , не используя операцию «<». Для массива  $N$  используется *сортировка подсчетом* (рис. 7.6), которая вычисляет, сколько раз встречается каждое число, и создает отсортированный массив подсчетов за время  $O(n + N)$ . Сортировка работает стабильно.

```

void countingSort(int* vector, int n, int N)
{
    Vector<int> counter(N, 0);
    for(int i = 0; i < n; ++i) ++counter[vector[i]]; // подсчет
    for(int i = 0, index = 0; i < N; ++i) // пересоздание в правильном порядке
        while(counter[i]-- > 0) vector[index++] = i;
}

```

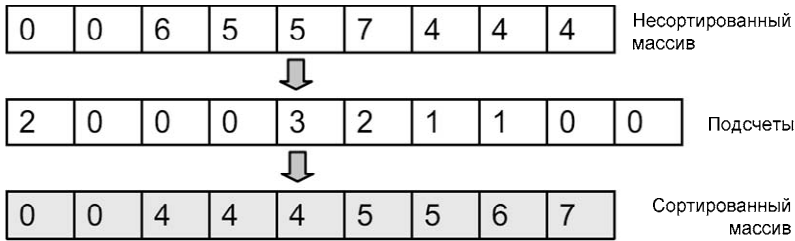


Рис. 7.6. Сортировка подсчетом с созданием отсортированного массива для  $N = 10$

Для элементов с целочисленными ключами по модулю  $N$  *сортировка с подсчетом по ключу* (KSort) работает аналогично, но для нее требуется временное хранилище, потому что из подсчетов нельзя создавать элементы. Можно использовать кумулятивные подсчеты, которые говорят, сколько элементов с меньшим значением предшествует элементу с определенным ключом. Ничто не предшествует самому маленькому элементу, поэтому для значения 0 тоже нужна метка. По мере размещения элементов счетчик приоритета увеличивается. Алгоритм KSort:

1. Подсчитываем, сколько элементов имеют определенный ключ.
2. Используем кумулятивный счетчик для создания временного отсортированного массива.
3. Копируем новый массив в оригинал.

Для этой реализации требуется функтор `ORDERED_HASH`, который извлекает ключи элементов. Например, он может получить байт 0 из целого числа. Помните, что поведение таких функторов зависит от порядка следования байтов:

```
template<typename ITEM, typename ORDERED_HASH> void KSort(ITEM* a, int n,
    int N, ORDERED_HASH const& h)
{
    ITEM* temp = rawMemory<ITEM>(n);
    Vector<int> prec(N + 1, 0);
    for(int i = 0; i < n; ++i) ++prec[h(a[i]) + 1];
    for(int i = 0; i < N; ++i) prec[i + 1] += prec[i]; // накопление
                                                    // счетчиков
    // перестановка элементов
    for(int i = 0; i < n; ++i) new(&temp[prec[h(a[i]) + 1]]) ITEM(a[i]);
    for(int i = 0; i < n; ++i) a[i] = temp[i];
    rawDelete(temp);
}
```

Алгоритм стабилен и занимает время  $O(n + N)$ .

## 7.6. Векторная сортировка

Сортировка  $n$  векторов размера  $k$  как *элементов* занимает время  $O(kn \lg(n))$ . Но их можно сортировать как *векторы*, особенно в случае косвенного доступа, при котором сортируется массив индексов, чтобы не копировать сами векторы. Для быстрой сортировки  $n$  случайных векторов длины  $\infty$  (см. [7.8]):

$$E[\text{время выполнения}] = O(n \lg(n)^2).$$

Это нетрудно заметить, поскольку для двух случайных последовательностей в наборе из  $n$  элементов из алфавита размера  $A$ ,  $E[\text{наименее распространенный префикс, НРП}] = \log A(n)$ , требуется именно такое время. В *главе 14. Строковые алгоритмы* параметр «наименее распространенный префикс» (least common prefix, lcp) рассматривается подробно.

Рассмотрим *многоключевую быструю сортировку* с тройным разделением по первой букве и рекурсией по каждому подмассиву (рис. 7.7).



**Рис. 7.7.** Рекурсивная трехсекционная многоключевая быстрая сортировка по последовательным символам слов

Пользовательский компаратор — например, приведенный далее компаратор векторов, отслеживает текущую глубину, начиная с 0, что позволяет сортировать кортежи произвольного размера. Для повторного использования кода требуется метод доступа к вектору и компаратор элементов. Также предусмотрен метод доступа по индексам:

```
template<typename VECTOR, typename ITEM> struct DefaultVectorAccessor
{ // для сортировки векторов
    ITEM const& getItem(VECTOR const& v, int i) const { return v[i]; }
    int getSize(VECTOR const& v) const { return v.getSize(); }
};

struct StringAccessor
{ // для сортировки строк
    char getItem(string const& s, int i) const { return s[i]; }
    int getSize(string const& s) const { return s.length(); }
};

template<typename VECTOR, typename ITEM, typename ACCESSOR =
    DefaultVectorAccessor<VECTOR, ITEM> > struct IndexedAccessor
{
    VECTOR const* const v;
    ACCESSOR a;
    IndexedAccessor(VECTOR const* const theV, ACCESSOR const& theA = ACCESSOR())
        : v(theV), a(theA) {}
    ITEM getItem(int i, int j) const { return a.getItem(v[i], j); }
    int getSize(int i) const { return a.getSize(v[i]); }
};

template<typename VECTOR, typename ITEM, typename ACCESSOR =
    DefaultVectorAccessor<VECTOR, ITEM>, typename COMPARATOR =
    DefaultComparator<ITEM> > struct MultikeyQuicksortVectorComparator
{
    ACCESSOR s;
    COMPARATOR c;
```

```

mutable int depth;
MultikeyQuicksortVectorComparator(ACCESSOR const& theS = ACCESSOR(),
    COMPARATOR const& theC = COMPARATOR()): s(theS), c(theC), depth(0){}
bool operator()(VECTOR const& lhs, VECTOR const& rhs) const
{
    return depth < s.getSize(lhs) ?
        depth < s.getSize(rhs) && c(s.getItem(lhs, depth),
            s.getItem(rhs, depth)) : depth < s.getSize(rhs);
}
bool isEqual(VECTOR const& lhs, VECTOR const& rhs) const
{
    return depth < s.getSize(lhs) ?
        depth < s.getSize(rhs) && c.isEqual(s.getItem(lhs, depth),
            s.getItem(rhs, depth)) : depth >= s.getSize(rhs);
}
};

```

Удалите рекурсию, чтобы не исчерпать объем стека при работе с длинными векторами с высоким значением НРП. Алгоритм управляется из стека, который содержит набор диапазонов и глубин для обработки. Начальная точка: (left, right, 0). Вам не следует:

- ◆ использовать сортировку вставками для небольших подмассивов, т. к. она не может эффективно работать с векторами;
- ◆ сначала обрабатывать меньший подмассив. Его время обработки либо такое же, либо в худшем случае больше, чем у большей части. Среднюю часть следует обрабатывать последней.

```

template<typename VECTOR, typename COMPARATOR> void multikeyQuicksortNR(
    VECTOR* vector, int left, int right, COMPARATOR c,
    int maxDepth = numeric_limits<int>::max())
{
    Stack<int> stack;
    stack.push(left);
    stack.push(right);
    stack.push(0);
    while(!stack.isEmpty())
    {
        c.depth = stack.pop();
        right = stack.pop();
        left = stack.pop();
        if(right - left > 0 && c.depth < maxDepth)
        {
            int i, j;
            partition3(vector, left, right, i, j, comparator);
            // левая часть
            stack.push(left);
            stack.push(j);
            stack.push(c.depth);
            // правая часть
            stack.push(i);
            stack.push(right);
            stack.push(c.depth);
        }
    }
}

```



```

        // средняя часть
        stack.push(j + 1);
        stack.push(i - 1);
        stack.push(c.depth + 1);
    }
}
}

```

$E[\text{время выполнения}] = O(n \lg(n))$ , а ожидаемое время выполнения по отношению к длине (length) оптимально и равно  $O(n (\text{length} + \lg(n)))$  (см. [7.7]). Маловероятный наихудший случай —  $O(n(\text{length} + n))$ .

Для векторов малых целых чисел фиксированной длины  $k$  стабильным и наиболее эффективным алгоритмом является *сортировка LSD*. Сортировка выполняется  $k$  раз, в качестве ключа для KSort на проходе  $i$  используется  $\text{vector}[k - i]$  с общим временем выполнения  $O(nk)$ . Алгоритм работает правильно, потому что KSort стабильна. Эта задача весьма редка, поэтому реализацию здесь мы не приводим. Вызывающая сторона должна настроить вызовы KSort. Для примера взгляните на построение массива суффиксов в *главе 14. Строковые алгоритмы*. Учтите, что неправильно использовать LSD для сортировки целых чисел побайтово, если только не задействован специальный регистр для доступа к нужным байтам, в противном случае их порядок может быть неправильным.

## 7.7. Сортировка перестановкой

Чтобы отсортировать массив  $a$  в соответствии с перестановкой  $p$ , заданной массивом отсортированных индексов, вы можете копировать элементы во временный массив и заполнять из него оригинал в соответствии с данным о перестановке  $p$ .

Но можно и не использовать временный массив (см. [7.2]). Любая перестановка состоит из независимых подперестановок (в терминологии абстрактной алгебры это *произведение непересекающихся циклов*). Пусть  $f = p[i]$  — это своего рода указание, откуда брать элемент для позиции  $i$ . Цикл по массиву получает все перестановки за время  $O(n)$ . Перестановка запускается, если  $p[i] \neq i$ .

1. Запомните первый элемент в перестановке и установите значение  $to =$  его индекс  $i$ .
2. Пока  $p[to] \neq to$ .
3.  $from = p[to]$ .
4.  $a[to] = a[from]$ .
5. Перестановка становится обработанной,  $p[to] = to$ .
6.  $to = from$ .
7. Завершите цикл с  $a[to] =$  первый элемент.

Для примера рассмотрим перестановку 3210, примененную к массиву  $abcd$ . Начнем с 0. Запоминаем  $v[0] = a$ , с позиции  $p[0] = 3$  берем букву  $d$  и помещаем ее в позицию 0. Проверяем условие  $p[3] = 0$ , чтобы увидеть, не настал ли конец цикла по условию  $p[0] = 0$ , и помещаем сохраненное  $a$  в позицию 3. Переходим в позицию 1. Та же логика меняет местами  $b$  и  $c$ . После этого все позиции становятся правильными, и переход на 2

и 3 ничего не меняет. Имейте в виду, что перестановка после выполнения теряет исходное значение, поэтому вызывающая сторона должна ее скопировать:

```
template<typename ITEM> void permutationSort(ITEM* a, int* permutation, int n)
{ // нужна правильная перестановка, иначе получится бесконечный цикл
    for(int i = 0; i < n; ++i) if(permutation[i] != i)
    { // начало цикла
        ITEM temp = a[i];
        int to = i;
        do
        {
            a[to] = a[permutation[to]]; // элемент помещается в корректную позицию
            swap(permutation[to], to); // перестановка выполнена,
                                     // продолжение цикла
        }while(permutation[to] != i); // выполняем, пока не найдем элемент i
        a[to] = temp; // цикл завершен
        permutation[to] = to; // идентификация
    }
}
```

Задачи, в которых нужен такой алгоритм, встречаются редко, потому можно получить отсортированный порядок через перестановку напрямую с помощью перебора или бинарного поиска.

## 7.8. Выбор

Иногда нужно расположить элементы так, чтобы указанный элемент находился в правильном месте, — например, чтобы найти медиану. *Быстрый выбор* похож на быструю сортировку, но этот алгоритм не сортирует подмассив, который не может содержать элемент. Таким образом, мы избавляемся от одного рекурсивного вызова и можем реализовать алгоритм итеративно:

```
template<typename ITEM, typename COMPARATOR> ITEM quickSelect(ITEM* vector,
    int left, int right, int k, COMPARATOR c)
{
    assert(k >= left && k <= right);
    for(int i, j; left < right;)
    {
        partition3(vector, left, right, i, j, c);
        if(k >= i) left = i;
        else if(k <= j) right = j;
        else break;
    }
    return vector[k];
}
```

$E[\text{время выполнения}] = O(n)$ . Маловероятно, что наихудший случай —  $O(n^2)$ . Для случайных векторных элементов неочевиден тот факт, что  $E[\text{время выполнения}] = O(n)$  (см. [7.8]), но можно расширить быструю сортировку с несколькими ключами до *быстрого выбора с несколькими ключами*. То же самое справедливо для множественного выбора (см. *разд. 7.9*. Но ни одно из расширений не реализовано):

```

template<typename VECTOR, typename COMPARATOR> void multikeyQuickselect(
    VECTOR* vector, int left, int right, int k, COMPARATOR c)
{
    assert(k >= left && k <= right);
    for(int d = 0, i, j; right - left >= 1;)
    {
        partition3(vector, left, right, i, j, c);
        if(k <= j) right = j;
        else if (k < i) // Случай равенства j > k > i
        {
            left = j + 1;
            right = i - 1;
            ++c.depth;
        }
        else left = i;
    }
}

```

## 7.9. Множественный выбор

Чтобы отсортировать только первые  $k$  элементов, оптимальное решение с временем  $O(n + k \lg(k))$  состоит в том, чтобы запустить быструю сортировку  $(0, k - 1)$  для результата `quickselect(k)`.

Чаще возникает задача вывести массив только с  $k$  указанными элементами в правильных местах. Нужные элементы задаются с помощью логического массива, и быстрая сортировка не рекурсирует в подмассивы, в которых ничего не надо выводить. Например, таким образом можно вычислить статистические квантили:

```

template<typename ITEM, typename COMPARATOR> void multipleQuickSelect(ITEM*
    vector, bool* selected, int left, int right, COMPARATOR c)
{
    while(right - left > 16)
    {
        int i, j;
        for(i = left; i <= right && !selected[i]; ++i);
        if(i == right + 1) return; // ничего не выбрано
        partition3(vector, left, right, i, j, c);
        if(j - left < right - i) // сначала меньший подмассив
        {
            multipleQuickSelect(vector, selected, left, j, c);
            left = i;
        }
        else
        {
            multipleQuickSelect(vector, selected, i, right, c);
            right = j;
        }
    }
    insertionSort(vector, left, right, c);
}

```

Для любого выбора  $E$  [время выполнения] окажется оптимальным, но зависеть будет от количества и позиций указанных элементов (см. [7.3]). Маловероятно, что наихудший случай —  $O(n^2)$ .

## 7.10. Поиск

*Последовательный поиск* на небольших массивах является самым быстрым алгоритмом, несмотря на время выполнения  $O(n)$ . Кроме того, это очевидный выбор, если элементы не отсортированы. Для отсортированных последовательностей *двоичный поиск* оптимален в худшем случае и выполняется за  $O(\lg(n))$ . Он начинается посередине, и если нужный элемент не равен текущему, идет влево, когда нужный элемент меньше, и вправо, если больше:

```
template<typename ITEM, typename COMPARATOR> int binarySearch(ITEM const*
    vector, int left, int right, ITEM const& key, COMPARATOR const& c)
{
    while(left <= right)
    { // остерегайтесь переполнения при вычислении среднего элемента
        int middle = left + (right - left)/2;
        if(c.isEqual(key, vector[middle])) return middle;
        c(key, vector[middle]) ? right = middle - 1 : left = middle + 1;
    }
    return -1; // элемент не найден
}
```

Опасность переполнения при расчете середины может показаться преувеличенной, но на самом деле вполне возможна. Этот вариант использования маловероятен, и в целом лучше защититься от проблем и не думать о них в будущем.

Перед использованием двоичного (бинарного) поиска нужно знать, отсортирован ли массив. Это легко проверить за время  $O(n)$ :

```
template<typename ITEM, typename COMPARATOR> bool isSorted(ITEM const*
    vector, int left, int right, COMPARATOR const& c)
{
    for(int i = left + 1; i <= right; ++i)
        if(c(vector[i], vector[i - 1])) return false;
    return true;
}
```

*Экспоненциальный поиск* полезен при поиске в неявном неограниченном «массиве». Предположим, что мы перебираем значения 1, затем 2, 4, 8 и т. д. Найдя нужное значение, выполняем бинарный поиск между нужными степенями двойки. Этот алгоритм может отгадать загаданное положительное число, если имеется информация только о результатах сравнения его с другими числами. Время выполнения составляет  $O(\lg(\text{требуемое значение}))$ .

## 7.11. Примечания по реализации

Несмотря на множество представленных алгоритмов, мой выбор — по сравнению со многими другими источниками — на самом деле очень ограничен. Наиболее полезными алгоритмами сортировки являются быстрая сортировка и KSort, и именно они ис-

пользуются далее в книге. Сортировка слиянием, сортировка подсчетом и сортировка перестановками полезны в особых случаях, поэтому совсем сбрасывать их со счетов не стоит.

Интересное наблюдение: чтобы выяснить, как реализовать даже простую быструю сортировку, требуется много исследований. Каждая книга в списке литературы содержит те или иные сведения, которые вам помогут.

То же самое и с алгоритмами выбора — пришлось прошерстить немало литературы, чтобы найти хорошие алгоритмы и их анализ. Для таких базовых задач книг должно быть достаточно.

## 7.12. Комментарии

В отличие от сортировки вставками, другие сортировки с быстродействием  $O(n^2)$  бесполезны. Например:

- ♦ *сортировка выбором* меняет местами минимальный элемент с первым элементом, затем алгоритм выполняется для остальной части массива. Этот алгоритм делает минимально возможное количество перемещений элемента;
- ♦ *пузырьковая сортировка* меняет местами соседние элементы, пока последовательность не станет упорядоченной.

Разделение на три части позволяет решить *проблему национального флага Нидерландов*, которая неявно определяется этим методом. Классический алгоритм Дейкстры требует выполнения меньшего числа инструкций и немного проще, но у него более высокие постоянные коэффициенты при наличии повторяющихся элементов, а при сортировке такое часто бывает (см. [7.7]).

Интересной идеей было использование нескольких опорных точек, что позволяет создать, например, *быструю сортировку с двумя опорными точками*. Она работает немного быстрее, чем обычная быстрая сортировка, но требует в два раза больше кода и устроена сложнее. Анализ алгоритма все еще продолжается, но в настоящее время делается вывод о том, что у этого алгоритма меньше промахов кеша, что вполне компенсирует большее количество инструкций (см. [7.4]).

В C++ STL используется быстрая сортировка с детерминированной медианой из трех элементов, которая при достижении достаточно большой глубины по месту переключается на более медленную и безопасную *пирамидальную сортировку* (см. главу 7. *Приоритетные очереди*). Эта стратегия обеспечивает время выполнения  $O(n \lg(n))$ , но использование случайных опорных точек могло бы улучшить ее. Из-за гарантированного  $E[\text{время выполнения}]$  переключение алгоритма кажется ненужным и не распространяется на другие ситуации, такие как сортировка векторов. Но оно все же необходимо, потому что стандарт C++ требует гарантий производительности в наихудшем случае, что, возможно, чрезмерно связывает нам руки, но выполнимо. *Сортировка по Шеллу* неоптимальна, но эмпирически немного быстрее, чем пирамидальная сортировка (см. [7.7], несмотря на плохую применимость).

Интересным алгоритмом поиска отсортированных числовых элементов является *интерполяционный поиск* (см. [7.9]). Он похож на бинарный поиск, но вместо использования среднего индекса задействуется индекс, основанный на значениях элемента.

$E[\text{время выполнения для элементов } \sim \text{uniform}] = O(\ln(\ln(n)))$ , но вариант использования ограничен, и на практике любой выигрыш по сравнению с бинарным поиском незначителен.

Чтобы убедиться, что массив отсортирован, можно выполнить несколько двоичных поисков случайных элементов. Я не смог найти ссылку на этот метод с вероятностной гарантией правильности.

Еще одна интересная задача — сортировка связанного списка. Поскольку список не поддерживает произвольный доступ, сортировка выполняется для удобства обхода. Для этого можно адаптировать сортировку слиянием без использования дополнительной памяти (см. [7.6]).

## 7.13. Советы по дополнительной подготовке

- ◆ Добавьте проверки диапазона ввода для сортировки подсчетом и KSort.
- ◆ Реализуйте проверку стабильности для стабильных алгоритмов

## 7.14. Список рекомендуемой литературы

- 7.1. Cormen T. H., Leiserson C. E., Rivest R. L., & Stein C. (2009). Introduction to Algorithms. MIT Press.
- 7.2. Flamig B. (1995). Practical Algorithms in C++. Wiley.
- 7.3. Kaligosi K., Mehlhorn K., Munro J. I., & Sanders P. (2005). Towards optimal multiple selection. In Automata, Languages, and Programming (pp. 103–114). Springer.
- 7.4. Kushagra S., López-Ortiz A., Munro J. I., & Qiao A. (2013). Multi-pivot Quicksort: theory and experiments. In Proc. 16th Workshop on Algorithm Engineering and Experiments (ALENEX). SIAM.
- 7.5. Mehlhorn K., & Sanders P., Dietzfelbinger M., & Dementiev R. (2019). Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox. Springer.
- 7.6. Roura S. (1999). Improving mergesort for linked lists. In Algorithms-ESA'99 (pp. 267–276). Springer.
- 7.7. Sedgewick R. (1999). Algorithms in C++, Parts 1–4. Addison-Wesley.
- 7.8. Vallée B., Clément J., Fill J. A., & Flajolet P. (2009). The number of symbol comparisons in Quicksort and Quickselect. In Automata, Languages, and Programming (pp. 750–763). Springer.
- 7.9. Wikipedia (2015). Interpolation search. [https://en.wikipedia.org/wiki/Interpolation\\_search](https://en.wikipedia.org/wiki/Interpolation_search). Accessed November 3, 2015.

## 8. Динамически сортируемые последовательности

### 8.1. Введение

На любом вводном курсе по алгоритмам преподаются сбалансированные бинарные деревья поиска — например, простое, но устаревшее дерево AVL. На более сложных курсах изучают красно-черное дерево, слегка касаясь также вопросов его балансировки. Часто рассматривают и базовое дерево, но бывает, что при этом упускают важные детали реализации. Это подтверждается моим опытом проведения собеседований, на которых я задавал вопросы по основополагающим знаниям.

Мы в этой главе сосредоточимся на более общей концепции динамической отсортированной последовательности, не привязываясь конкретно к реализации дерева. Мы коснемся списков с пропусками — структуры данных, которая в некоторых случаях бывает полезна. В качестве сбалансированного представим декартово дерево (random treap), которое работает столь же эффективно, но намного проще, чем красно-черное дерево. Рандомизированная балансировка также применяется и к префиксным деревьям. Наконец, мы рассмотрим расширение дерева для использования векторных ключей, которое обычно работает лучше, чем обычные префиксные деревья, но не обсуждается ни в одной из известных мне книг.

### 8.2. Требования

*Динамическая отсортированная последовательность* — это набор элементов, который поддерживается в отсортированном состоянии. Для ключа  $x$  эффективно выполняются:

- ♦ *тар-операции поиска, вставки и удаления;*
- ♦ операции поиска максимума и минимума;
- ♦ *итерация по порядку* между любыми двумя элементами;
- ♦ операции поиска *предшествующего* и *последующего*, которые соответственно являются предыдущим максимальным элементом, меньшим чем  $x$ , и следующим минимальным элементом, большим, чем  $x$ . В *инклюзивных* версиях этих операций используются операторы « $\leq$ » и « $\geq$ », которые всегда существуют для вставленного ключа. Итерация и поиск предшествующего лежат в основе реализации поиска последующего. Сочетание их с итерацией позволяет эффективно выполнять *поиск по диапазону*;
- ♦ *объединение* двух последовательностей, таких что ключи в одной меньше ключей в другой;
- ♦ *разделение* последовательности так, что первая содержит элементы  $\leq x$ , а вторая —  $> x$ .

Различные дополнения поддерживают другие операции — такие как поиск  $k$ -го элемента. Объединение и разделение двух последовательностей редко бывает полезно и неэффективно выполняется реализацией на основе свободных списков, потому что необходимо перемещать элементы между свободными списками. В качестве альтернативы может использовать для всех экземпляров один список, но это некорректно. Мар-операции и операции поиска минимума, максимума, предыдущего и последующего обычно выполняются за одинаковое в худшем случае время  $O(\lg(n))$ , где  $n$  — количество элементов в последовательности. Итерация по всем элементам выполняется за  $O(n)$  времени, что быстрее, чем поиск одного элемента. В отличие от сортировки, элементы с неуникальными ключами большинством структур данных не поддерживаются и должны обрабатываться путем добавления ключей с информацией о разрешении конфликтов.

### 8.3. Список с пропусками

*Список с пропусками* (skip list) — это набор связанных списков в отсортированном порядке, где список  $i$  содержит каждый элемент с вероятностью  $p^i$  для некоторой константы  $p$ . Нижний список содержит все элементы, а списки более высокого уровня содержат лишь часть элементов, чтобы ускорить некоторые операции (рис. 8.1).

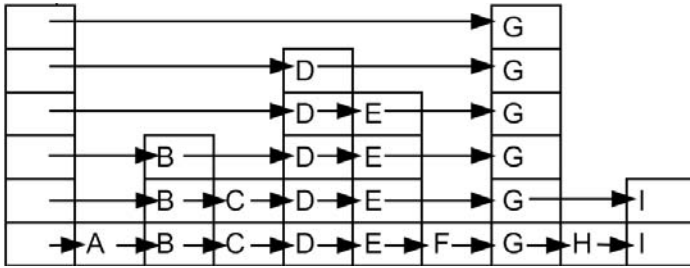


Рис. 8.1. Структура указателя списка пропуска

Количество уровней  $\approx \lg(n)$ , а на практике  $\lg(n) < 32$ , поэтому для простоты реализации высота ограничивается 32 уровнями. Большинство операций поддерживают элементы с неуникальными ключами. Структура памяти списка с пропусками с целыми элементами 0–4 представлена на рис. 8.2 и в следующем коде:

```
template<typename KEY, typename VALUE, typename COMPARATOR =
    DefaultComparator<VALUE> > class SkipList
{
    COMPARATOR c;
    enum{MAX_HEIGHT = 32};
    struct Node
    {
        KEY key;
        VALUE value;
        Node** tower;
        Node(KEY const& theKey, VALUE const& theValue, int height):
            key(theKey), value(theValue), tower(new Node*[height]) {}
        ~Node() {delete[] tower;}
    } * head[MAX_HEIGHT];
```



```

Freelist<Node> f;
int currentLevel;
public:
Skiplist(COMPARATOR const& theC = COMPARATOR()): currentLevel(0), c(theC)
{for(int i = 0; i < MAX_HEIGHT; ++i) head[i] = 0;}
Skiplist(Skiplist const& rhs): currentLevel(0), c(rhs.c)
{ // порядок элементов с неуникальными ключами в копии
  // изменен на обратный
  for(int i = 0; i < MAX_HEIGHT; ++i) head[i] = 0;
  for(Node* node = rhs.head[0]; node; node = node->tower[0])
    insert(node->key, node->value, false);
}
Skiplist& operator=(Skiplist const&rhs){return genericAssign(*this,rhs);}
};

```

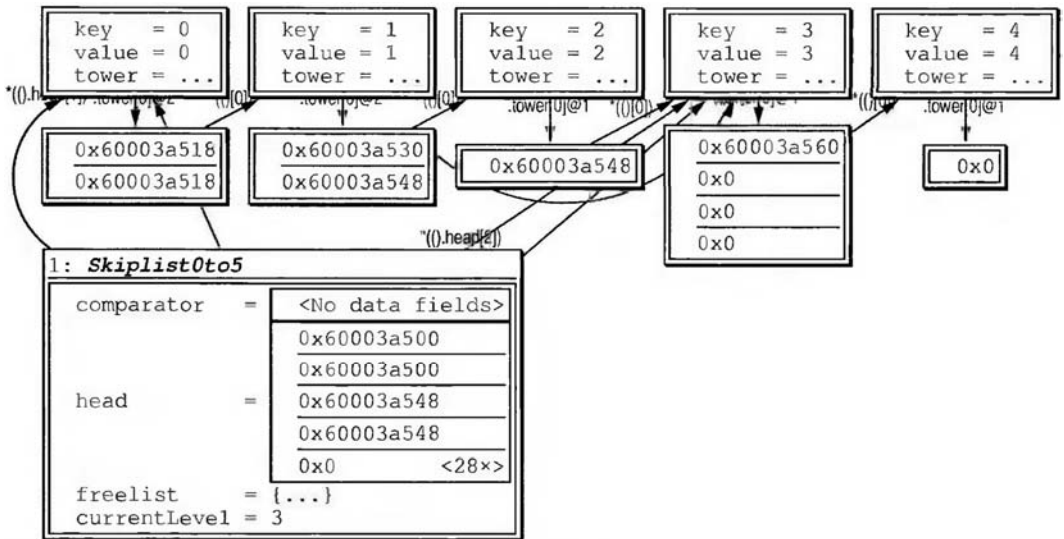


Рис. 8.2. Структура памяти списка с пропусками с целыми элементами 0—4

Алгоритм поиска:

1. Начните с самого верхнего непустого списка.
2. Следуйте по ссылкам, пока не дойдете до списка 0, переходя на один список вниз, когда следующий элемент больше искомого.
3. Выполните линейный поиск в списке 0.

Алгоритм эффективно ищет предыдущий элемент, а по нему выполняются многие другие операции. Вместо указателя узлов возвращаются итераторы для обеспечения безопасности типов:

```

Iterator predecessor(KEY const& key)
{
  Node **tower = head, *pred = 0;
  for(int level = currentLevel; level >= 0; --level)

```

```

    for(Node* node; (node = tower[level]) && c(node->key, key);
        tower = node->tower) pred = node;
    return Iterator(pred);
}
Iterator inclusiveSuccessor(KEY const& key)
{ // next(pred) = inc succ.
    Iterator pred = predecessor(key);
    assert(pred == end() || c(pred->key, key));
    return pred == end() ? begin() : Iterator(pred->tower[0]);
}
Iterator findNode(KEY const& key)
{ // общий шаблон - возвращается указатель для использования
  // в других операциях
    Iterator node = inclusiveSuccessor(key); // совпадение считается,
                                           // если не превышает значение ключа
    assert(node == end() || !c(node->key, key));
    return node == end() || c(key, node->key) ? end() : node;
}
VALUE* find(KEY const& key)
{
    Iterator result = findNode(key);
    return result == end() ? 0 : &result->value;
}
Iterator successor(KEY const& key)
{ // inclusiveSuccessor с неуникальными ключами, иначе перебираем равные ключи
    Node* node = inclusiveSuccessor(key)->current;
    while(node && !c(node->key, key)) node = node->tower[0];
    return Iterator(node);
}

```

Вставка выполняется с учетом того, что списки более высокого уровня содержат меньше узлов, чем списки нижнего уровня:

1. Сгенерируйте высоту элемента как геометрическое распределение  $(1 - p)$ .
2. Начните с самого высокого уровня.
3. Для любого уровня:
4. Найдите место для вставки узла, чтобы сохранить сортировку.
5. Свяжите предыдущий узел со вставленным узлом, а вставленный — со следующим.
6. Перейдите вниз на один уровень.
7. При желании верните дескриптор вставленного узла.

$E[\text{количество указателей на узел}] = \frac{1}{1-p}$ .  $E[\text{количество сравнений для поиска}] \leq$

$\leq \frac{\log_{1/p}(n)}{p} + \frac{1}{1-p}$ , что для больших  $n$  минимизируется при  $p = 1/e$  (см. [8.11]). Это

время выполнения для большинства операций.

```

Iterator insert(KEY const& key, VALUE const& value, bool unique = true)
{
    if(unique)

```

```

{ // для уникальных ключей выполняется проверка, существует ли он
  Iterator result = findNode(key);
  if(result != end())
  {
    result->value = value;
    return result;
  }
} // уровень = высота - 1
int newLevel = min<int>(MAX_HEIGHT, GlobalRNG().geometric(0.632)) - 1;
Node* newNode = new(f.allocate())Node(key, value, newLevel + 1);
if(currentLevel < newLevel) currentLevel = newLevel;
Node** tower = head;
for(int level = currentLevel; level >= 0; --level)
{
  for(Node* node; (node = tower[level]) &&
    c(node->key, key); tower = node->tower);
  if(level <= newLevel)
  { // пересоздание указателей
    newNode->tower[level] = tower[level];
    tower[level] = newNode;
  }
}
return Iterator(newNode);
}

```

Удаление элемента выполняется из всех уровней по тому же алгоритму поиска:

```

void remove(KEY const& key)
{ // если есть неуникальные элементы, удален будет первый найденный
  Node **prevTower = head, *result = 0;
  for(int level = currentLevel; level >= 0; --level)
  { // идем вниз, если node->key < key (когда result == 0),
    // иначе продолжаем двигаться вправо
    for(Node* node; (node = prevTower[level]) && !c(key, node->key);
      prevTower = node->tower)
      // значение найдено, если попали в запомненный узел
      // или выполнено равенство
      if(node == result || (!result && !c(node->key, key)))
      { // отвязываем узел от текущего уровня
        prevTower[level] = node->tower[level];
        node->tower[level] = 0;
        if(!head[currentLevel]--currentLevel; // если удален верхний узел
          result = node; // запоминаем его
          break; // спускаемся на уровень вниз
        }
      }
  }
  if(result) f.remove(result);
}

```

Минимум — это первый элемент, его поиск выполняется за время  $O(1)$ . Максимум находится в конце нижнего списка, и чтобы добраться до него, нужно использовать указатели более высокого уровня, многократно переходя к крайнему правому ненулевому узлу, а затем вниз:

```

Iterator findMin(){return Iterator(head[0]);}
Iterator findMax()
{
    Node *result = 0, **tower = head;
    for(int level = currentLevel; level >= 0; --level)
        for(Node* node; node = tower[level]; tower = node->tower)
            result = node;
    return Iterator(result);
}

```

Нижний список легко перебирать, потому что это просто связанный список:

```

class Iterator
{
    Node* current;
    Iterator(Node* node): current(node){}
    friend SkipList;
public:
    Iterator& operator++()
    {
        assert(current);
        current = current->tower[0];
        return *this;
    }
    Node& operator*()
    {
        assert(current);
        return *current;
    }
    Node* operator->()
    {
        assert(current);
        return current;
    }
    bool operator==(Iterator const& rhs) const
    {return current == rhs.current;}
};
Iterator begin(){return Iterator(findMin());}
Iterator end(){return Iterator(0);}

```

$E[\text{время выполнения}] = O(\lg(n) + k)$  для поиска по диапазону из  $k$  элементов, и  $E[\text{время выполнения для других операций, помимо минимума}] = O(\lg(n))$ . Список с пропусками немного менее эффективен, чем динамическая отсортированная последовательность на основе дерева, из-за узлов переменной длины и больших постоянных коэффициентов, зато более расширяем из-за:

- ◆ отсортированности нижнего уровня;
- ◆ поддержки неуникальных элементов, что позволяет использовать его как мульти-отображение или приоритетную очередь;
- ◆ поддержки разных дополнений.

Маловероятное время выполнения большинства операций в наихудшем случае равно  $O(n)$ .

## 8.4. Декартово дерево

Высота декартова дерева (treap) равна максимальному количеству предков любого узла. Упорядоченное бинарное дерево высотой  $h$  реализует операции динамической отсортированной последовательности за время  $O(h)$ . Первый элемент становится одноузловым деревом высотой 0. Любой последующий элемент рекурсивно заменяет текущий, если равен ему, вставляется в левый дочерний элемент, если меньше, и в правый дочерний элемент, если больше. Когда элементы вставляются в отсортированном или случайном порядке,  $h$  равно  $n$  или  $O(\lg(n))$  соответственно (см. [8/15]). Дерево *сбалансировано*, если  $h = O(\lg(n))$ , что обеспечивает время выполнения  $O(\lg(n))$  для большинства операций.

Декартово дерево — простейшее сбалансированное бинарное дерево. Вновь вставленный узел получает случайный приоритет. Дерево структурировано таким образом, что приоритет узла  $\leq$  приоритета его родителя. Получается *порядок в куче* (см. главу 10. *Приоритетные очереди*), который поддерживается при вставке и удалении (рис. 8.3).

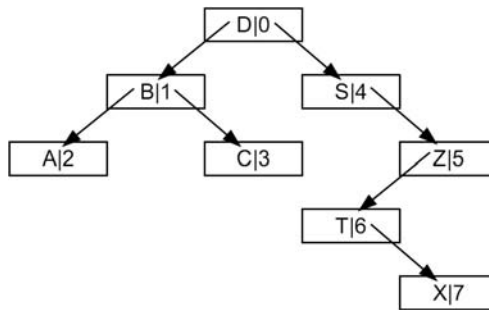


Рис. 8.3. Элементы дерева (буквы) и приоритеты (цифры)

Узлы могут также хранить указатели на родителя и счетчики дочерних узлов (обсуждается позже в этой главе). Структура памяти декартова дерева с целочисленными элементами 0–9 представлена на рис. 8.4.

Конструктор копирования работает рекурсивно. Он копирует корень, а затем каждого потомка за время  $O(n)$ . Это общий шаблон любой реализации дерева:

```

template<typename KEY, typename VALUE, typename COMPARATOR =
    DefaultComparator<KEY> > class Treap
{
    COMPARATOR c;
    struct Node
    {
        KEY key;
        VALUE value;
        Node *left, *right, *parent;
        unsigned int priority, nodeCount;
        Node(KEY const& theKey, VALUE const& theValue): key(theKey),
            value(theValue), left(0), right(0), parent(0),
            priority(GlobalRNG().next()), nodeCount(1) {}
    } * root;
    Freelist<Node> f;

```

```

Node* constructFrom(Node* node)
{
    Node* tree = 0;
    if(node)
    {
        tree = new(f.allocate())Node(node->key, node->value);
        tree->priority = node->priority;
        tree->nodeCount = node->nodeCount;
        tree->left = constructFrom(node->left);
        if(tree->left) tree->left->parent = tree;
        tree->right = constructFrom(node->right);
        if(tree->right) tree->right->parent = tree;
    }
    return tree;
}

public:
    typedef Node NodeType;
    unsigned int getSize(){return root ? root->nodeCount : 0;}
    Treap(COMPATOR const& theC = COMPATOR()): root(0), c(theC){}
    Treap(Treap const& other): c(other.c)
    {
        root = constructFrom(other.root);
        if(root) root->parent = 0;
    }
    Treap& operator=(Treap const& rhs){return genericAssign(*this, rhs);}

```

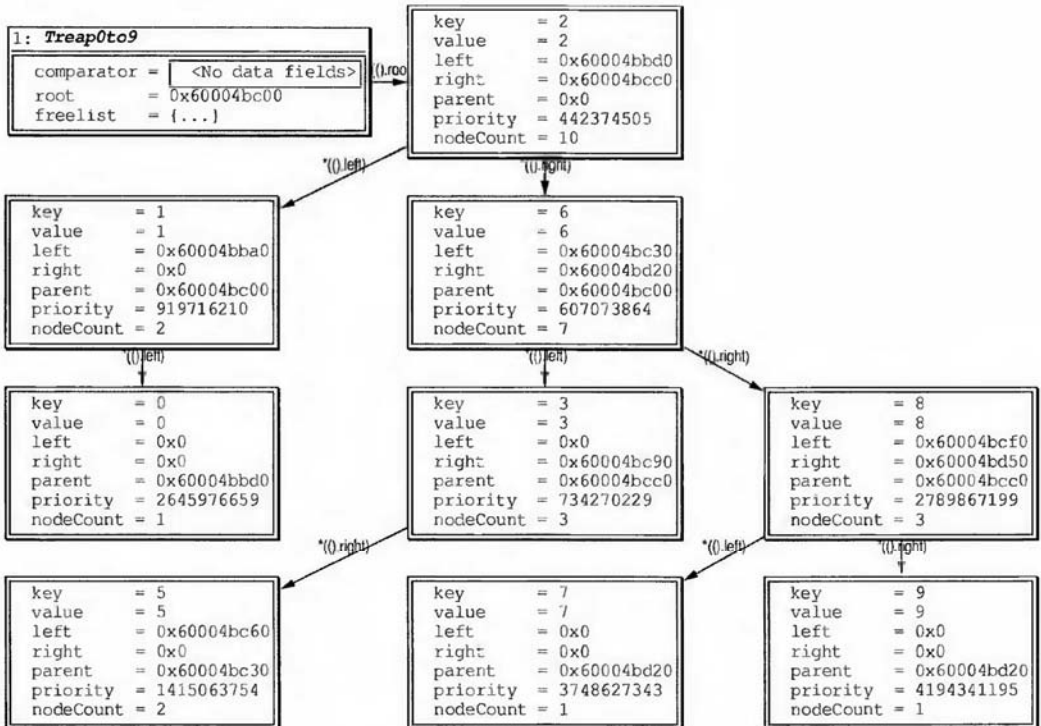


Рис. 8.4. Структура памяти декартова дерева (treap) с целочисленными элементами 0–9

Алгоритм поиска:

1. Начинаем с корня.
2. Пока не достигнем желаемого или нулевого узла:
3. Переход влево, если ключ  $<$  текущего ключа узла, в противном случае вправо.
4. Возвращаем указатель на узел, содержащий элемент.

Реализация поиска идентична списку с пропусками:

```
Node* findNode(KEY const& key)
{
    Node* node = root;
    while (node && !c.isEqual(key, node->key)) node =
        c(key, node->key) ? node->left : node->right;
    return node;
}
```

*Вращение* — это такое преобразование дерева, после которого получается порядок кучи с сохранением при этом отсортированного порядка:

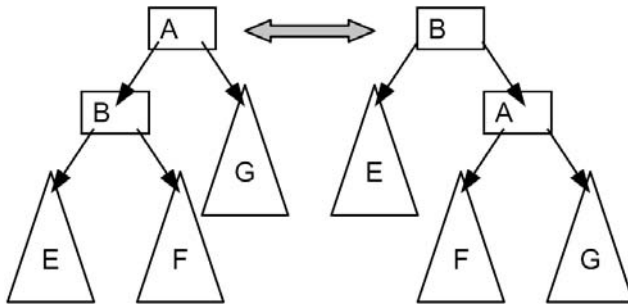


Рис. 8.5. Логика перестановки указателя вращения дерева

Вращение вправо:

1. Новый корень = левый дочерний элемент **В** корня поддерева.
2. Левый дочерний элемент корня = правый дочерний элемент нового корня **В**.
3. Правый потомок нового корня = корень **А**.
4. Родительские указатели **В** и **В** сбрасываются.
5. Количество узлов **В** = количеству узлов **А**, и последнее пересчитывается.
6. Можно добавить что-то еще, если это необходимо.

Левое вращение работает симметрично. И то и другое выполняется за время  $O(1)$ , если используются несложные дополнения. У сбалансированного дерева не может быть неуникальных ключей, потому что тогда вращения нарушают условие « $<$ ». Помощники вращения реализуются в общем случае для повторного использования в ТДД (см. разд. 8.10):

```
template<typename NODE> struct TreapGeneric
{
    static void rotateHelper(NODE* node, NODE* goingUp, NODE*& movedChild)
```

```

{
    if(movedChild) movedChild->parent = node;
    movedChild = node;
    goingUp->nodeCount = node->nodeCount;
    goingUp->parent = node->parent;
    node->parent = goingUp;
    node->nodeCount = 1 + (node->left ? node->left->nodeCount: 0) +
        (node->right ? node->right->nodeCount: 0);
}
};

```

Вращения являются функциями-членами:

```

Node* rotateRight(Node* node)
{
    Node *goingUp = node->left, *&movedChild = goingUp->right;
    node->left = child;
    TreapGeneric<Node>::rotateHelper(node, goingUp, movedChild);
    return goingUp;
}

Node* rotateLeft(Node* node)
{
    Node *goingUp = node->right, *&movedChild = goingUp->left;
    node->right = movedChild;
    TreapGeneric<Node>::rotateHelper(node, goingUp, movedChild);
    return goingUp;
}

```

Узел сначала вставляется, как если бы дерево было несбалансированным, т. е. продвигается по ветвям, пока не наткнется на лист, и становится его потомком. Затем он вращается вверх до тех пор, пока его приоритет не будет соответствовать порядку в куче — правый поворот поднимает левый дочерний элемент, и наоборот. Вращения сохраняют порядок кучи других элементов:

```

Node* insertNode(Node* newNode, Node* node)
{
    if(!node) return newNode;
    bool goLeft = c(newNode->key, node->key);
    Node*& chosenChild = goLeft ? node->left : node->right;
    chosenChild = insertNode(newNode, chosenChild);
    chosenChild->parent = node;
    ++node->nodeCount;
    if(chosenChild->priority < node->priority)
        node = goLeft ? rotateRight(node) : rotateLeft(node);
    return node;
}

NodeType* insert(KEY const& key, VALUE const& value)
{
    Node* node = findNode(key);
    if(node) node->value = value;
    else
    {
        node = new(f.allocate())Node(key, value);
    }
}

```



```

    root = insertNode(node, root);
}
return node;
}

```

Результирующее дерево уникально, если его приоритеты уникальны. Использование случайных приоритетов эквивалентно вставке в несбалансированное дерево в случайном порядке, поэтому  $E[h] = O(\lg(n))$ , независимо от предыдущих вставок и удалений. Для малой константы  $c$   $\Pr(h > 1 + 2c \ln(n)) < 2(n/e)^{-c \ln(c/e)}$  эффективно обеспечивает баланс при не слишком малых  $n$ . Наихудший маловероятный случай  $h = O(n)$ .  $E[\text{количество оборотов для вставки}] = O(1)$  (см. [8.15]), что важно при использовании дополнений.

При удалении используется вращение на лист, что является самым простым вариантом с дополнениями (обсуждается позже в этой главе):

1. Найдите узел, соответствующий  $x$ .
2. Вращайте его на лист или узел с одним дочерним элементом, подняв дочерние узлы с более низким приоритетом.
3. Удалите его.

```

Node* removeFound(Node* node)
{
    Node *left = node->left, *right = node->right;
    if(left && right)
    {
        bool goRight = left->priority < right->priority;
        node = goRight ? rotateRight(node) : rotateLeft(node);
        Node* child = goRight ? node->right : node->left;
        child = removeFound(child);
        if(child) child->parent = node;
        --node->nodeCount;
    }
    else
    {
        f.remove(node);
        node = left ? left : right;
    }
    return node;
}

void remove(KEY const& key)
{
    Node* node = findNode(key);
    if(node)
    {
        Node* parent = node->parent;
        bool wasLeft = parent && node == parent->left;
        node = removeFound(node);
        if(node) node->parent = parent;
        (parent ? (wasLeft ? parent->left : parent->right) : root) = node;
        for(;; parent; parent = parent->parent) --parent->nodeCount;
    }
}

```

Время выполнения равно  $O(h)$ .

Минимальное значение — это самый левый узел, а максимум — самый правый. Их поиск реализован в `TreapGeneric`:

```
static NODE* findMin(NODE* root)
{
    NODE* node = root;
    if(node) while(node->left) node = node->left;
    return node;
}

static NODE* findMax(NODE* root)
{
    NODE* node = root;
    if(node) while(node->right) node = node->right;
    return node;
}
```

Применительно к декартову дереву:

```
NodeType* findMin() {return TreapGeneric<Node>::findMin(root);}
NodeType* findMax() {return TreapGeneric<Node>::findMax(root);}
```

Дополнение подсчета узлов позволяет найти  $n$ -й элемент, такой как медиана, и реализовано в `TreapGeneric`. Пусть  $n_{lc}$  = количество левых дочерних элементов текущего узла. Тогда:

1. Если  $n = n_{lc}$ , возвращается корень.
2. В противном случае, если  $n < n_{lc}$ , двигаемся налево.
3. В противном случае идем направо и уменьшаем  $n$  на  $n_{lc} + 1$ .

```
static NODE* nthElement(int n, NODE* root)
{
    assert(n >= 0 && root && n < root->nodeCount);
    NODE* node = root;
    for(;;)
    {
        unsigned int lc = node->left ? node->left->nodeCount : 0;
        if(n == lc) break;
        if(n < lc) node = node->left;
        else
        {
            n -= lc + 1;
            node = node->right;
        }
    }
    return node;
}
```

Применительно к декартову дереву:

```
NodeType* nthElement(int n)
{return TreapGeneric<Node>::nthElement(n, root);}
```

Поиск предыдущего и последующего элементов реализован аналогично, и у них симметричная логика. Предыдущий элемент находится максимально близко к  $x$  слева:

1.  $Pred = 0$ .
2. Пока не дошли до листа:
3. Если  $node < x$  и  $pred = node$ , идем вправо.
4. В противном случае идем влево.
5. Возврат  $pred$ .

```

NodeType* predecessor (KEY const& key)
{
    Node* pred = 0;
    for (Node* node = root; node;)
        if (c(node->key, key)) // поиск предыдущего элемента
        {
            pred = node;
            node = node->right;
        }
        else node = node->left;
    return pred;
}

NodeType* inclusivePredecessor (KEY const& key)
{
    Node* node = findNode(key);
    return node ? node : predecessor(key);
}

NodeType* successor (KEY const& key)
{
    Node* succ = 0;
    for (Node* node = root; node;)
        if (c(key, node->key)) // поиск следующего элемента
        {
            succ = node;
            node = node->left;
        }
        else node = node->right;
    return succ;
}

NodeType* inclusiveSuccessor (KEY const& key)
{
    Node* node = findNode(key);
    return node ? node : successor(key);
}

```

## 8.5. Итераторы деревьев

Несколько способов итерации дерева по порядку:

- ◆ с помощью указателей на родителей из дочерних элементов, что упрощает операции с итераторами и позволяет выполнять двунаправленную итерацию. Но нужен дополнительный указатель на узел, и его надо обновлять во время вставок и удалений. Все операции могут возвращать и принимать итераторы, что удобно для поль-

зователей, но это не обобщается на структуры, не имеющие итераторов, — такие как многомерное дерево (см. главу 18. *Вычислительная геометрия*). Это единственный способ создать итераторы как легковесные объекты, и он реализован здесь и в C++ STL;

- ◆ передать функтор в поиск по диапазону. Например, при суммировании всех значений функтор имеет указатель на текущее значение суммы и обновляет ее. Это наиболее эффективно и обобщается на многомерное дерево, но создание функтора неудобно для пользователей;
- ◆ выполнить поиск по диапазону, как описано ранее, с предопределенным функтором, который создает вектор найденных элементов, — способ более удобный для пользователей, но требующий памяти для вектора.

Основная идея итерации с родительскими указателями заключается в том, что итератор увеличивается, когда правый дочерний элемент равен null, до тех пор, пока не вернется из левого дочернего элемента. Обратная итерация симметрична. По соглашению итерация с нулевого узла не разрешена, хотя может иметь смысл запомнить предыдущий узел, если он есть, и разрешить обратную итерацию к нему:

```
template<typename NODE> class TreeIterator
{
    NODE* current;
public:
    TreeIterator(NODE* node){current = node;}
    TreeIterator& operator++()
    {
        assert(current);
        if(current->right)
        { // если правый потомок идет туда, двигаемся максимально влево
            current = current->right;
            while(current->left) current = current->left;
        }
        else
        { // берем родителя, если пришли из левого дочернего элемента,
          // в противном случае продолжаем подниматься вверх
            while(current->parent && current != current->parent->left)
                current = current->parent;
            current = current->parent;
        }
        return *this;
    }
    TreeIterator& operator--()
    {
        assert(current);
        if(current->left)
        { // если левый потомок идет туда, двигаемся максимально вправо
            current = current->left;
            while(current->right) current = current->right;
        }
        else
        { // берем родителя, если пришли из правого дочернего элемента,
          // в противном случае продолжаем подниматься вверх
```

```

        while(current->parent && current != current->parent->right)
            current = current->parent;
        current = current->parent;
    }
    return *this;
}
NODE& operator*() {assert(current); return *current;}
NODE* operator->() {assert(current); return current;}
bool operator==(TreeIterator const& rhs) const
    {return current != rhs.current;}
};

```

Применительно к декартову дереву:

```

typedef TreeIterator<Node> Iterator;
Iterator begin() {return Iterator(findMin());}
Iterator end() {return Iterator(0);}
Iterator rBegin() {return Iterator(findMax());}
Iterator rEnd() {return Iterator(0);}

```

## 8.6. Дополнения и варианты API

Деревья часто расширяют возможностью сохранения числа узлов в каждом поддереве и указателями на родителя указателей. Но бывают и другие полезные расширения.

API динамической отсортированной последовательности может быть таким:

- ◆ отображение (map) и реализация других вариантов на его основе;
- ◆ мультиотображение (multimap), допускающее равные элементы. В этом отображении тип элемента представляет собой вектор элементов;
- ◆ *множество* с ключами, но без значений, — отображение, в котором тип элемента игнорируется логическим или пустым из-за нехватки памяти;
- ◆ *мультимножество*, где ключи не уникальны, — отображение, где элемент представляет собой количество ключей.

Список с пропусками напрямую поддерживает несколько вариантов. В STL реализованы отображения, похожие на наборы, где в качестве элементов используются пары ключ-значение, но для элементов требуется конструктор по умолчанию, поскольку поиск создает пару ключ-значение. Помимо использования вектора ключей, можно сохранить глобальный счетчик вставок и использовать его в качестве вторичного приоритета. Это проще, если хранить информацию о количестве узлов.

Операции с отображениями в хеш-таблице (см. главу 9. *Хеширование*) работают быстрее. Для выбора предыдущего/следующего или итерации по порядку нужна динамическая отсортированная последовательность. В целом вопрос о том, какую структуру данных и когда использовать, — один из моих любимых вопросов на собеседовании. Декартово дерево немного более эффективно с точки зрения скорости и использования памяти, поэтому если стандартных расширений достаточно, лучше выбрать его. А простота списка с пропусками быстрее выполняет итерацию и позволяет добавлять расширения.

## 8.7. Векторные ключи

Когда ключи представляют собой массивы переменной длины, содержащие до  $k$  объектов, сравнимых за время  $O(1)$ , сравнение ключей занимает время  $O(k)$ , что дает  $O(k \lg(n))$  операций. Кроме того,  $E[\text{lcp} \text{ двух случайных строк в алфавите размера } a \text{ из набора } n] = \log_a(n)$ , так что ожидаемая производительность обычно лучше.

Основные операции расширения уже поддерживаются:

- ♦ *поиск по префиксу* — по заданной длине найти все элементы, у которых  $\text{lcp}(x, \text{ключ элемента}) \geq \text{длины}$ . Итерация от предшественника префикса дает нужный результат;
- ♦ *самое длинное совпадение* — по ключу  $x$  можно найти элемент, максимизирующий  $\text{lcp}(x, \text{ключ элемента})$ . Результатом является предыдущий или следующий элемент от  $x$ .

На практике производительность этих операций достаточно хороша, но ее можно улучшить. Идея состоит в том, чтобы найти  $\text{lcp}$  и сравнить  $\text{key}[\text{lcp}]$ .  $\text{lcp}$  любого ключа  $s - \infty$  или  $\infty$  равно 0. Для любого векторного ключа  $x \leq y \leq z$ :

- ♦  $\text{lcp}(x, y) \geq \text{lcp}(x, z)$ , с равенством, только если  $y = z$ ;
- ♦  $\text{lcp}(x, z) = \min(\text{lcp}(x, y), \text{lcp}(y, z))$ .

## 8.8. Расширение LCP для деревьев

В каждом узле можно хранить значение  $\text{lcp}$  между ним и его соседями среди узлов на пути поиска (см. [8.6, 8.5]). Используйте формат `unsigned short` для экономии места. У корня с бесконечным числом ключей  $\text{lcp}$  равны 0 (рис. 8.6). Структура памяти LCP декартова дерева (LCPTreap) с некоторыми словами представлена на рис. 8.7.

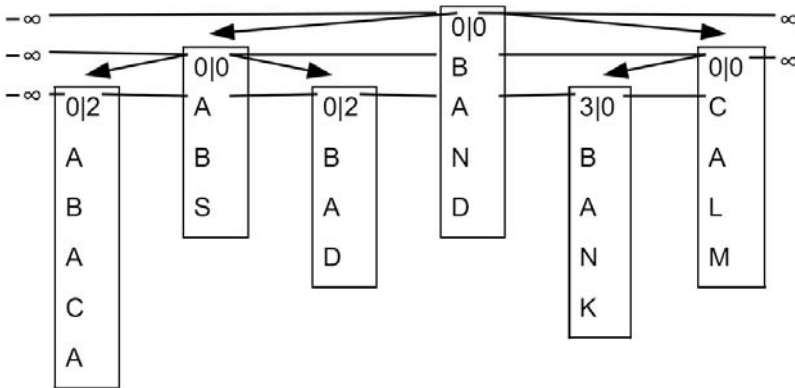


Рис. 8.6.  $\text{lcp}$  между узлами и их предшествующими ( $\text{pred}$ )/последующими ( $\text{succ}$ ).

Например, для слова «bad» в списке предшествующих есть только «abs», поэтому  $\text{pred lcp} = 0$ , а в списке последующих есть только «band», поэтому  $\text{succ lcp} = 2$

```
template<typename KEY, typename VALUE, typename INDEXED_COMPARATOR =
    LexicographicComparator<KEY> > class LCPTreap
{
    INDEXED_COMPARATOR c;
```

```

struct Node
{
    KEY key;
    VALUE value;
    Node *left, *right, *parent;
    unsigned int priority, nodeCount;
    unsigned short predLcp, succLcp;
    Node(KEY const& theKey, VALUE const& theValue): key(theKey),
        value(theValue), left(0), right(0), priority(GlobalRNG().next()),
        predLcp(0), succLcp(0), parent(0), nodeCount(1) {}
}* root;
Freelist<Node> f;
Node* constructFrom(Node* node)
{
    Node* tree = 0;
    if(node)
    {
        tree = new(f.allocate())Node(node->key, node->value);
        tree->priority = node->priority;
        tree->nodeCount = node->nodeCount;
        tree->predLcp = node->predLcp;
        tree->succLcp = node->succLcp;
        tree->left = constructFrom(node->left);
        if(tree->left) tree->left->parent = tree;
        tree->right = constructFrom(node->right);
        if(tree->right) tree->right->parent = tree;
    }
    return tree;
}
public:
    typedef Node NodeType;
    LCPTreap(INDEXED_COMPARATOR theC = INDEXED_COMPARATOR()):root(0), c(theC){}
    LCPTreap(LCPTreap const& other): c(other.c)
    {root = constructFrom(other.root);}
    LCPTreap& operator=(LCPTreap const&rhs){return genericAssign(*this,rhs);}
    unsigned int getSize(){return root ? root->nodeCount : 0;}
};

```

При сравнении  $x$  с текущим узлом  $c$  с предком-предыдущим  $p$  и предком-следующим  $s$ :

- ◆  $p < x < s$ , и пусть  $\max(\text{lcp}(p, x), \text{lcp}(s, x)) = m$ ;
- ◆  $p < c < s$ , и пусть  $\text{lcp}(p, c) = l$  и  $\text{lcp}(s, c) = r$ .

Если  $c \neq x$ ,  $\text{lcp}(c, x)$  является первым индексом, при котором  $c \neq x$ , и вы можете найти его на основе следующей логики:

- ◆ если  $\text{lcp}(p, x) = m$ :
  - если  $l < m$ ,  $\text{lcp}(c, x) = \min(\text{lcp}(c, p), \text{lcp}(p, x)) = l$  — можно сравнить  $x[l]$  с  $c[l]$ ;
  - в противном случае  $\text{lcp}(c, x) \geq m$  — необходимо продолжать сравнение;
- ◆ если  $\text{lcp}(s, x) = m$ :
  - если  $r < m$ ,  $\text{lcp}(c, x) = \min(\text{lcp}(s, p), \text{lcp}(p, x)) = r$  — можно сравнить  $x[r]$  с  $c[r]$ ;
  - в противном случае  $\text{lcp}(c, x) \geq m$  — необходимо продолжать сравнение.

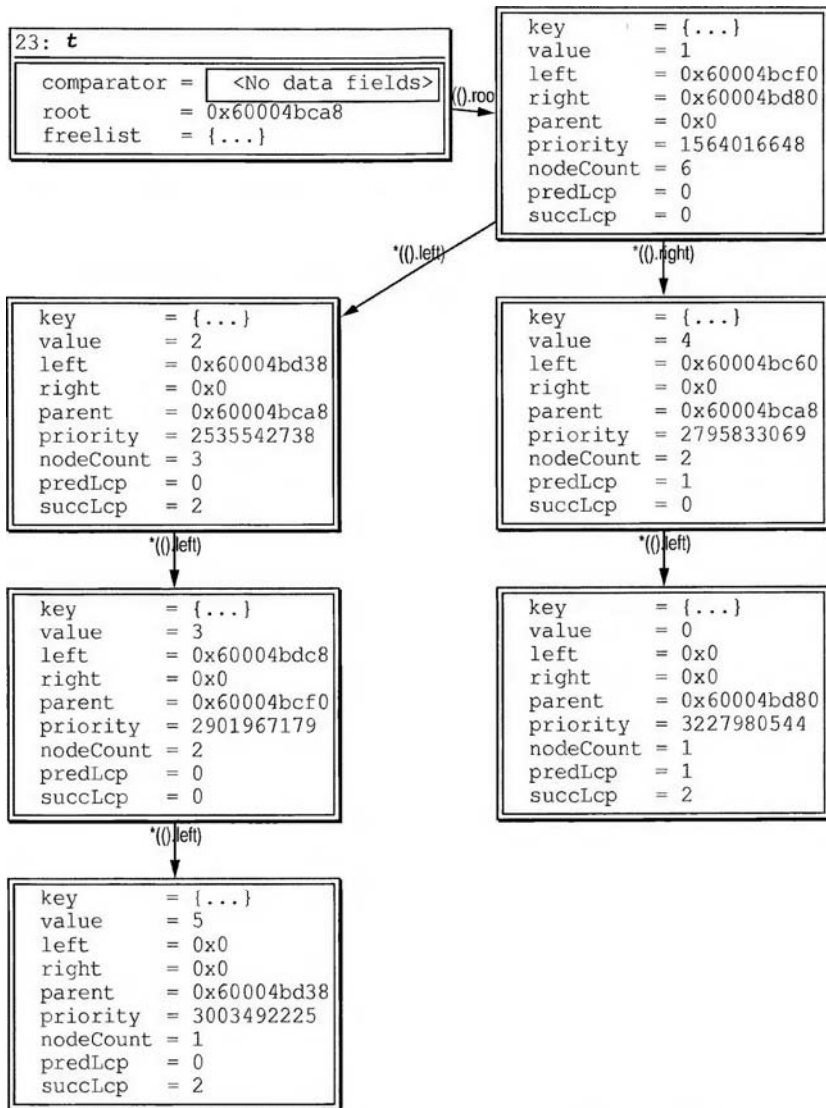


Рис. 8.7. Структура памяти LCPTreap с некоторыми словами

Чтобы решить, принадлежит ли  $m$  списку  $p$  или  $s$ , сохраните  $\text{lcp}(p, x)$  в  $\text{predM}$ . Функция  $\text{findLcp}$  вычисляет  $\text{lcp}(c, x)$  и обновляет  $m$ , чтобы включить текущий узел в качестве предка-предыдущего:

```

int findLCP(KEY const& key, Node* node, int predM, int& m)
{
    int lcp = predM == m ? node->predLcp : node->succLcp; // получаем 1
                                                         // или r

    if(lcp >= m)
    {
        while(m < c.getSize(key) &&
              c.isEqual(key, node->key, m)) ++m;
    }
}

```



```

        lcp = m;
    }
    return lcp;
}

```

Если  $c$  и  $x$  имеют одинаковую длину  $= \text{lcp}$ , то  $c = x$ . Значение  $m \leq k$  и никогда не уменьшается, поэтому операции отображения выполняются за  $O(k + \lg(n))$ . Двигаясь вправо, мы присваиваем  $\text{predM} = \text{lcp}(c, x)$ , потому что  $c$  становится равным  $p$ . `find` работает так же, как у декартова дерева:

```

Node* findNode(KEY const& key)
{
    Node* node = root;
    int m = 0, predM = 0;
    while(node)
    {
        int lcp = findLCP(key, node, predM, m);
        if(c.getSize(key) == lcp && c.getSize(node->key) == lcp) break;
        if(c(key, node->key, lcp)) node = node->left;
        else
        {
            node = node->right;
            predM = lcp;
        }
    }
    return node;
}

```

Вращения поддерживают работу с информацией о  $\text{lcp}$ . Для правого вращения левый дочерний элемент узла становится его предком-предыдущим. Поэтому присваиваем  $\text{predLcp} = \text{succLcp}$  левого дочернего элемента — узел является преемником своего предка. Левый дочерний элемент теряет своего предка-предыдущего, поэтому ему присваиваем  $\text{succLcp} = \text{lcp}(\text{it}, \text{предок-следующий узла}) = \text{минимальному из двух значений succLcp}$  (по равенству треугольника). Левое вращение работает симметрично, и оба они выполняются за  $O(1)$ :

```

Node* rotateRight(Node* node)
{
    Node *goingUp = node->left, *&movedChild = goingUp->right;
    node->predLcp = goingUp->succLcp;
    goingUp->succLcp = min(node->succLcp, goingUp->succLcp);
    node->left = movedChild;
    TreapGeneric<Node>::rotateHelper(node, goingUp, movedChild);
    return goingUp;
}

Node* rotateLeft(Node* node)
{
    Node *goingUp = node->right, *&movedChild = goingUp->left;
    node->succLcp = goingUp->predLcp;
    goingUp->predLcp = min(node->predLcp, goingUp->predLcp);
    node->right = movedChild;
    TreapGeneric<Node>::rotateHelper(node, goingUp, movedChild);
    return goingUp;
}

```

Вставка — это сочетание поиска и вставки декартова дерева. Операция создает узел и обновляет его значением `predLcp/succLcp` при движении соответственно влево/вправо. Во время вставки `predLcp = predM`:

```
Node* insertNode(Node* newNode, Node* node, int m = 0)
{
    if(!node) return newNode;
    int lcp = findLCP(newNode->key, node, newNode->predLcp, m);
    bool goLeft = c(newNode->key, node->key, lcp);
    (goLeft ? newNode->succLcp : newNode->predLcp) = lcp;
    Node*& chosenChild = goLeft ? node->left : node->right;
    chosenChild = insertNode(newNode, chosenChild, m);
    chosenChild->parent = node;
    ++node->nodeCount;
    if(chosenChild->priority < node->priority)
        node = goLeft ? rotateRight(node) : rotateLeft(node);
    return node;
}
```

Операция *вставки* аналогична декартову дереву. То же самое справедливо для операций `removeFound`, `remove`, `min`, `max`, `nthElement`, функций итератора и `inclusiveSuccessor`.

Предыдущий и последующий элементы отличаются только тем, что они находят `lcp` и сравнивают `key[lcp]`. Время выполнения такое же, как и для операций с отображениями:

```
NodeType* predecessor(KEY const& key)
{
    int m = 0, predM = 0;
    Node* pred = 0;
    for(Node* node = root; node;)
    {
        int lcp = findLCP(key, node, predM, m);
        if(c(node->key, key, lcp)) // поиск предыдущего члена множества
        {
            pred = node;
            node = node->right;
            predM = lcp;
        }
        else node = node->left;
    }
    return pred;
}

NodeType* successor(KEY const& key)
{
    int m = 0, predM = 0;
    Node* succ = 0;
    for(Node* node = root; node;)
    {
        int lcp = findLCP(key, node, predM, m);
        if(c(key, node->key, lcp)) // поиск последующего члена множества
        {
            succ = node;

```

```

        node = node->left;
        predM = lcp;
    }
    else node = node->right;
}
return succ;
}

```

*Последующий по префиксу* аналогичен обычному последующему, но он пропускает элементы с большим префиксом. Время выполнения такое же, как у операций с отображениями:

```

NodeType* prefixSuccessor(KEY const& prefix)
{
    int m = 0, predM = 0;
    for(Node* node = root; node;)
    {
        int lcp = findLCP(prefix, node, predM, m);
        if(c.getSize(prefix) == lcp || !c(prefix, node->key, lcp))
        {
            node = node->right;
            predM = lcp;
        }
        else if(!node->left) return node;
        else node = node->left;
    }
    return 0;
}

```

Расширение lcp полезно, но его часто можно избежать ради повышения эффективности:

- ♦ реализуйте отображение с целочисленными парами ключей, закодировав пару в `int` или `long long` и используя ее в качестве ключа. Нужна осторожность — например, при заданных  $0 \leq a < m$  и  $0 \leq b < n$  с  $m > n$  отображение  $x = a + bm$  работает, а  $x = b + an$  — нет, потому что  $(b + an) \% n \neq b$ , если  $b > n$ . Во время собеседований кандидаты часто предпочитают составлять строковые ключи из пар, потому что они хорошо работают с API на разных языках (включая хеш-таблицы), но это неэффективно:

```

template<typename WORD = unsigned long long> class Key2DBuilder
{
    unsigned int n;
    bool firstNotSmaller;
public:
    typedef WORD WORD_TYPE;
    Key2DBuilder(unsigned int theN = numeric_limits<unsigned int>::max(),
        bool theFirstNotSmaller = true): n(theN),
        firstNotSmaller(theFirstNotSmaller){}
    WORD toID(unsigned int n1, unsigned int n2) const
    {
        assert(n1 < n && n2 < n);
        return firstNotSmaller ? n1 * n + n2 : n2 * n + n1;
    }
}

```

```

pair<unsigned int, unsigned int> to2D(WORD key) const
{
    pair<unsigned int, unsigned int> nln2(key/n, key % n);
    assert(nln2.first < n && nln2.second < n);
    if(!firstNotSmaller) swap(nln2.first, nln2.second);
    return nln2;
}
};

```

- ♦ для поиска самого длинного совпадения в битовых последовательностях, меньших, чем размер слова, — например, в сетевых маршрутизаторах, используйте отображение (map) с самым длинным совпадением как предшественника.

## 8.9. Префиксное дерево

Префиксное дерево (trie) — это структура, ключи в которой представляют собой массивы объектов переменной длины, сравниваемые за время  $O(1)$ . Предположим, что векторы — это строки, а у дерева в каждом узле хранится указатель на строку, состоящую из  $i$  символов, ведущую к узлу, и отображение (map) от следующей буквы до соответствующего узла. Корень разветвляется на букве 0 и указывает на элемент, представленный пустой строкой. Дочерний элемент узла, проверяющий букву  $i$ , проверяет букву  $i + 1$ . Поиск начинается с корня и следует по указателям до тех пор, пока не закончатся символы или узлы, что занимает время  $O(hM)$ , где  $M$  — время поиска следующего узла (рис. 8.8).

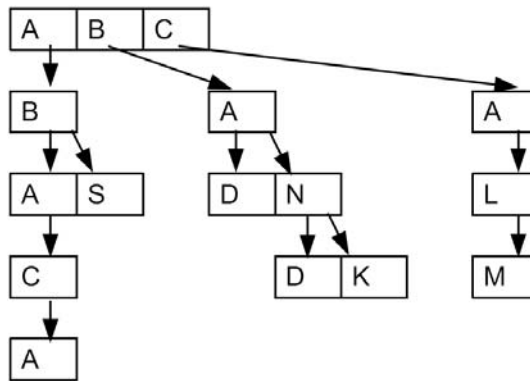


Рис. 8.8. Структура абстрактного указателя префиксного дерева

Отображение можно реализовать разными способами, но чаще всего это делается в виде массива или динамической отсортированной последовательности. В первом случае  $M = O(1)$ , но используется слишком много места, а итерация по порядку неэффективна. Во втором —  $M = O(\lg(\text{размер алфавита}))$ . Префиксные деревья дополнительно поддерживают:

- ♦ *инкрементный поиск* — например, последовательный поиск «h», «he», «hel», «hell» и «hello» занимает  $O(h)$  времени;
- ♦ расширения суффиксов — ключи, достижимые из узла высоты  $i$ , имеют общий префикс длины  $i$ .

Если в задаче нужно выполнять такие операции — это достаточно веская причина для использования префиксного дерева. Дерево с любой реализацией отображения обычно занимает больше памяти, чем LCP-Treap, и не представляет ключи как отдельные объекты, что затрудняет итерацию, если для нее нужны значения ключей. Отображение и *расширение суффиксов* влекут за собой повышение затрат памяти. После достижения некоторой глубины префикс ключа однозначно идентифицирует его, но в дереве один узел используется для каждого ключевого объекта вместо одного узла с суффиксом. Таким образом, для длинных строк с небольшим средним значением lcp требуется много памяти.

## 8.10. Тернарное декартово дерево

Тернарное декартово дерево (ТДД) эффективно реализует отображение узлов динамической отсортированной последовательности. Это дерево имитирует многоключевую быструю сортировку с помощью указателей влево, вниз и вправо с символом поворота. Поиск идет вниз, если  $i$ -й символ — опорная точка узла, влево, если он меньше, и вправо, если больше. Элемент хранится в узле, где последний символ ключа является опорным. Приоритеты декартова дерева определяют древовидную структуру узлов, проверяющих  $i$ -й символ. При выполнении операций используются асимптотически оптимальные затраты памяти и времени. Последнее равно  $O(\lg(n) + \text{средняя длина ключа})$ . Дерево не поддерживает ключи длины 0, но при необходимости может обрабатывать их (рис. 8.9). Структура памяти ТДД представлена на рис. 8.10.

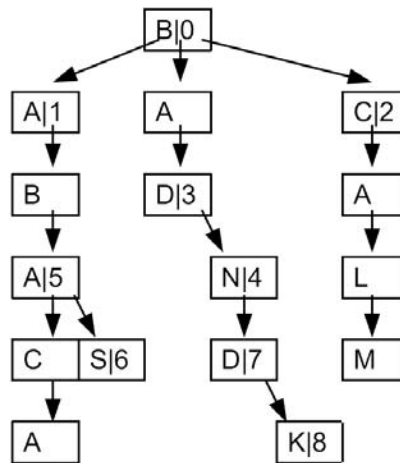


Рис. 8.9. Структура указателя ТДД и приоритеты.  
При наличии боковых ссылок приоритет определяют числа

```

template<typename ITEM, typename KEY_OBJECT = unsigned char, typename
    COMPARATOR = DefaultComparator<ITEM> > class TernaryTreapTrie
{
    COMPARATOR c;
    struct Node
    {
        KEY_OBJECT pivot;
        unsigned int priority;
    };
};
  
```

```

Node *next, *left, *right;
ITEM* item;
Node(KEY_OBJECT const& thePivot): next(0), left(0), right(0),
    item(0), pivot(thePivot), priority(GlobalRNG().next()){}
}* root;
Freelist<ITEM> itemF;
Freelist<Node> nodeF;
Node* constructFrom(Node* node)
{
    Node* result = 0;
    if(node)
    {
        result = new (nodeF.allocate()) Node(node->pivot);
        result->priority = node->priority;
        if (node->item) result->item = new (itemF.allocate()) ITEM(node->item);
        result->left = constructFrom(node->left);
        result->next = constructFrom(node->next);
        result->right = constructFrom(node->right);
    }
    return result;
}
public:
TernaryTreapTrie (COMPARATOR const& theC = COMPARATOR()): root(0), c(theC){}
TernaryTreapTrie (TernaryTreapTrie const& other):
    c(other.c){root = constructFrom(other.root);}
TernaryTreapTrie& operator=(TernaryTreapTrie const& rhs)
    {return genericAssign(*this, rhs);}
};

```

Затраты памяти составляют пять слов на ключевой объект, поэтому расширение суффикса обходится дорого, но многие узлы используются многократно, особенно если у элементов высокое среднее значение *lcr*. Для сравнения: в структуре *LCPTreap* используются пять служебных слов на ключ (без расширений).

Поиск реализован с помощью добавочного поиска, который управляет дескриптором текущего узла. Идея последнего состоит в том, чтобы начать не с корня, а с существующего узла, который, как известно, является префиксом искомого ключа. Объект дескриптора инкапсулирует это для удобства вызывающей стороны:

```

struct Handle
{
    Node* node;
    int i;
    Handle(Node* theNode = 0, int theI = 0): node(theNode), i(theI){}
};
Node* findNodeIncremental (KEY_OBJECT* key, int keySize, Handle& h)
{
    while (h.node && h.i < keySize)
    {
        if (c.isEqual (key[h.i], h.node->pivot))
        {
            if (h.i == keySize - 1) return h.node;
            else {h.node = h.node->next; ++h.i;}
        }
    }
}

```

```

    else if(c(key[h.i], h.node->pivot)) h.node = h.node->left;
    else h.node = h.node->right;
}
h = Handle(); // не найдено
return 0;
}
ITEM* findIncremental(KEY_OBJECT* key, int keySize, Handle& h)
{
    if(!h.node) h.node = root;
    Node* result = findNodeIncremental(key, keySize, h);
    return result ? result->item : 0;
}
Node* findNode(KEY_OBJECT* key, int keySize)
{
    Handle h(root, 0);
    return findNodeIncremental(key, keySize, h);
}
ITEM* find(KEY_OBJECT* key, int keySize)
{
    Node* result = findNode(key, keySize);
    return result ? result->item : 0;
}

```

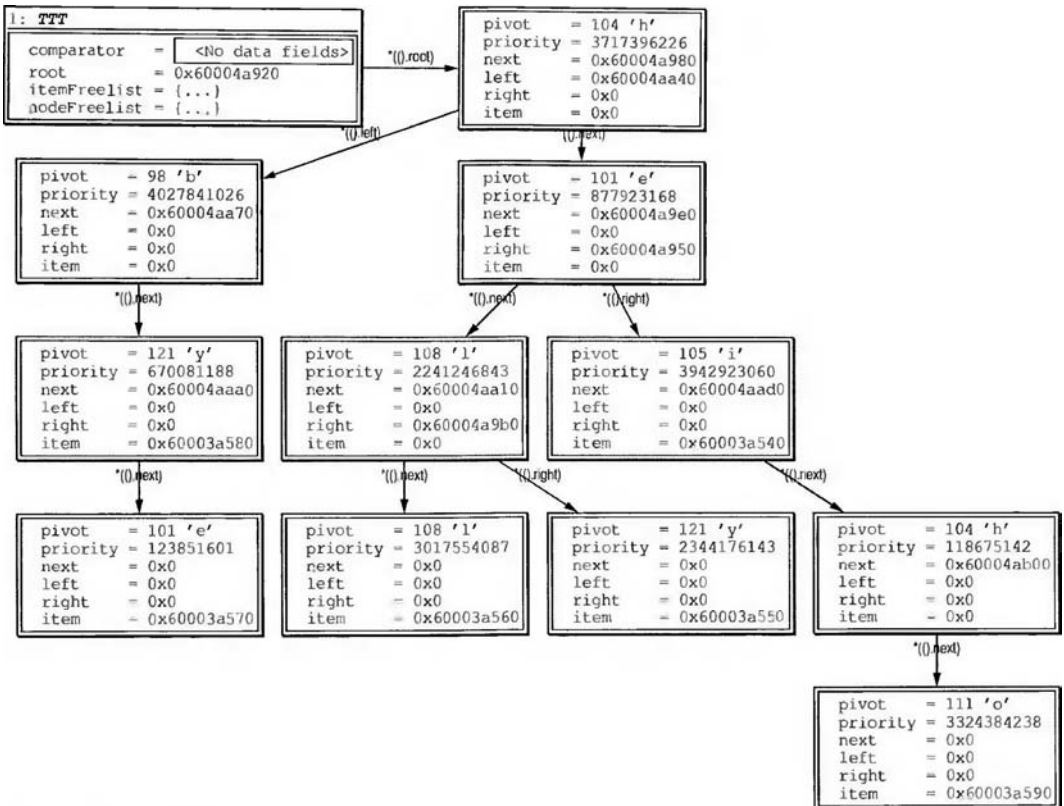


Рис. 8.10. Структура памяти ТДД

Время выполнения инкрементного поиска такое же, как и при обычном поиске, но  $n$  — это размер поддерева начального узла, а длина ключа уменьшается на длину переданного префикса.

Вращения и объединение выполняются так же, как у декартова дерева. Операция вставки работает аналогично поиску, но добавляет новые узлы, когда заканчиваются существующие, и, как и в случае с декартовым деревом, возвращает ссылки на пути вверх, чтобы восстановить порядок кучи:

```
Node* insertNode(Node* node, KEY_OBJECT* key, int keySize,
    ITEM const& item, int i)
{
    if(!node) node = new(nodeF.allocate())Node(key[i]);
    if(c.isEqual(key[i], node->pivot)) // спуск на уровень вниз
        if(i == keySize - 1) // переход к желаемому узлу
        {
            if(node->item) *node->item = item;
            else node->item = new(itemF.allocate())ITEM(item);
        }
        else node->next = insertNode(node->next, key, keySize, item,
            i + 1); // продолжаем
    else
    { // движение вбок
        bool goLeft = c(key[i], node->pivot);
        Node* chosenChild = goLeft ? node->left : node->right;
        chosenChild = insertNode(chosenChild, key, keySize, item, i);
        if(chosenChild->priority < node->priority)
            node = goLeft ? rotateRight(node) : rotateLeft(node);
    }
    return node;
}

void insert(unsigned char* key, int keySize, ITEM const& item)
{
    assert(keySize > 0);
    root = insertNode(root, key, keySize, item, 0);
}
```

Операция удаления находит элемент и на обратном пути удаляет на пути к нему узлы, в которых нет элемента или указателя вниз. Это осуществляется путем непосредственного удаления узла и присоединения к дочерним узлам, и более эффективно, чем вращение, но менее удобно для дополнений, что в рассматриваемом случае не проблема:

```
Node* join(Node* left, Node* right)
{
    if(!left) return right;
    if(!right) return left;
    if(left->priority < right->priority) // низкий приоритет повышается
    {
        left->right = join(left->right, right);
        return left;
    }
}
```



```

    else
    {
        right->left = join(left, right->left);
        return right;
    }
}

Node* removeR(Node* node, KEY_OBJECT* key, int keySize, int i)
{
    if(node)
    {
        bool isEqual = c.isEqual(key[i], node->pivot);
        if(isEqual && i == keySize - 1)
        { // удаляем найденный элемент, если он существует
            if(!node->item) return node;
            itemF.remove(node->item);
            node->item = 0;
        }
        else
        { // переход к следующему узлу
            Node** child;
            if(isEqual){child = &node->next; ++i;}
            else if(c(key[i], node->pivot)) child = &node->left;
            else child = &node->right;
            *child = removeR(*child, key, keySize, i);
        }
        if(!node->item && !node->next)
        { // удаление пустого узла
            Node* left = node->left, *right = node->right;
            nodeF.remove(node);
            node = (left || right) ? join(left, right) : 0;
        }
    }
    return node;
}

void remove(KEY_OBJECT* key, int keySize)
{
    assert(keySize > 0);
    root = removeR(root, key, keySize, 0);
}

```

Самое длинное совпадение — это любой элемент с наибольшим  $\text{lcp}$ , где  $x$  — элемент с самым коротким ключом. Для префиксных деревьев скорость получается высокой. Время выполнения составляет  $O(\lg(n) + \text{lcp})$ :

```

ITEM* longestMatch(KEY_OBJECT* key, int keySize)
{
    Node* node = root, *result = 0;
    for(int i = 0; node && i < keySize; i++)
        if(c.isEqual(key[i], node->pivot))
        {
            result = node;
            if(i == keySize - 1) break; // достигнут последний ключ
        }
}

```

```

        else {node = node->next; ++i;}
    }
    else if(c(key[i], node->pivot)) node = node->left;
    else node = node->right;
    return result ? result->item : 0;
}

```

Обычная итерация невозможна из-за разбиения ключей, поэтому вместо нее выполняется поиск по диапазону. В каждой итерации алгоритм перебирает узлы по порядку и выполняет указанный пользователем функтор на каждом из них. Например, функтор может копировать все элементы в вектор. Используя их, можно найти все элементы, ключи которых имеют общий префикс:

```

template<typename ACTION> void forEachNode(Node* node, ACTION& action)
{
    if (node)
    {
        action(node);
        forEachNode(node->left);
        forEachNode(node->next);
        forEachNode(node->right);
    }
}

struct CopyAction
{
    Vector<ITEM*>& result;
    CopyAction(Vector<ITEM*>& theResult): result(theResult) {}
    void operator() (Node* node)
    {if (node->item) result.append(node->item);}
};

Vector<ITEM*> prefixFind(KEY_OBJECT* key, int lcp)
{
    Vector<ITEM*> result;
    CopyAction action(result);
    forEachNode(findNode(key, lcp), action);
    return result;
}

```

Время выполнения —  $O(\text{префикс поиска} + \text{количество узлов в найденном поддереве})$ .

## 8.11. Сравнение производительности

Сравнение производительности рассмотренных в этой главе структур представлено в табл. 8.1.

Как можно видеть, время выполнения  $10^6$  вставок в 10 раз больше времени поиска и такого же количества удалений. Затраты памяти оценивались после вставок. В реализациях декартова дерева (treap) и LCPTreap не было кода расширения. Struct10 — это структура из 10 целых чисел, разработанная таким образом, чтобы LCP был либо высоким, либо низким, в зависимости от задачи.

Декартово дерево выигрывает у списка с пропусками, а LCPTreap — у ТДД. Хеш-таблицы выигрывают по времени, как и ожидалось, но незначительно. Использование

памяти в хеш-таблице линейной задачи связано с удвоением массива, что не подходит для больших типов.

Таблица 8.1

Тип ключа	Список с пропусками		Декартово дерево		Цепочка		Линейное зондирование		LCPTgear		ТДД	
	Время (сек)	Память (Мбайт)	Время (сек)	Память (Мбайт)	Время (сек)	Память (Мбайт)	Время (сек)	Память (Мбайт)	Время (сек)	Память (Мбайт)	Время (сек)	Память (Мбайт)
int	27	42	16	38	8	33	8.3	25				
Struct10	93	95	72	90	49	86	52	117	26	96	123	98
Struct10_2	34	95	21	90	49	86	51	117	26	96	164	1578
Struct10_4									44	96	59	55
Struct10_5									145	96	164	1578

## 8.12. Примечания по реализации

У большинства реализаций есть свои уникальные черты:

- ♦ реализация декартова дерева полностью поддерживает итераторы и общие расширения;
- ♦ так же и у LCPTgear, реализация которой взята непосредственно из литературы;
- ♦ реализация ТДД оригинальна и основана на предложении в работе [8.2].

Решение исключить другие реализации префиксных деревьев довольно редко, но вполне обоснованно (см. *разд. «Комментарии»*). Я провел несколько месяцев в экспериментах, прежде чем решил оставить только ТДД, т. к. функциональность динамической отсортированной последовательности достаточно проста.

## 8.13. Комментарии

Использование свободных списков со сборкой мусора позволяет не писать деструкторы. Там, где они нужны в реализации дерева, проще всего реализовать рекурсивное удаление дерева, как в конструкторе копирования. Более умное нерекурсивное решение состоит в том, чтобы итеративно удалить верхний узел и неявно выполнить объединение левого и правого поддеревьев. В сбалансированном дереве необходимости избавляться от рекурсии нет, потому что стек, скорее всего, будет маленьким, что в целом всегда упрощает работу.

В работе [8.8] вы можете подробнее почитать о других операциях со списками с пропусками, в частности о *расширении обратного указателя*. Для расширения lcp можно прикрепить lcp к каждому указателю башни (см. [8.4]).

Существует множество разных сбалансированных деревьев, у которых операции отображения в худшем случае выполняются за  $O(\lg(n))$  и/или есть другие хорошие свойства, но все они сложнее декартова дерева и в среднем его не превосходят. В частности,

в многих библиотеках реализовано *красно-черное дерево* (см. [8.1]). Амортизированная (но неожиданная) скорость вставки у него равна  $O(1)$ , но его стратегия балансировки в тернарном префиксном дереве не работает. Операции с отображениями в декартовом дереве выполняются быстрее, чем в красно-черном дереве, которое, в свою очередь, быстрее, чем список с пропусками (см. [8.7]). *Красно-черное дерево с левым обучением* (см. [8.14, 8.12]) проще, но уже не выполняет вращение за  $O(1)$ , т. к. оно изоморфно более старому *2-3-дереву*, в котором операции вращения нет. В-дерево (см. главу 13. *Алгоритмы для работы с внешней памятью*) с  $B = 2$  — это *дерево 2-3-4* (см. [8.9]), изоморфное красно-черному дереву и обладающее теми же свойствами. Последнее было получено из первого для экономии места на пустых указателях.

У *расширяющегося дерева* (см. [8.3]) не самые лучшие постоянные коэффициенты, но оно интересно с точки зрения теории, потому что в нем не используется дополнительное пространство для балансировки, а каждая последовательность из  $t$  операций над  $k$  элементами выполняется за амортизированное время  $O(\text{mlg}(k))$ . *Дерево AVL* (см. [8.3]) уравнивается с помощью вращения, поддерживающего ограниченную разницу в высоте. По сравнению с красно-черным деревом, оно немного более сбалансировано, обеспечивает более быстрый поиск, но вставка выполняется медленнее, а также нет амортизированного вращения за  $O(1)$ . *Дерево со сбалансированным весом* (см. [8.3]) похоже на *дерево AVL*, но балансируется по весу — т. е. по количеству узлов в каждом поддереве. У этого дерева увеличение размера узла не требует дополнительных затрат памяти, но поиск и вставка выполняются медленнее. То же самое характерно для *рандомизированного дерева* (см. [8.9]), которое настраивается на случайность после каждой операции, используя пропорции количества узлов. В работе [8.10] обсуждаются вопросы балансировки в целом. В работе [8.3] упоминается несколько других реализаций, о которых вы, вероятно, никогда не слышали (и не удивительно).

Использование приоритетов в декартовом дереве, по-видимому, является одним из немногих способов сбалансировать тернарное дерево. Худшего случая стратегии балансировки нет, потому что обычная балансировка роста или веса не работает (см. [8.2]). У каждого из деревьев бывают свои проблемы:

- ◆ у префиксного дерева (array-map trie) с ключевым объектом размером в 4 бита разумные размеры отображения, но тем не менее оно занимает много памяти, не имеет универсальности и простых операций с префиксами, из-за чего становится плохой хеш-таблицей;
- ◆ у префиксного дерева с корзиной (bucket trie) помещайте элементы с общим префиксом в маленькую корзину, в которой выполняется линейный поиск, что позволяет сэкономить память, затрачиваемую на расширение суффикса. У любого дерева, включая ТДД, это делает операции громоздкими и неэффективными из-за необходимости отслеживать несколько типов узлов и иногда преобразовывать их типы при вставке и удалении;
- ◆ *цифровое дерево поиска* (digital search tree, DST) — похоже на дерево, но вместо сравнений для ветвления в нем используются биты ключей (см. [8.12]). То есть на уровне  $i$ , если  $i$ -й бит равен 0, идем налево, иначе — направо. То есть  $h \leq$  размера ключа в битах. Это работает только с битовыми последовательностями и неэффективно для последовательностей с высоким битом  $\text{lsr}$ . Производительность зависит от порядка следования байтов и поддерживает другие операции — например, опера-

цию поиска предшественника, но только если лексикографический порядок битов обеспечивает правильное сравнение. Таким образом, DST даже в лучшем случае является плохой хеш-таблицей;

- ◆ *базисное дерево* (patricia trie) — лучше, чем цифровое дерево поиска, за счет перехода к отличающимся битам (см. [8.12, 8.10]). В результате эта структура данных используется для сопоставления самой длинной битовой последовательности в сетевых маршрутизаторах (см. [8.10]). Можно сделать то же самое с любой динамической отсортированной последовательностью с использованием предшественника, как обсуждалось ранее, но медлительность побитовой проверки и применимость этого дерева только для работы с битами никуда не девается.

## 8.14. Советы по дополнительной подготовке

- ◆ Исследуйте список с пропусками на предмет его производительности как функции  $p$ . В частности, попробуйте  $p = 0,25$ , как это рекомендуется в различных источниках.
- ◆ Для списка с пропусками реализуйте обратный указатель и расширение узлов после подсчета, чтобы обеспечить двунаправленную итерацию и поиск  $i$ -го элемента.
- ◆ Для списка пропуска реализуйте расширение lcr.
- ◆ Измените реализацию декартова дерева и LCPTreap, чтобы они возвращали итераторы вместо указателей узлов.
- ◆ Некоторые библиотечные стандарты могут требовать в худшем случае логарифмической производительности для динамически отсортированной последовательности. Здесь лучше всего подойдет красно-черное дерево. Изучите его, внедрите и сравните производительность с помощью декартова дерева, списка с пропусками и карт STL.
- ◆ Автоматизируйте сравнение производительности с помощью сценария на Python (или другого языка, обеспечивающего простое взаимодействие с ОС) для регистрации использования памяти различными структурами.

## 8.15. Список рекомендуемой литературы

- 8.1. Cormen T. H., Leiserson C. E., Rivest R. L., & Stein C. (2009). Introduction to Algorithms. MIT Press.
- 8.2. Badr G. H., & Oommen B. J. (2005). Self-adjusting of ternary search tries using conditional rotations and randomized heuristics. The Computer Journal, 48(2), 200–219.
- 8.3. Brass P. (2008). Advanced Data Structures. Cambridge University Press.
- 8.4. Ciriani V., Ferragina P., Luccio F., & Muthukrishnan S. (2007). A data structure for a sequence of string accesses in external memory. ACM Transactions on Algorithms (TALG), 3(1), 6.
- 8.5. Crescenzo, P., Grossi R., & Italiano G. F. (2003). Search data structures for skewed strings. In Experimental and Efficient Algorithms (pp. 81–96). Springer.
- 8.6. Grossi R., & Italiano G. F. (1999). Efficient techniques for maintaining multidimensional keys in linked data structures. In Automata, Languages, and Programming (pp. 372–381). Springer.
- 8.7. Hege D. A. (2004). A disquisition on the performance behavior of binary search tree data structures. European Journal for the Informatics Professional, 5(5), 67–75.

- 8.8. Mehlhor K., & Näher S. (1999). LEDA: a Platform for Combinatorial and Geometric Computing. Cambridge University Press.
- 8.9. Mehlhorn K., & Sanders P., Dietzfelbinger M., & Dementiev R. (2019). Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox. Springer.
- 8.10. Mehta D. P., & Sahni S. (Eds). (2018). Handbook of Data Structures and Applications. CRC.
- 8.11. Pugh W. (1990). Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM, 33(6), 668–676.
- 8.12. Sedgewick R. (1999). Algorithms in C++. Parts 1–4. Addison-Wesley.
- 8.13. Sedgewick R. (2008). Left-leaning red-black trees. In Dagstuhl Workshop on Data Structures.
- 8.14. Sedgewick R., & Wayne K. (2011). Algorithms. Addison-Wesley.
- 8.15. Seidel R., & Aragon C. R. (1996). Randomized search trees. Algorithmica, 16(4–5), 464–497.

## 9. Хеширование

### 9.1. Введение

Мое образование и опыт проведения собеседований говорят, что студенты, как правило, плохо понимают алгоритмы хеширования даже после прохождения курса по алгоритмам. Многих сбивает с толку просьба сравнить хеш-таблицу с динамически отсортированной последовательностью. Я считаю, что проблема заключается в рандомизации и изменениях размера хеш-таблиц, так что в этой главе будут приведены подробные реализации полезных хеш-таблиц и хеш-функций.

### 9.2. Хеш-функции

Хеш-функция  $h$  отображает объекты  $x$  в целые числа по модулю  $m$  для некоторого  $m$ . Функция  $h$  должна быть регулярной функцией и порождать мало *коллизий*, где для  $x \neq y$  получится  $h(x) = h(y)$ . Теоретически функция  $h$  нужна только для байтовых массивов переменной длины, поскольку она позволяет интерпретировать любой объект как таковой, но с некоторыми типами данных использовать их можно эффективно. Вы можете хешировать разные типы — например, строки, путем преобразования их в эффективно хешируемый тип.

К сожалению, простой функции  $h(x) = x \% m$  для целых чисел недостаточно. Для объектов в более широком смысле такая  $h$  будет конкатенировать несколько ненулевых байтов и формировать целое число  $x$ . Часто числа  $x$  отличаются только младшими битами, и эта ситуация идеальна для хеш-функций. Некоторые источники даже рекомендуют использовать такую  $h$  из-за ее скорости и *справедливости*, т. к. при хешировании всех возможных  $x$  все возможные значения хеша будут равновероятны. Но производительность может резко ухудшиться, например, при переходе с системы с прямым порядком байтов на систему с обратным порядком байтов, или если умножить все  $x$  на какой-то коэффициент для удобства в рамках приложения. В общем случае функция  $h$  должна просматривать все байты  $x$  и извлекать из них уникальный информационный контент, генерируя из него уникальный идентификатор. Скорость работы функции  $h$  обычно не является узким местом, а вот если она будет быстрой, но некачественной, то проблемы обязательно возникнут.

Если есть набор объектов размера  $n > m$ , то все функции  $h$  хешируют по крайней мере  $n/m$  из них одинаковым значением. В худшем случае любая функция  $h$  хеширует все  $x$  одним и тем же значением, поэтому производительность нужно оценивать не по худшему, а по типичному сценарию. Чтобы быть полезной, функция  $h$  должна вычисляться за время  $O(1)$  по отношению к  $n$ . Для объекта  $x$  размера  $s$  требуется время  $O(s)$ . Также нужно избегать коллизий, поэтому *случайная* функция  $h$  теоретически идеальна, т. к. работает без смещения в сторону каких-то значений. Хеш-функция  $h$ :

- ◆ *универсальна*, если инициализируется случайным начальным числом  $a$ , вычисляет  $h(x) = f(x, a)$  и  $\forall x \neq y, \Pr(h(x) = h(y)) < \frac{\text{константа}}{m}$ . Тогда для целого числа  $k$  и любого  $x \Pr(h(x) = k) < \frac{\text{константа}}{m}$ . Случайное начальное значение не нарушает регулярность, поскольку оно сохраняется и используется повторно. Как будет показано далее, на практике этот подход максимально близок к случайной генерации;
- ◆ *поддерживает вращение*, если используется для хеширования массива объектов, и после изменения одного объекта может обновить хеш массива за время  $O(s)$ ;
- ◆ *переносимая*, если выдает одно и то же значение в любой архитектуре и при любом размере слова. Обычно это не проблема, т. к. значения  $h$  не хранятся, но любая зависимость от размера слова или неявная обработка переполнения нарушает переносимость.

Большинство хеш-функций имеют форму  $h(x) \% m$  для некоторого  $m$ , а некоторым приложениям не требуется оператор « $\%m$ » или его можно заменить более быстрым  $\&(m - 1)$ , когда  $m = 2b$ . Здесь  $x$  обычно представляет собой число или массив чисел. Однако нередко требуется автоматически хешировать и другие встроенные типы. Итак, когда мы говорим о реализации:

- ◆ базовая функция  $h$  должна поддерживать целые числа и массивы целых чисел того типа слова, который она поддерживает;
- ◆ массивы `char` упаковываются для заполнения слова и хешируются как массивы слов. Большие целые числа разбиваются на массивы слов, если у них кратный размер. Другие числа преобразуются в массивы символов и хешируются напрямую;
- ◆ вместо массивов поддерживаются более общие объекты-конструкторы, полезные в случаях, когда нужно создать  $h$  для пользовательских типов;
- ◆ некоторые обертки обрабатывают оператор « $\%m$ » и показанные ранее операции приведения;
- ◆ каждая  $h$  знает свое максимальное значение, и значение  $m - 1$  не должно его превышать.

```
template<typename HASHER> class MHash
{
    unsigned long long m;
    HASHER h;
public:
    MHash(unsigned long long theM): m(theM)
    {assert(theM > 0 && theM <= h.max());}
    typedef typename HASHER::WORD_TYPE WORD_TYPE;
    unsigned long long max() const{return m - 1;}
    unsigned long long operator()(WORD_TYPE const& x) const{return h(x) % m;}
    typedef typename HASHER::Builder Builder;
    Builder makeBuilder() const{return h.makeBuilder();}
    unsigned long long operator()(Builder b) const{return h(b) % m;}
};
```



```
template<typename HASHER> class BHash
{
    unsigned long long mask;
    HASHER h;
public:
    BHash(unsigned long long m): mask(m - 1){assert(m > 0 && isPowerOfTwo(m));}
    typedef typename HASHER::WORD_TYPE WORD_TYPE;
    unsigned long long max()const{return mask;}
    unsigned long long operator()(WORD_TYPE const& x)const{return h(x) & mask;}
    typedef typename HASHER::Builder Builder;
    Builder makeBuilder()const{return h.makeBuilder();}
    unsigned long long operator()(Builder b)const{return h(b) & mask;}
};
```

Рассмотрим еще одну обертку, которая делает базовую функцию *h* эффективной для работы со многими типами данных. Значение *x* хешируется как число, если это возможно, а в противном случае вычисляется как массив наименьшего размера. Имейте в виду, что у некоторых типов объектов с одинаковым значением могут быть разные хеши:

- ◆ у типа `double`  $-0 = 0$ , и несколько битовых комбинаций представляют NaN;
- ◆ скрытые атрибуты у типов классов могут влиять на хеш-значение;
- ◆ у структур в байтах заполнения может лежать мусор.

Таким образом, никакое автоматическое хеширование на основе приведения типов не является безопасным во всех возможных случаях, и безопасности ради только типы, занимающие одно слово, разрешено хешировать вслепую. При использовании построителя, если вы хотите немного повысить производительность при работе со специализированными типами, лучшая разумная стратегия для автоматического хеширования — использовать пользовательскую функцию *h*:

```
template<typename HASHER> class EHash
{ // обрабатываем случай возможных ошибок компиляции подстановки шаблона
    HASHER h;
    template<typename WORD> unsigned long long hash(WORD x, true_type)const
    { // хешируем тип, если он занимает целое слово
        if(sizeof(WORD) <= sizeof(WORD_TYPE)) return h(x);
        return operator()(&x, 1); // если слово слишком большое,
                                   // оно будет разбито
    }
public:
    EHash(){} // для h, которые не используют m
    EHash(unsigned long long m): h(m) {}
    typedef typename HASHER::WORD_TYPE WORD_TYPE;
    unsigned long long max()const{return h.max();}
    template<typename WORD> unsigned long long operator()(WORD x)const
    {return hash(x, is_integral<WORD>());} // только целые слова
    class Builder
    {
        enum{K = sizeof(WORD_TYPE)};
        union
```

```

{
    WORD_TYPE xi;
    unsigned char bytes[K];
};
int byteIndex;
typename HASHER::Builder b;
friend EHash;
Builder(EHash const& eh): xi(0), byteIndex(0), b(eh.h.makeBuilder()) {}
template<typename WORD> void add(WORD const& xi, true_type)
{ // в целях безопасности поддерживается только тип размером в слово
    if(sizeof(WORD) % sizeof(WORD_TYPE) == 0)
        // точное кратное, добавляем как word_type
        for(int i = 0; i < sizeof(WORD)/sizeof(WORD_TYPE); ++i)
            b.add(((WORD_TYPE*)&xi)[i]);
    else // как последовательность символов
        for(int i = 0; i < sizeof(xi); ++i)
            add(((unsigned char*)&xi)[i]);
}
public:
void add(unsigned char bi)
{
    bytes[byteIndex++] = bi;
    if(byteIndex >= K)
    {
        byteIndex = 0;
        b.add(xi);
        xi = 0;
    }
}
template<typename WORD> void add(WORD const& xi)
{ add(xi, is_integral<WORD>()); } // только целые слова
typename HASHER::Builder operator() ()
{ // завершаем оставшиеся xi, если они есть
    if(byteIndex > 0) b.add(xi);
    return b;
}
};
Builder makeBuilder() const { return Builder(*this); }
unsigned long long operator() (Builder b) const { return h(b()); }
template<typename WORD>
unsigned long long operator() (WORD* array, int size) const
{
    Builder b(makeBuilder());
    for(int i = 0; i < size; ++i) b.add(array[i]);
    return operator() (b);
}
};

```

Для типов данных, которые не являются целыми словами или массивами слов, нужно создать пользовательскую функцию *h* с помощью предоставленных строителей. Здесь нужно использовать максимальное значение объекта. Есть и менее очевидные источники информации:

- ♦ у последовательностей также хешируется размер;
- ♦ у объединений также хешируется выбор текущего члена;
- ♦ у указателей хешируется переменная состояния null.

Любой хешируемый тип должен выполнять роль значимого идентификатора, поэтому все, что содержит, например, число `double`, должно хешироваться не просто так, а по разумной причине. Приведенный далее код включает вектор целочисленных типов и строки (`BUHash` обсуждается позже):

```
template<typename HASHER = EHash<BUHash> > class DataHash
{
    HASHER h;
public:
    DataHash(unsigned long long m): h(m){}
    typedef typename HASHER::WORD_TYPE WORD_TYPE;
    unsigned long long max()const{return h.max();}
    unsigned long long operator()(string const& item)const
    {return h(item.c_str(), item.size());}
    unsigned long long operator()(Bitset<> const& item)
    const{return h(item.getStorage().getArray(), item.wordSize());}
    template<typename VECTOR> unsigned long long operator()(VECTOR const& item)
    const{return h(item.getArray(), item.getSize());}
    typedef EMPTY Builder;
};
```

### 9.3. Универсальные хеш-функции

**Теорема А:** Если  $h$  является универсальной функцией с константой  $c$  и хеширует  $n$  элементов,  $E[\text{количество элементов, хешируемых одним и тем же значением}] < cn/m$ .

Доказательство. Пусть  $X_i = (h(x_i) = k)$ .  $E[\text{количество элементов, хешируемых в } k] = E[\sum X_i] = \sum \Pr(h(x_i) = k) < \sum \frac{c}{m} = \frac{cn}{m}$ .

**Теорема Лагранжа** (см. [9.16]): Многочлен  $\% p$  степени  $k$ , где  $p$  — простое число, имеет  $\leq k$  решений по модулю  $p$ , если один или несколько коэффициентов не делятся на  $p$  и  $k < p$ . Если некоторые коэффициенты отрицательны, это не имеет значения, потому что операция  $\% p$  делает их положительными.

**Теорема Б:** Для массива чисел переменной длины  $x$ , простого  $p$ ,  $m \leq p$  и случайного начального числа  $a \in [0, p-1]$  функция  $h(x) = (\sum x_i a^i) \% p \% m$  является универсальной и имеет  $c = 2k$ , где  $k$  — длина самого длинного хешированного массива (индексация  $x$  ведется с единицы).

Доказательство. Пусть  $x \neq y$  — это массивы переменной длины  $\leq k$ . Тогда  $h(x) = h(y) \leftrightarrow (\sum x_i a^i) \% p = (\sum y_i a^i) \% p + qm$  для  $qm \in (-p, p) \leftrightarrow (\sum (x_i - y_i) a^i \pm qm) \% p = 0$ , с любыми  $x_i$  или  $y_i$ , которые не существуют из-за того, что разность длин равна 0. Это полином  $\% p$  с коэффициентами  $(x_i - y_i)$  и  $\pm qm$ . По теореме Лагранжа, поскольку  $\pm qm$  не делится на  $p$ ,  $q \exists \leq k$  существует  $a$  вариантов, которые приводят к равенству для данного  $\pm q$ .

$\exists \leq 2 \frac{p}{m}$  значений  $\pm q$ , удовлетворяющих равенству  $\leq \frac{2kp/m}{p} = 2 \frac{k}{m}$ .

- ◆ Доказательство справедливо только в том случае, если вычисления не вызывают переполнения. Используйте 64 бита с  $x_i$ ,  $p < 2^{32}$  и вычислите  $a^i$  пошагово.
- ◆ Если  $m = p$ , операция « $\%m$ » не требуется, а значение  $h$  является скользящим, поскольку вы можете складывать или вычитать члены  $x_i a^i$  за время  $O(\lg(i))$ , если  $a^i$  вычисляется с использованием эффективного модульного возведения в степень (см. главу 17. Большие числа).
- ◆  $K = 1$  дает универсальную  $h$  для целых чисел с  $c = 2$ .
- ◆ Значение  $c$  велико для длинных строк.
- ◆ Функцию  $h$  можно использовать для обнаружения ошибок, потому что для повторного хеширования нужно только начальное значение. Функция CRC (см. главу 12. Разные алгоритмы и методы) имеет большую вероятность обнаружения ошибки, но использование нескольких таких  $h$  делает ее сколь угодно близкой к 1.

```
class PrimeHash2
{
    static uint32_t const PRIME = (1ull << 32) - 5;
    uint32_t seed;
public:
    PrimeHash2(): seed((GlobalRNG().next() | 1) % PRIME) {} // проверяем ненулевое значение
    typedef uint32_t WORD_TYPE;
    unsigned long long max() const { return PRIME - 1; }
    unsigned long long operator() (WORD_TYPE x) const
    { return (unsigned long long) seed * x % PRIME; }
    class Builder
    {
    public:
        unsigned long long sum;
        WORD_TYPE seed;
        friend PrimeHash2;
        Builder(WORD_TYPE theSeed): sum(0), seed(theSeed) {}
    public: // в редких случаях может возникнуть переполнение,
        // но это не страшно
        void add(WORD_TYPE xi) { sum = seed * (sum + xi) % PRIME; }
    };
    Builder makeBuilder() const { return Builder(seed); }
    unsigned long long operator() (Builder b) const { return b.sum; }
};
```

**Теорема В:** Для массивов чисел переменной длины  $x$ , простого числа  $p$ ,  $m \leq p$  и  $k$  случайных начальных чисел  $a \in [0, p - 1]$ , где  $k$  — длина самого длинного хеш-вектора,  $h(x) = (\sum a_i x_i) \% p \% m$  является универсальным с  $c = 2$ .

Доказательство. Пусть  $x \neq y$  — массивы переменной длины длиной  $\leq k$ . Тогда  $h(x) = h(y) \Leftrightarrow (\sum a_i x_i) \% p = (\sum a_i y_i) \% p + qm$  для  $qm \in (-p, p) \Leftrightarrow (a_j(y_j - x_j) + \sum_{i \neq j} (x_i - y_i) \pm qm) \% p = 0$ , где  $j$  таково, что  $y_j - x_j \neq 0$ . Значение  $a_i$  для  $i \neq j$  можно считать константой, поэтому по теореме Лагранжа результирующий линейный многочлен с коэффициентами  $y_i - x_i$  и  $\sum_{i \neq j} a_i(x_i - y_i) \pm qm$  имеет не более одного решения для данных  $\pm q$  и  $a_j$ . Существует

$\exists \leq 2 \frac{p}{m}$  значений  $\pm q$  и  $p^{k-1}$  значений  $a_i$ , для которых  $i \neq j$ , поэтому деление  $\{a_i\}$  удовле-

творяет равенству  $< \frac{2p}{m} \frac{p^{k-1}}{p^k} = \frac{2}{m}$ .

- ◆ Доказательство выполняется только в том случае, если вычисления не вызывают переполнения. Вы можете использовать 64 бита с  $x_i$  и условие  $p < 2^{32}/k$ , но на практике можно обойтись условием  $p < 2^{32}$ .
- ◆ Если  $m = p$ , оператор « $\%m$ » не требуется.
- ◆  $k = 1$  приводит к универсальному  $h$  для целых чисел с  $c = 2$ .

Из-за псевдослучайной генерации  $a_i$  из начального числа, состоящего из одного слова, доказательство неверно, т. к. становится математически невозможно изолировать произвольное  $a_j$ . Но из-за энтропии генератора псевдослучайных чисел по отношению к работе  $h$  примерно равные доли начального числа и выбор  $q$  должны дать решение уравнения. Для констант нет лучшего доказательства, чем доказательство теоремы Б, но выполнение отображения из хорошего генератора псевдослучайных чисел работает не хуже возведения в степень. К тому же оператор « $\%m$ » будет нужен лишь однажды:

```
class PrimeHash
{
    static uint32_t const PRIME = (1ull << 32) - 5;
    uint32_t seed;
public:
    PrimeHash(): seed((GlobalRNG().next() | 1) % PRIME) {} // проверяем
                                                         // ненулевое значение

    typedef uint32_t WORD_TYPE;
    unsigned long long max() const { return PRIME - 1; }
    unsigned long long operator() (WORD_TYPE x) const
    { return (unsigned long long) seed * x % PRIME; }
    class Builder
    {
    public:
        unsigned long long sum;
        WORD_TYPE a;
        friend PrimeHash;
        Builder(WORD_TYPE theSeed): sum(0), a(theSeed) {}
    public:
        void add(WORD_TYPE xi)
        { // в редких случаях может возникнуть переполнение,
          // но это не страшно
            sum += (unsigned long long) a * xi;
            a = xorshiftTransform(a);
        }
    };
    Builder makeBuilder() const { return Builder(seed); }
    unsigned long long operator() (Builder b) const { return b.sum % PRIME; }
};
```

**Теорема Г:** Для  $w$ -битных целых чисел  $m = 2^b$  и нечетных случайных чисел  $a \in [0, 2^w - 1]$

функция  $h(x) = \frac{xa \% 2^w}{2^{w-b}}$  универсальна, даже если  $xa$  переполняется (см. [9.3]). Объединение функций  $h$  из теорем В и Г дает очень эффективную комбинированную универсальную  $h$  для  $m = 2b$ . Поскольку первая работает с 32-битными словами, последнюю можно реализовать так же, не убирая поддержку 64-битных слов. Помните, что малое  $a$  хеширует малое  $x$  значением 0, хотя на практике это не проблема:

```

class BUHash
{
    uint32_t a, wLB;
    BHash<PrimeHash> h;
public:
    BUHash(unsigned long long m): a(GlobalRNG().next() | 1), // проверяем
                                                                    // ненулевое значение
        wLB(32 - lgCeiling(m)), h(m) {assert(m > 0 && isPowerOfTwo(m));}
    typedef uint32_t WORD_TYPE;
    unsigned long long max() const {return h.max();}
    uint32_t operator() (WORD_TYPE const& x) const {return (a * x) >> wLB;}
    typedef BHash<PrimeHash>::Builder Builder;
    Builder makeBuilder() const {return h.makeBuilder();}
    unsigned long long operator() (Builder b) const {return h(b);}
};

```

## 9.4. Неуниверсальные хеш-функции

Иногда функция  $h$  всегда должна давать один и тот же результат для одного и того же  $x$  и поэтому не может быть универсальной. Например, в Java функция `hashCode` каждого объекта вычисляет один и тот же хеш (в отличие от `unordered_map` в C++, который позволяет передавать начальное значение в аргумент конструктора). У таких высококачественных функций  $h$  имеются следующие характеристики (см. [9.5, 9.6]):

- ◆ лавина — изменение любого входного бита изменяет каждый бит хеш-функции с вероятностью 50%;
- ◆ смещение — входные данные любой длины и хеш-значения одинаково вероятны в любом диапазоне;
- ◆ столкновения:  $E[\text{число}] \approx E$  случайной функции  $h$ .

У качественной хеш-функции *FNV* для байтовых массивов переменной длины,

$$hC(x, i) = \begin{cases} a, & i = 0 \\ hC(x, i-1)b^x[i], & \end{cases} \quad \text{где } a, b = 2166136261, 16777619 \text{ для 32 бит и}$$

14695981039346656037, 1099511628211 для 64 бит (см. [9.15]). Значения  $b$  — это подобранные простые числа, а значения  $a$  — произвольные. Тщательно протестирована была только 32-битная версия (см. [9.5, 9.6, 9.14]), но 64-битная должна быть такого же качества:

```

struct FNVHash
{
    typedef unsigned char WORD_TYPE;
    unsigned long long max() const {return numeric_limits<uint32_t>::max();}
    uint32_t operator() (WORD_TYPE const& x) const
    {
        Builder b(makeBuilder());
        b.add(x);
        return b.sum;
    }
};

```

```

class Builder
{
    uint32_t sum;
    friend FNVHash;
    Builder(): sum(2166136261u) {}
public: // в редких случаях может возникнуть переполнение,
        // но это не страшно
    void add(WORD_TYPE xi){sum = (sum * 16777619) ^ xi;}
};
Builder makeBuilder()const{return Builder();}
uint32_t operator() (Builder b)const{return b.sum;}
};

struct FNVHash64
{
    typedef unsigned char WORD_TYPE;
    unsigned long long max()const{return numeric_limits<uint64_t>::max();}
    uint64_t operator() (WORD_TYPE const& x)const
    {
        Builder b(makeBuilder());
        b.add(x);
        return b.sum;
    }
    class Builder
    {
    {
        uint64_t sum;
        friend FNVHash64;
        Builder(): sum(14695981039346656037ull) {}
    public:
        void add(WORD_TYPE xi){sum = (sum * 1099511628211ull) ^ xi;}
    };
    Builder makeBuilder()const{return Builder();}
    uint64_t operator() (Builder b)const{return b.sum;}
};

```

$h_i(x) = \left\{ \begin{matrix} 0, i = 0 \\ h_{i-1}(x), x[i] \end{matrix} \right\}$  — это хеш в виде массива переменной длины и инкрементно вы-

числяемой  $h$ . Этот метод работает лучше, чем  $\text{combine}(f(h_{i-1}(x)), x[i])$ , потому что у простых объединителей последний байт не влияет на все биты  $h$ . В качестве функции  $f$  очевидным выбором являются случайные переходы генератора, а в качестве операторов-объединителей: «+» и «^». Объединитель должен отличаться от  $f$ . Для эффективности  $h$  должна хешировать массивы целых чисел, а не побайтно, но иногда этого не избежать, например в случае с FNV.

*Хеш-функция Xorshift* использует переход QualityXorshift64 и оператор объединения «+»:

```

struct Xorshift64Hash
{
    typedef uint64_t WORD_TYPE;
    unsigned long long max()const{return numeric_limits<WORD_TYPE>::max();}
    uint64_t operator() (WORD_TYPE x)const
    {return QualityXorshift64::transform(x);}
}

```

```

class Builder
{
    uint64_t sum;
    friend Xorshift64Hash;
    Builder(): sum(0) {}
public:
    void add(WORD_TYPE xi){sum = QualityXorshift64::transform(sum + xi);}
};
Builder makeBuilder()const{return Builder();}
uint64_t operator()(Builder b)const{return b.sum;}
};

```

Следующие обобщенные свойства существенно различны и друг в друга не преобразуются:

- ◆ универсальность — любые два элемента с высокой вероятностью не совпадут;
- ◆ лавина и смещение — максимально используется энтропия данных;
- ◆ отсутствие корреляции генератора псевдослучайных чисел — свойство последовательности, похожее на лавину, но соседние числа могут переходить в соседние числа.

Прежде, чем двигаться дальше, отметим, что хеш-функция Xorshift, несмотря на превосходную лавину и полностью прозрачную механику, является лишь иллюстрацией общей конструкции. В неуниверсальных задачах всегда нужны хорошо изученные  $h$  вроде FNV.

## 9.5. Скользящие хеш-функции

Самая быстрая и удобная скользящая функция  $h$  — это *табличная хеш-функция* (также называемая *хешированием Зобриста*). Она интерпретирует  $x$  как массив байтов переменной длины. Тогда  $h(x) = \text{XOR} \sum \text{table}[x_i]$ , где таблица имеет размер 256 и содержит случайные числа. Скольжение выполняется по формуле  $\wedge = \text{таблица}[\text{старая}] \wedge \text{таблица}[\text{новая}]$  и реализуется за время  $O(1)$ . Все перестановки байтов хеша  $x$  дают одно и то же число, что делает функцию  $h$  непригодной для общего использования:

```

class TableHash
{
    enum{N = 1 << numeric_limits<unsigned char>::digits};
    unsigned int table[N];
public:
    TableHash(){for(int i = 0; i < N; ++i) table[i] = GlobalRNG().next();}
    typedef unsigned char WORD_TYPE;
    unsigned long long max()const{return numeric_limits<unsigned int>::max();}
    unsigned int operator()(WORD_TYPE const& x)const
    {
        Builder b(makeBuilder());
        b.add(x);
        return b.sum;
    }
    unsigned int update(unsigned int currentHash, unsigned char byte)
        const{return currentHash ^ table[byte];} // для сложения и вычитания

```



```

class Builder
{
    unsigned long long sum;
    TableHash const& h;
    friend TableHash;
    Builder(TableHash const& theH): sum(0), h(theH) {}
public: // в редких случаях может возникнуть переполнение,
        // но это не страшно
    void add(unsigned char xi){sum ^= h.table[xi];}
};
Builder makeBuilder() const{return Builder(*this);}
unsigned long long operator()(Builder b) const{return b.sum;}
};

```

В настольных играх хешируются пары «фигурка-клетка». Доска обычно состоит из квадратов, где в каждом находится не более одной фигуры. Например, в шахматах  $64 \times (2 \times 6 + 1) = 832$  возможных значения. Каждое из них можно представить двумя байтами, но более эффективно и качественно использовать таблицу размером 832. Если клеток на поле много, можно не прибегать к таблице, а взять хеш-функцию  $g$  и формулу  $h(x) = \text{XOR} \sum g(x_i)$ .

## 9.6. Коллекция хеш-функций

При  $k > 2$  требуется  $k$  хеш-функций — например, для фильтра Блума (см. *разд. 9.11*). В этом случае набор вида  $h_i = h_1 + ih_2$  дает достаточно случайные результаты и является более эффективным, чем выполнение хеширования  $k$  раз (см. [9.7]).

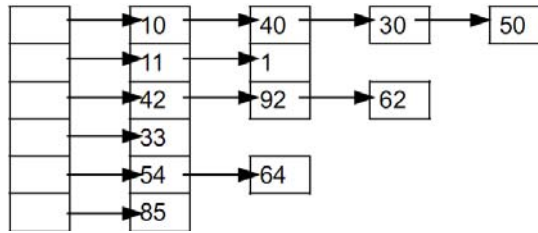
## 9.7. Хеш-таблицы

Хеш-таблица эффективно поддерживает операции сопоставления, т. е. помещения объектов  $x$  в массив размера  $m$  по адресу  $h(x)$ . Наличие дубликатов не допускается и должно обрабатываться внешней логикой. При  $n$  элементах коэффициент загрузки равен  $m/n$ . Хеш-таблицы отличаются правилами изменения размера и разрешения коллизий, т. е. ситуаций, когда несколько  $x$  оказываются в одном и том же месте. В разумных реализациях  $E[\text{время выполнения операции отображения}] = O(1)$ . Невозможно иметь отображение размера  $O(n)$  с амортизированным временем поиска и вставки  $O(1)$  (см. [9.2]). Таким образом, в худшем случае время равно  $O(n)$ . Имейте в виду, что при использовании  $h$  со случайным начальным значением порядок итерации элементов оказывается случайным, что сказывается на вычислении чувствительных к порядку функций — например, вычисления максимума.

Ради эффективности размеры таблицы являются степенями двойки.

## 9.8. Цепочка хеш-таблиц

Каждая ячейка массива — это связанный список с элементами, хешированными до соответствующего индекса. В отличие от других форм представления, массив содержит указатели только на узлы списка, а не на элементы. Это необходимо для сохранения

Рис. 9.1. Структура цепочки хеш-таблицы с  $h(x) = x \% 10$ 

адреса при изменении размера (рис. 9.1). Структура памяти цепочечной хеш-таблицы с целыми элементами от 0 до 9 представлена на рис. 9.2.

```
template<typename KEY, typename VALUE, typename HASHER = EHash<BUHash>,
        typename COMPARATOR = DefaultComparator<KEY> >class ChainingHashTable
{
    int capacity, size; // capacity вычисляется до h
    struct Node
    {
        KEY key;
        VALUE value;
        Node* next;
        Node(KEY const& theKey, VALUE const& theValue): key(theKey),
            value(theValue), next(0) {}
    }
    ** table;
    Freelist<Node> f;
    HASHER h;
    COMPARATOR c;
    enum{MIN_CAPACITY = 8}; // для эффективности требуется
                           // размер не менее 8
    void allocateTable()
    {
        h = HASHER(capacity);
        table = new Node*[capacity];
        for(int i = 0; i < capacity; ++i) table[i] = 0;
    }
public:
    typedef Node NodeType;
    int getSize(){return size;}
    ChainingHashTable(int initialCapacity = 8, COMPARATOR const&
        theC = COMPARATOR()): capacity(nextPowerOfTwo(max<int>(initialCapacity,
            MIN_CAPACITY))), c(theC), h(capacity), size(0) {allocateTable();}
    ChainingHashTable(ChainingHashTable const& rhs): capacity(rhs.capacity),
        size(rhs.size), h(rhs.h), table(new Node*[capacity]), c(rhs.c)
    { // простое копирование без сжатия
        for(int i = 0; i < capacity; ++i)
        {
            table[i] = 0;
            Node** target = &table[i];
            for(Node* j = rhs.table[i]; j; j = j->next)
            {
                *target = new(f.allocate())Node(*j);
            }
        }
    }
};
```

```

        target = &(*target)->next;
    }
}
}
ChainingHashTable& operator=(ChainingHashTable const& rhs)
{return genericAssign(*this, rhs);}
~ChainingHashTable(){delete[] table;}
};

```

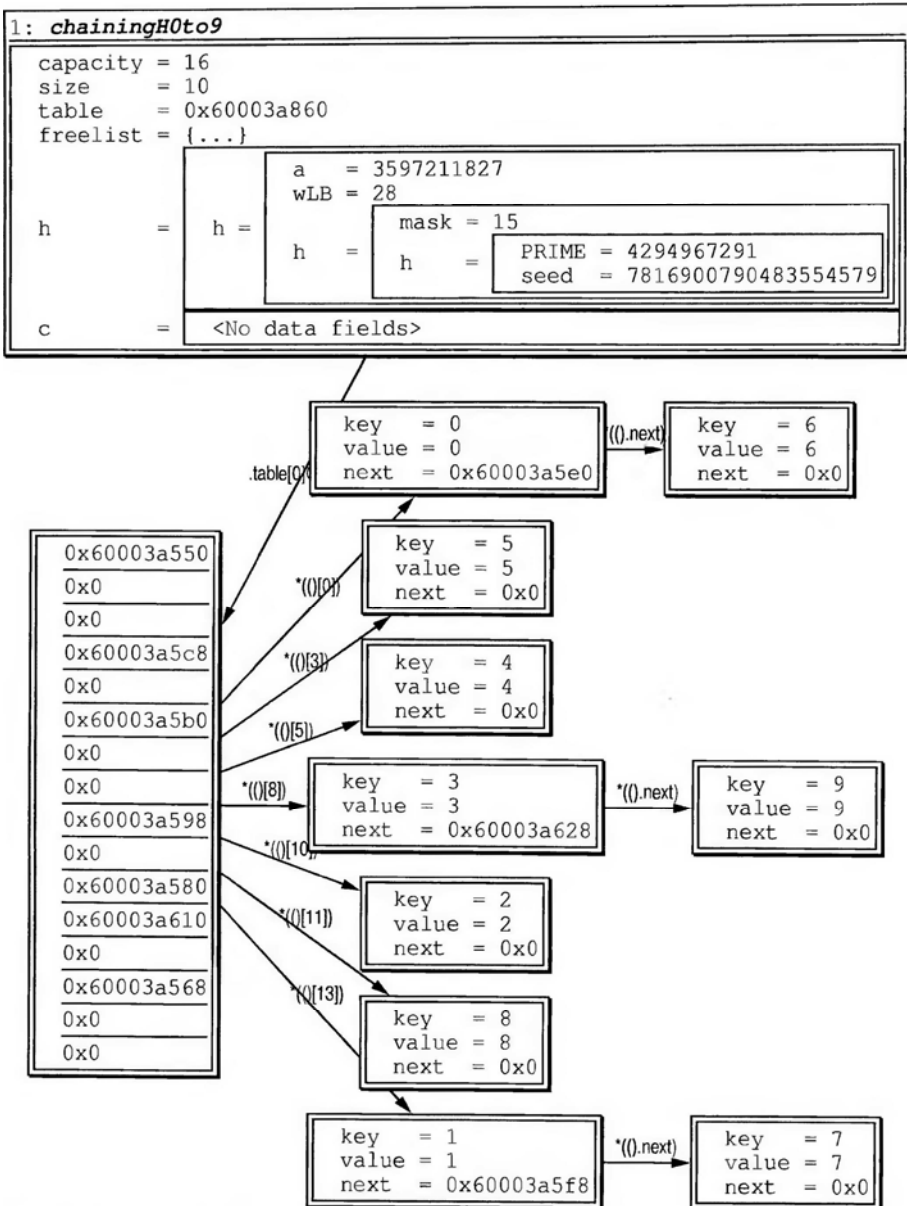


Рис. 9.2. Структура памяти цепочечной хеш-таблицы с целыми элементами от 0 до 9

Операции сопоставления хешируют ключ, линейно ищут элемент с ключом и обновляют список, если выполняется вставка/удаление:

```
Node** findPointer(KEY const& key)
{ // для повторного использования кода берется
  // указатель на указатель узла
  Node** pointer = &table[h(key)];
  for(*pointer && !c.isEqual((*pointer)->key, key);
    pointer = &(*pointer)->next);
  return pointer; // если значение не найдено, возвращаем
                  // указатель на следующий узел после последнего
}
// возврат указателя разрешен из-за постоянства узла
Node* findNode(KEY const& key){return *findPointer(key);}
VALUE* find(KEY const& key)
{
  Node* next = findNode(key);
  return next ? &next->value : 0;
}
```

Операция вставки обновляет значение либо добавляет его, а также удваивает размер таблицы, если она переполняется. Если хеш-функция универсальна, по теореме А с коэффициентом загрузки  $O(1)$   $E[\text{размер списка}] = O(1)$ , поэтому  $E[\text{время выполнения операции отображения}] = O(1)$ :

```
Node* insert(KEY const& key, VALUE const& value)
{
  Node** pointer = findPointer(key);
  if(*pointer)
  { // если значение существует, обновляем его
    (*pointer)->value = value;
    return *pointer;
  }
  else
  {
    Node* node = *pointer = new(f.allocate())Node(key, value);
    if(++size >= capacity) resize(); // операция > дала бы
                                     // увеличение в 4 раза
    return node;
  }
}
```

При изменении размера алгоритм перебирает старую таблицу и копирует все ее узлы в новую таблицу удвоенного размера.  $E[\text{время выполнения}] = O(n)$ , и изменение размера возникает лишь через каждые  $O(n)$  вставок, что дает амортизированное ожидаемое время вставки  $O(1)$ . После изменения размера  $a = 0,5$ , а указатели узлов не меняются:

```
void resize()
{
  int oldCapacity = capacity;
  Node** oldTable = table;
  capacity = nextPowerOfTwo(size * 2);
  allocateTable();
```

```

for(int i = 0; i < oldCapacity; ++i)
    for(Node* j = oldTable[i], *tail; j; j = tail)
    {
        tail = j->next;
        j->next = 0;
        *findPointer(j->key) = j; // добавление узла
    }
delete[] oldTable;
}

```

Операция удаления вырезает найденный узел и изменяет размер таблицы, если  $\alpha < 0,1$ , что кажется разумным компромиссом между использованием памяти и частотой изменения размера:

```

void remove(KEY const& key)
{
    Node** pointer = findPointer(key);
    Node* i = *pointer;
    if(i)
    { // найдено
        *pointer = i->next;
        f.remove(i);
        if(--size < capacity * 0.1 && size * 2 >= MIN_CAPACITY) resize();
    }
}

```

Время выполнения также амортизируется к  $O(1)$ . Цикл перебирает массив, пропуская пустые ячейки. Перебор всех элементов выполняется за  $O(n)$ . Поддерживается только прямая итерация, потому что порядка у элементов нет:

```

class Iterator
{
    int i; // индекс текущей ячейки
    Node* node; // узел в ячейке
    ChainingHashTable& t;
    friend ChainingHashTable;
    void advanceCell() // если находимся в нулевом узле, но не в конце,
                       // пробуем следующую ячейку
    {if(!node) while(i + 1 < t.capacity && !(node = t.table[++i]));}
    Iterator(ChainingHashTable& theHashTable, int theI = -1): i(theI),
        node(0), t(theHashTable) {advanceCell();}
public:
    Iterator& operator++()
    {
        assert(node);
        node = node->next;
        advanceCell();
        return *this;
    }
    NodeType& operator*(){assert(node); return *node;}
    NodeType* operator->(){assert(node); return node;}
    bool operator==(Iterator const& rhs) const{return node == rhs.node;}
};

```

```

Iterator begin() {return Iterator(*this); }
Iterator end()
{
    Iterator result(*this, capacity);
    return result;
}

```

При случайной функции  $h$   $\Pr(\text{элемент попадает в заданную ячейку}) = 1/m$ . Пусть  $k =$  размер списка в данной ячейке после вставки элементов, а именно  $\text{binomial}(k, 1/m, n) \approx \text{Poisson}(k, a)$ . Формула плотности вероятности для распределения Пуассона:  $\frac{a^k}{k!e^a}$  (см. [9.17]). Тогда:

- ◆  $E[\text{размер списка}] = a$ ;
- ◆  $E[\% \text{ ячеек с } k \text{ элементами}] = \text{PoissonPDF}(k, a)$ , что для  $a = 1$  равняется  $\approx 0,37$  для  $k = 0$  и  $1$ ,  $0,18$  при  $2$ ,  $0,06$  при  $3$  и т. д.;
- ◆  $Pr(\text{размер списка} = l \geq k) =$

$$\sum_{k \leq l \leq \infty} \text{PoissonPDF}(l, a) = \sum_{k \leq l \leq \infty} \frac{a^l}{l!e^a} = \frac{a^k}{k!} \sum_{0 \leq l \leq \infty} \frac{a^l}{l!e^a} = \frac{a^k}{k!} < 10^{-12} \quad \text{для } k = 15 \text{ и } a = 1.$$

За счет более высоких постоянных коэффициентов как в использовании памяти, так и в ожидаемом времени выполнения, разрешение коллизий цепочек (обсуждается позже) позволяет использовать динамически отсортированные последовательности вместо связанных списков, чтобы гарантировать время выполнения  $O(\lg(n))$ . Это кажется нецелесообразным, потому что сверхконстантное время выполнения экспоненциально маловероятно при хорошем выборе функции  $h$ .

## 9.9. Хеш-таблица с линейным зондированием

Цепочки в хеш-таблицах обладают недостатками: затраты места на хранение ссылок, использование свободных списков и неуклюжая итерация. Линейное зондирование позволяет избежать этих проблем за счет использования для отметки занятых ячеек массива элементов и массива логических значений. Элементы, у которых возникла коллизия, перемещаются в следующую свободную ячейку, как книги на библиотечных полках. Зондирование выполняется без проверки кеша (см. главу 13. Алгоритмы для работы с внешней памятью) и допускает удаление (рис. 9.3). Структура памяти хеш-таблицы линейного зондирования с целыми элементами от 0 до 9 представлена на рис. 9.4.

1	10	↪
1	11	
1	40	↩
0		
1	54	↪
1	64	

**Рис. 9.3.** Структура хеш-таблицы линейного зондирования: левый столбец содержит логические значения, а правый — сами элементы

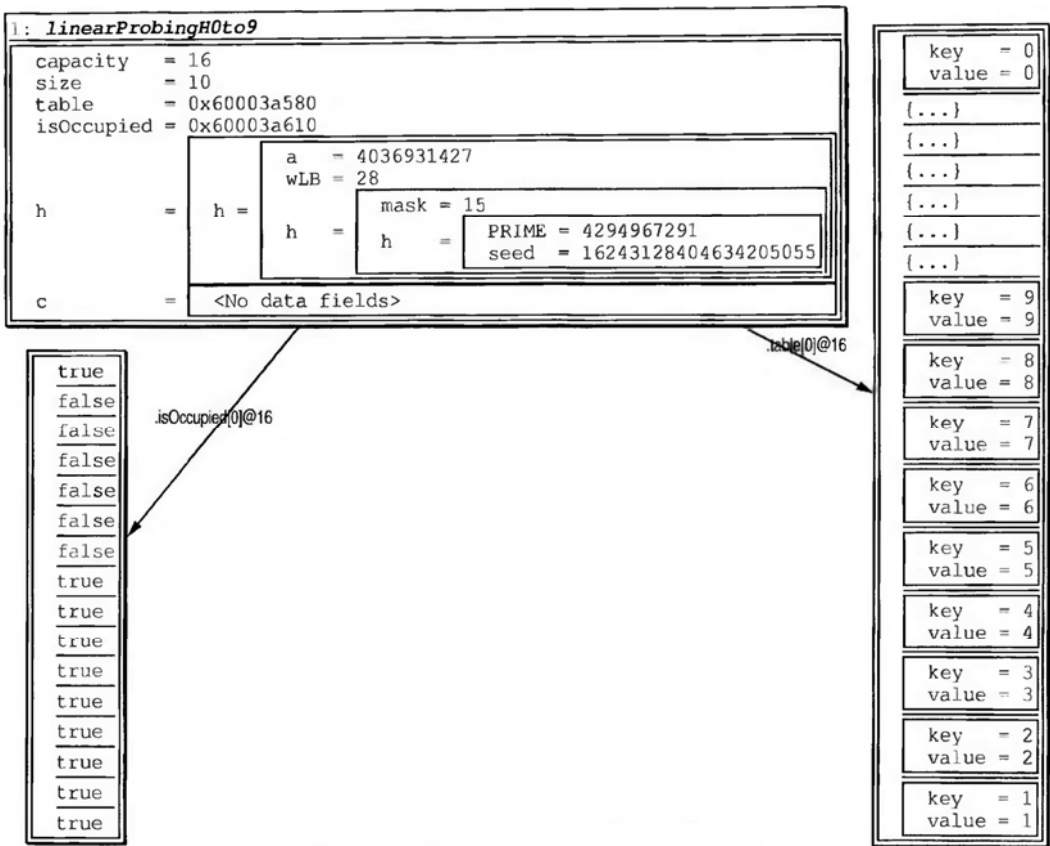


Рис. 9.4. Структура памяти хеш-таблицы линейного зондирования с целыми элементами от 0 до 9

```
template<typename KEY, typename VALUE, typename HASHER = EHash<BUHash>,
typename COMPARATOR = DefaultComparator<KEY> > class LinearProbingHashTable
{
    int capacity, size; // capacity определяется до h
    typedef KVPair<KEY, VALUE> Node;
    Node* table;
    bool* isOccupied;
    HASHER h;
    COMPARATOR c;
    enum{MIN_CAPACITY = 8}; // для эффективности требуется
                           // размер не менее 8

    void allocateTable()
    { // создаем незанятую таблицу размера capacity
        h = HASHER(capacity);
        size = 0;
        table = rawMemory<Node>(capacity);
        isOccupied = new bool[capacity];
        for(int i = 0; i < capacity; ++i) isOccupied[i] = false;
    } // помощник для удаления неиспользуемой таблицы
    static void cleanUp(Node* theTable, int theCapacity, bool* isOccupied)
```

```

{ // уничтожение занятых узлов и высвобождение массивов
    for(int i = 0; i < theCapacity; ++i)
        if(isOccupied[i]) theTable[i].~Node();
    rawDelete(theTable);
    delete[] isOccupied;
}
public:
    typedef Node NodeType;
    int getSize(){return size;}
    LinearProbingHashTable(int initialCapacity = 8, COMPARATOR const& theC =
        COMPARATOR(), c(theC), h(capacity) {allocateTable();}
    LinearProbingHashTable(LinearProbingHashTable const& rhs):
        capacity(rhs.capacity), h(rhs.h), size(rhs.size), c(rhs.c),
        isOccupied(new bool[capacity]), table(rawMemory<Node>(capacity))
    { // копия просто отражает исходный код, не пытаюсь его сжать
        for(int i = 0; i < capacity; ++i)
            if(isOccupied[i] = rhs.isOccupied[i]) table[i] = rhs.table[i];
    }
    LinearProbingHashTable& operator=(LinearProbingHashTable const& rhs)
        {return genericAssign(*this, rhs);}
    ~LinearProbingHashTable(){cleanUp(table, capacity, isOccupied);}
};

```

Операции отображения находят индекс  $i$  ячейки, где должен быть элемент. Если есть другой элемент, алгоритм переходит в ячейку  $(i + 1) \% m$ , пока не найдется пустая;

```

int findNode(KEY const& key)
{ // поиск ячейки, в которой должен был бы находиться ключ,
  // если бы он был вставлен
  int cell = h(key);
  for(;isOccupied[cell] && !c.isEqual(key, table[cell].key);
      cell = (cell + 1) % capacity);
  return cell;
}
VALUE* find(KEY const& key)
{
    int cell = findNode(key);
    return isOccupied[cell] ? &table[cell].value : 0;
}

```

Операция вставки помещает элемент в следующую доступную ячейку или обновляет значение найденного элемента. При случайном значении  $h$   $E$  [количество проб для успешной находки] =  $\frac{1}{2} \left( 1 - \frac{1}{1-a} \right)$ . Для неудачного поиска или вставки получаем

$\frac{1}{2} \left( 1 - \frac{1}{(1-a)^2} \right)$  (см. [9.1]). Например, при  $a = 0,8$  получаем 3 и 13. Значение не возвращается, поскольку будущее изменение размера сделает его недействительным:

```

void insert(KEY const& key, VALUE const& value)
{
    int cell = findNode(key);

```



```

    if(isOccupied[cell]) table[cell].value = value; // обновление
    else
    { // вставка
        new(&table[cell])Node(key, value);
        isOccupied[cell] = true;
        if(++size > capacity * 0.8) resize(); // изменение размера при достижении  $\alpha$ 
    }
}

```

Операция изменения размера при вставке и удалении амортизируется временем  $O(1)$ , как и в цепочке. Изменение размера делает значение  $\alpha$  равным 0,5 и снова вставляет все занятые элементы:

```

void resize()
{
    int oldCapacity = capacity;
    Node* oldTable = table;
    bool* oldIsOccupied = isOccupied;
    capacity = nextPowerOfTwo(size * 2);
    allocateTable();
    for(int i = 0; i < oldCapacity; ++i) // повторная вставка
        if(oldIsOccupied[i]) insert(oldTable[i].key, oldTable[i].value);
    cleanUp(oldTable, oldCapacity, oldIsOccupied); // удаление старой таблицы
}

```

Операция удаления удаляет найденный элемент и, поскольку эта операция может превратить поиск следующих элементов, пока не найдется незанятая ячейка, выполняется изменение размера, если  $\alpha < 0,1$ . Время выполнения амортизируется как  $O(1)$ , поскольку  $E$  [длина цепи] в худшем случае квадратична по длине цепи:

```

void destroy(int cell)
{
    table[cell].~Node();
    isOccupied[cell] = false;
    --size;
}

void remove(KEY const& key)
{
    int cell = findNode(key);
    if(isOccupied[cell])
    { // повторная вставка последующих узлов
        // в цепочку найденных значений
        destroy(cell); // удаление элемента
        if(size < capacity * 0.1 && size * 2 >= MIN_CAPACITY) resize();
        else // повторная вставка
            while(isOccupied[cell = (cell + 1) % capacity])
            {
                Node temp = table[cell];
                destroy(cell); // уничтожение элемента
                insert(temp.key, temp.value); // повторная вставка
            }
    }
}

```

Прохождение по всем элементам занимает  $O(n)$  времени и эффективно кешируется:

```
class Iterator
{
    int i; // индекс текущей ячейки
    LinearProbingHashTable& t;
    friend LinearProbingHashTable;
    void advance(){while(i < t.capacity && !t.isOccupied[i]) ++i;}
    Iterator(LinearProbingHashTable& theHashTable, int theI = 0): i(theI),
        t(theHashTable) {advance();}
public:
    Iterator& operator++()
    {
        ++i;
        advance();
        return *this;
    }
    NodeType& operator*()const{assert(i < t.capacity); return t.table[i];}
    NodeType* operator->()const{assert(i < t.capacity);return &t.table[i];}
    bool operator==(Iterator const& rhs)const{return i == rhs.i;}
};
Iterator begin(){return Iterator(*this);}
Iterator end()
{
    Iterator result(*this, capacity);
    return result;
}
```

Универсальная хеш-функция  $h$  не гарантирует  $E[\text{время выполнения операции карты}] = O(1)$ , потому что неявные элементы списка, хешированные в одну и ту же ячейку, не являются независимыми. Но объединение случайных факторов универсального  $h$  и данных обычно дает характеристики случайного  $h$  (см. [9.11]). Можно заменить логический массив более медленным битовым вектором — по моим приблизительным оценкам, если ключ и элемент занимают 8 байтов, это экономит  $\approx 10\%$  памяти при замедлении  $\approx 50\%$ .

## 9.10. Оценка времени

Здесь использовалась 32-битная машина, а 64-битной машине больше подойдет 64-битная функция  $h$  (рис. 9.5).

В зависимости от таймингов по умолчанию следует использовать E-BU. М-хеши работают медленнее, особенно для целых чисел. Сравнение цепочки и линейного зондирования:

- ◆ цепочка сохраняет постоянство элементов — стандарт C++ требует этого для неупорядоченного отображения;
- ◆ линейное зондирование использует для мелких элементов меньше памяти;
- ◆ в цепочке лучше выполняется разрешение коллизий и гарантируется время  $O(1)$  с универсальной хеш-функцией  $h$ ;

Hasher	Int	Str10	Ch	+-	Min	Max	LP	+-	Min	Max	Ch	+-	Min	Max	LP	+-	Min	Max
E-BU	0.63	13.11	0.36	0.04	0.25	1.56	0.21	0.03	0.13	1.218	0.18	0	0.17	0.25	0.218	0.01	0.187	0.469
E-B-Prime	6	12.92	0.45	0.02	0.36	1.19	0.33	0.03	0.22	1.485	0.18	0	0.17	0.2	0.219	0.01	0.187	0.515
E-B-Prime2	6	21.6	0.44	0.01	0.31	1	0.36	0.06	0.22	3.14	0.3	0	0.28	0.34	0.334	0.03	0.296	1.985
E-B-FNV	2.12	4.015	0.35	0	0.33	0.44	0.15	0	0.14	0.172	0.09	0	0.08	0.11	0.1	0	0.093	0.11
E-M-FNV	8.84	4.953	0.45	0	0.44	0.47	0.29	0	0.27	0.312	0.1	0	0.09	0.11	0.111	0	0.094	0.125
E-B-FNV64	2.11	4.031	0.34	0	0.33	0.36	0.15	0	0.14	0.157	0.09	0	0.08	0.11	0.1	0	0.093	0.11
E-B-X64	3.06	11.33	0.29	0	0.27	0.34	0.17	0	0.16	0.172	0.16	0	0.16	0.17	0.183	0	0.171	0.219
E-B-Table	1.89	4.094	0.48	0	0.47	0.5	0.45	0	0.44	0.469	0.08	0	0.08	0.09	0.113	0	0.109	0.125

**Рис. 9.5.** Результаты теста — время в секундах для  $10^6$  и  $10^5$  вставок соответственно объектов `int` и `Struct10` из 10 `int`, и в 10 раз больше операций поиска. Операции хеширования объектов `int` и `Struct10` выполнялись соответственно  $10^9$  и  $10^8$  раз. Выделяются лучшие и почти лучшие результаты. Тесты хеш-таблиц выполнялись 100 раз, и результаты — это средние значения. «+-» представляет собой стандартную ошибку 95%-ного доверительного интервала для средних значений. Кроме того, указываются минимальные/максимальные значения

- ◆ качество распределителя памяти и использование свободного списка влияют на скорость цепочки;
- ◆ линейное зондирование быстрее выполняет итерацию.

Используйте цепочку по умолчанию и линейное зондирование для мелких элементов, чтобы сократить использование памяти.

## 9.11. Фильтр Блума

Фильтр Блума поддерживает только вставку с вероятностью правильности `isInserted`. У него фиксированный размер, но зато он занимает гораздо меньше места, чем хеш-таблица. Это полезно, например, для уменьшения объема памяти словарей при проверке орфографии. Задачи, в которых он нужен, встречаются редко, но подробнее о них можно почитать здесь [9.16].

Фильтр состоит из набора битов размера  $m$  и  $k$  хеш-функций (рис. 9.6). Чтобы вставить число  $x$ , установите каждый бит  $h_i(x)$ . Если все биты  $h_i(x)$  установлены, то  $x$  с высокой вероятностью вставляется, а если нет, то нет. Восстановление не поддерживается. Хеширование использует логику набора хеш-функций (см. *разд. 9.6*):

```
template<typename KEY, typename HASHER = EHash<BUHash> >
class BloomFilter
{
    Bitset<unsigned char> items; // определяется до h1 и h2
    HASHER h1, h2;
    int nHashes;
    int hash(int hash1, int hash2, int i)
    {
        if(i == 0) return hash1;
        if(i == 1) return hash2;
        return (hash1 + i * hash2) % items.getSize();
    }
public:
    BloomFilter(int m, int theNHashes = 7): nHashes(theNHashes), items(
        nextPowerOfTwo(m)), h1(items.getSize()), h2(items.getSize())
    {assert(m > 0 && theNHashes > 0);}
```

```

void insert(KEY const& key)
{
    int hash1 = h1(key), hash2 = h2(key);
    for(int i = 0; i < nHashes; ++i) items.set(hash(hash1, hash2, i));
}

bool isInserted(KEY const& key)
{
    int hash1 = h1.hash(key), hash2 = h2.hash(key);
    for(int i = 0; i < nHashes; ++i)
        if(!items[hash(hash1, hash2, i)]) return false;
    return true;
}
};

```

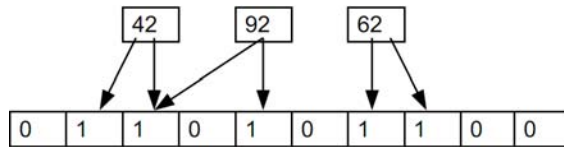


Рис. 9.6. Структура фильтра Блума

Как и в случае с цепочкой,  $\Pr[x \text{ элементов находится в конкретной ячейке}] = \text{PoissonPDF}(x, ka)$ . При  $x = 0$  это  $= e^{-ka} = \Pr(\text{этот бит остается равным } 0)$ . Таким образом,  $\Pr(\text{isInserted равно false}) = p = \Pr[k \text{ заданных битов равны } 1] = (1 - e^{-ka})^k$ ;  $k = \frac{\ln(2)}{a}$ ,

минимизирует  $p$ , поэтому, пренебрегая целым числом  $k$ ,  $p = \frac{0.5 \ln(2)}{a} = e^{-0.48/a}$  и

$m = \frac{-n \ln(p)}{0.48}$ . Учитывая специфику значения  $p$  в этой задаче и оценку максимума  $n$ ,

уравнение нужно решить для  $m$ ,  $a$  и  $k$  в указанном порядке. В линейном зондировании используется  $8(\text{sizeof(Node)} + 1)n/a$  битов, поэтому только в том случае, если это невозможно, следует учитывать, допускает ли фильтр Блума достаточно хорошее  $p$  при допустимом  $m$ .

## 9.12. Примечания по реализации

Самая сложная часть — это API для хеш-функции. Я хотел повторно использовать написанный код, и для этого понадобился C++ версии 11. В противном случае потребуются использовать STL и определить хеш-функцию, переопределяющую любой тип на неуклюжем и подверженном ошибкам языке.

Выбор конкретной хеш-функции  $h$  также сложен. Универсальных функций немного, и их легко выбрать, а неуниверсальные — нет. FNV выбирают в основном из-за его простоты, популярности и исследований производительности, хотя я думаю, что хеш-функция на основе Xorshift потенциально лучше, но еще недостаточно изучена.

Реализация фильтра Блума сама по себе не очень полезна, потому что количество хешей должно рассчитываться автоматически из вероятности успеха, которая является логическим параметром. Но для начала это не самая полезная структура данных.

## 9.13. Комментарии

Вы можете превратить любой генератор псевдослучайных чисел в функцию  $h$ , интерпретируя ключ как начальное значение (хотя их размеры должны совпадать). Многие функции  $h$  подвергаются критике со стороны авторов работ [9.5, 9.6, 9.14]). На сайте [http://www.strchr.com/hash\\_functions](http://www.strchr.com/hash_functions) вы найдете исходный код и большое количество тестов различных  $h$ , но там в основном говорится о времени выполнения и коллизиях. Среди них функция Боба Дженкинса, которая обычно проходит все тесты, сложна и реализуется необъяснимым образом (погуглите, если любопытно). Криптографические хеш-функции, которые превосходят все тесты, использоваться не должны, потому что они очень медленные и сложные. С учетом скорости и простоты функция FNV кажется лучшим неуниверсальным выбором.

В C++ есть особенность: указание  $h$  изменяет тип контейнера, который логически должен зависеть только от данных. Можно исправить это, используя один класс для всех функций  $h$  и добавив возможность выбора.

При линейном зондировании и операции удаления в нем один из повторно вставленных элементов окажется на месте удаленного элемента, если все они не будут хешированы в более позднее место. Способы повышения эффективности:

1. Попробуйте двигать последний элемент в цепочке к местоположению удаленного элемента.
2. Если это не удастся, переместите предпоследний.
3. После успешного перемещения рекурсивно обработайте удаление перемещенного элемента.
4. Остановитесь при перемещении последнего элемента или если перемещение элементов не требуется.

Линейное зондирование — это частный случай более общего понятия *открытой адресации*, которая в учебниках часто описывается как *двойное хеширование*. Идея состоит в том, чтобы улучшить механизм разрешения коллизий при линейном зондировании путем перехода к случайной, а не к следующей ячейке, но используя другую функцию  $h$ . Это позволяет сократить ожидаемое время вставки, но на практике не повышает производительность (см. [9.4]). Выбирать этот алгоритм не следует, поскольку:

- ◆ теряется эффективность кэширования;
- ◆ будет необходимо оценить другую функцию  $h$ , а вызывающая сторона должна ее передавать;
- ◆ теряется операция удаления, остается лишь слабое удаление (см. главу 12. *Разные алгоритмы и методы*).

В *квадратичном зондировании* (см. [9.1]) используется детерминированный размер шага, что позволяет снизить стоимость неудобных операций, связанных с использованием другой хеш-функции  $h$ . Но прочие проблемы это не решает, и значение  $a$  должно быть  $< 0,5$ .

*Кукушкино хеширование* поддерживает операции поиска и удаления за  $O(1)$  путем хеширования элемента в одном из двух возможных местоположений, выбранных двумя независимыми  $h$ . Но вставка, несмотря на ожидаемое время  $O(1)$  для  $h$  с некоторыми

теоретическими гарантиями (функция должна быть лучше, чем универсальная, — см. [9.10]), не имеет наихудшего случая и может потребовать перестройки таблицы. В результате хеширование становится опасным и неуклюжим на практике, потому что вызывающая сторона не всегда может передать функцию  $h$  с нужными свойствами, а ошибки вроде несовершенства генератора случайных чисел могут привести к катастрофе. В моих экспериментах хороший результат дала только начальная версия хеш-функции Xorshift (т. е. с начальной суммой в виде запомненного случайного числа).

*Идеальное хеширование* (см. [9.1]) создает двухуровневые хеш-таблицы без коллизий для фиксированного  $n$ . Но его преимущество по сравнению с цепочкой и линейным зондированием неясно, а использование памяти и время построения у него не лучшие.

Можно попытаться улучшить фильтр Блума несколькими способами, но максимум что удастся — это получить более высокую эффективность при гораздо более сложной реализации (см. [9.12], а также [9.9]).

## 9.14. Советы по дополнительной подготовке

- ◆ Реализуйте более эффективную (и более сложную) операцию удаления для линейного зондирования. Сравните производительность, чтобы увидеть, стоит ли увеличение сложности полученной выгоды.
- ◆ Сравните производительность с типом `unordered_map` в C++ 11.

## 9.15. Список рекомендуемой литературы

- 9.1. Cormen T. H., Leiserson C. E., Rivest R. L., & Stein C. (2009). *Introduction to Algorithms*. MIT Press.
- 9.2. Dietzfelbinger M., Karlin A., Mehlhorn K., Meyer auf der Heide F., Rohnert H., & Tarjan R. E. (1994). Dynamic perfect hashing: upper and lower bounds. *SIAM Journal on Computing*, 23(4), 738–761.
- 9.3. Dietzfelbinger M., Hagerup T., Katajainen J., & Penttonen M. (1997). A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1), 19–51.
- 9.4. Heileman G. L., & Luo W. (2005). How caching affects hashing. In *ALENEX/ANALCO* (pp. 141–154). SIAM.
- 9.5. Henke C., Schmoll C., & Zseby T. (2008). Empirical evaluation of hash functions for multipoint measurements. *ACM SIGCOMM Computer Communication Review*, 38(3), 39–50.
- 9.6. Henke C., Schmoll C., & Zseby T. (2009). Empirical evaluation of hash functions for packetid generation in sampled multipoint measurements. In *Passive and Active Network Measurement* (pp. 197–206). Springer.
- 9.7. Kirsch A., & Mitzenmacher M. (2008). Less hashing, same performance: Building a better Bloom filter. *Random Structures & Algorithms*, 33(2), 187–218.
- 9.8. Mehlhorn K., & Sanders P., Dietzfelbinger M., & Dementiev R. (2019). *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox*. Springer.
- 9.9. Mehta D. P., & Sahni S. (Eds). (2018). *Handbook of Data Structures and Applications*. CRC.
- 9.10. Mitzenmacher M. (2009). Some open questions related to cuckoo hashing. In *Algorithms-ESA 2009* (pp. 1–10). Springer.
- 9.11. Mitzenmacher M., & Vadhan S. (2008). Why simple hash functions work: exploiting the entropy in a data stream. In *Proceedings of the Nineteenth Annual ACM–SIAM Symposium on Discrete Algorithms* (pp. 746–755). SIAM.

- 9.12. Putze F., Sanders P., & Singler J. (2009). Cache-, hash-, and space-efficient Bloom filters. *Journal of Experimental Algorithmics (JEA)*, 14, 4.
- 9.13. Tarkoma S., Rothenberg C. E., & Lagerspetz E. (2012). Theory and practice of Bloom filters for distributed systems. *Communications Surveys & Tutorials, IEEE*, 14(1), 131–155.
- 9.14. Valloud A. (2008). Hashing in Smalltalk: Theory and Practice.
- 9.15. Wikipedia (2018a). Fowler–Noll–Vo hash function. [https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo\\_hash\\_function](https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function). Accessed July 22, 2018.
- 9.16. Wikipedia (2018b). Lagrange's theorem (number theory). [https://en.wikipedia.org/wiki/Lagrange%27s\\_theorem\\_\(number\\_theory\)](https://en.wikipedia.org/wiki/Lagrange%27s_theorem_(number_theory)). Accessed July 22, 2018.
- 9.17. Wikipedia (2018c). Poisson distribution. [https://en.wikipedia.org/wiki/Poisson\\_distribution](https://en.wikipedia.org/wiki/Poisson_distribution). Accessed July 29, 2018.
- 9.18. Wikipedia (2018d). Universal hashing. [https://en.wikipedia.org/wiki/Universal\\_hashing](https://en.wikipedia.org/wiki/Universal_hashing). Accessed July 22, 2018.
- 9.19. Wikipedia (2019). Bloom filter. [https://en.wikipedia.org/wiki/Bloom\\_filter](https://en.wikipedia.org/wiki/Bloom_filter). Accessed December 21, 2019.

## 10. Приоритетные очереди

### 10.1. Введение

На курсах по алгоритмам обычно изучают бинарные кучи. Мы поговорим здесь об очень полезной операции смены ключа, которая, к сожалению, по умолчанию не предусмотрен. В дополнение к этому я хотел бы убедить вас в том, что более сложные и быстрые очереди с приоритетом, такие как куча Фибоначчи или парная куча, которые тоже проходят на курсах, на практике бесполезны.

### 10.2. API

*Очередь с приоритетом может:*

- ♦ вставлять элемент с заданным приоритетом;
- ♦ находить элемент с наименьшим приоритетом;
- ♦ удалять элемент с наименьшим приоритетом.

Это стандартный интерфейс такой очереди. Но есть и другие операции:

- ♦ изменение приоритета элемента — традиционно это называется *изменением ключа*, которое необходимо для многих алгоритмов;
- ♦ удаление элемента — по сути, операция сводится к изменению ключа на наименьшее возможное значение и удалению элемента с наименьшим приоритетом;
- ♦ объединение двух очередей с приоритетом — в некоторых реализациях работает очень быстро, но используется редко.

Ограничения функциональности наводят на мысль о более эффективной реализации по сравнению с динамически отсортированной последовательностью, которая поддерживает неуникальные ключи. Приоритетная очередь может выполнять сортировку, удаление минимума и вставку за амортизированное время  $O(\lg(n))$ .

### 10.3. Бинарная куча

Используйте вектор для представления двоичного дерева (рис. 10.1), где для узла с индексом  $i$  справедливо следующее:

- ♦ родитель =  $(i - 1)/2$ ;
- ♦ левый потомок =  $2i + 1$ ;
- ♦ правый потомок =  $2i + 2$ .

Элемент любого узла меньше или равен элементам в его дочерних элементах.



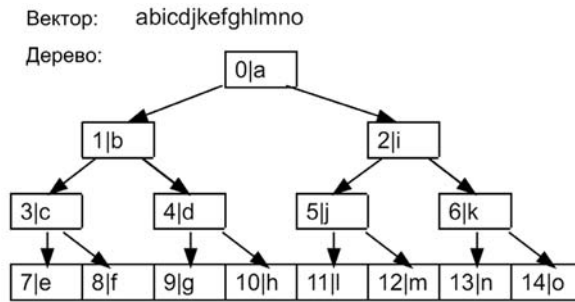


Рис. 10.1. Интерпретация вектора кучи в виде дерева. Числа являются индексами массива

О перемещении элемента может быть сообщено объекту, выполняющему индексацию, и это будет рассмотрено далее в этой главе. Я не видел, чтобы это расширение было представлено или реализовано где-либо еще, но без него из-за отсутствия постоянства элемента операция смены ключа бесполезна:

```

template<typename ITEM>
struct ReportDefault{void operator()(ITEM& item, int i){}};
template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM>,
typename REPORTER = ReportDefault<ITEM> > class Heap
{
    REPORTER r;
    int getParent(int i) const{return (i - 1)/2;}
    int getLeftChild(int i) const{return 2 * i + 1;}
    Vector<ITEM> items;
public:
    COMPARATOR c;
    Heap(COMPARATOR const& theC = COMPARATOR(), REPORTER const&
        theReporter = REPORTER()): r(theReporter), c(theC) {}
    bool isEmpty() const{return items.getSize() == 0;}
    int getSize() const{return items.getSize();}
    ITEM const& getMin() const
    {
        assert(!isEmpty());
        return items[0];
    }
    ITEM const& operator[](int i) const
    { // случайный доступ полезен
        assert(i >= 0 && i < items.getSize());
        return items[i];
    }
};

```

Функция `moveUp` меняет элемент со своим родителем, пока родитель меньше элемента, сообщая обо всех перемещениях:

```

void moveUp(int i)
{
    ITEM temp = items[i];
    for(int parent; i > 0 && c(temp, items[parent = getParent(i)]);
        i = parent) r(items[i] = items[parent], i);
}

```

```

    r(items[i] = temp, i);
}

```

Функция `moveDown` меняет элемент с наименьшим потомком, пока таковой есть, сообщая обо всех перемещениях:

```

void moveDown(int i)
{
    ITEM temp = items[i];
    for(int child; (child = getLeftChild(i)) < items.getSize(); i = child)
    { // поиск наименьшего потомка
        int rightChild = child + 1;
        if(rightChild < items.getSize() && c(items
            [rightChild], items[child])) child = rightChild;
        // замена на наименьшего потомка, если таковой имеется
        if(!c(items[child], temp)) break;
        r(items[i] = items[child], i);
    }
    r(items[i] = temp, i);
}

```

Метод вставки добавляет элемент и перемещает его вверх:

```

void insert(ITEM const& item)
{
    items.append(item);
    moveUp(items.getSize() - 1);
}

```

Чтобы удалить элемент  $i$ :

1. Замените его копией последнего элемента.
2. Переместите копию вниз (она не коснется последнего элемента).
3. Удалите последний элемент.

Возвращаемое значение местоположения  $-1$  указывает на то, что элемент был удален. При генерации сообщений о замене имейте в виду, что элемент может оказаться удален:

```

ITEM deleteMin(){return remove(0);}
ITEM remove(int i)
{
    assert(i >= 0 && i < items.getSize());
    ITEM result = items[i];
    r(result, -1);
    if(items.getSize() > i)
    { // не последний элемент
        items[i] = items.lastItem();
        r(items[i], i); // сообщить о перемещении
        moveDown(i);    // не трогаем последний элемент
    }
    items.removeLast();
    return result;
}

```

Операция *изменения ключа* элемента  $i$  вносит изменение в элемент и перемещает его вверх, если новый ключ меньше старого ключа, и вниз — в противном случае. Чтобы заменить минимальный элемент, вызовите функцию `changeKey(0, item)`:

```
void changeKey(int i, ITEM const& item)
{
    assert(i >= 0 && i < items.getSize());
    bool decrease = c(item, items[i]);
    items[i] = item;
    decrease ? moveUp(i) : moveDown(i);
}
```

Все операции выполняются за время  $O(\text{высота}) = O(\lg(n))$ , но вставка и удаление амортизируются из-за удвоения вектора. Чтобы объединить несколько куч, вставьте в одну все элементы из остальных (здесь это не реализовано). Если все элементы в конечном итоге будут удалены, стоимость амортизируется до  $O(\lg(n))$ .

Операция `Heapsort` использует кучу для сортировки. Элементы вставляются в кучу и извлекаются в отсортированном порядке с помощью операции удаления минимума. Вы можете запускать кучу непосредственно в несортированном массиве элементов и использовать более быструю массовую вставку, называемую `heapify` (здесь не реализована). Операция выполняется за время  $O(n \lg(n))$ , но с большей константой, чем оптимальные сортировки.

## 10.4. Индексированные кучи

В *индексированных кучах* используются хеш-таблицы с постоянным адресом, связывающие элементы с предоставленными вызывающей стороной дескрипторами для вставки в бинарную кучу. Это полезно, например, для реализации алгоритма кратчайших путей Дейкстры (см. главу 11. *Алгоритмы графов*). Проявив некоторую изобретательность, можно сделать индексированную кучу из стандартных компонентов:

1. Использовать множество кортежей, в которых хранятся значения приоритета и дескриптора, отсортированные сначала по приоритету, а потом по дескриптору.
2. Сопоставить (`map`) дескриптор с приоритетом с использованием хеш-таблицы.
3. Чтобы обновить этот дескриптор, нужно взять сопоставление, получить приоритет, обновить его, затем в множестве удалить старый кортеж и вставить новый.
4. Затем удалить наименьший элемент множества и обновить сопоставление (`map`).

Для вопросов на собеседовании это сгодится (кто-то однажды попросил меня написать модифицированную версию алгоритма Дейкстры), но постоянные коэффициенты можно улучшить, если использовать вместо множества приоритетную очередь. Множество — в отличие от приоритетной очереди — обычно не допускает дублирования ключей, но здесь вторичный ключ гарантирует общую уникальность ключа. Вот тут-то и появляется отчетность — куча сообщает карте, куда она перемещает элементы.

Другой подход, применимый только в особых случаях, состоит в том, чтобы вставлять новый приоритетный элемент, не удаляя старый. Например, для алгоритма Дейкстры (см. главу 11. *Алгоритмы графов*) это работает, потому что старый элемент никогда не удаляется. Но это излишняя сложность и пустая трата памяти.

Основная хитрость реализации индексированной кучи заключается в правильном управлении отчетами. Элемент кучи состоит из реального элемента и указателя на узел хеш-таблицы, содержащий дескриптор в качестве ключа и адрес в куче в качестве значения. Составитель отчетов использует указатели для обновления индексов элементов. Амортизированное время  $O(\lg(n))$  кучи и ожидаемое  $O(1)$  хеш-таблицы приводят к ожидаемому амортизированному  $O(\lg(n))$  всех операций. Структура памяти индексированной кучи с целочисленными элементами от 0 до 3 представлена на рис. 10.2.

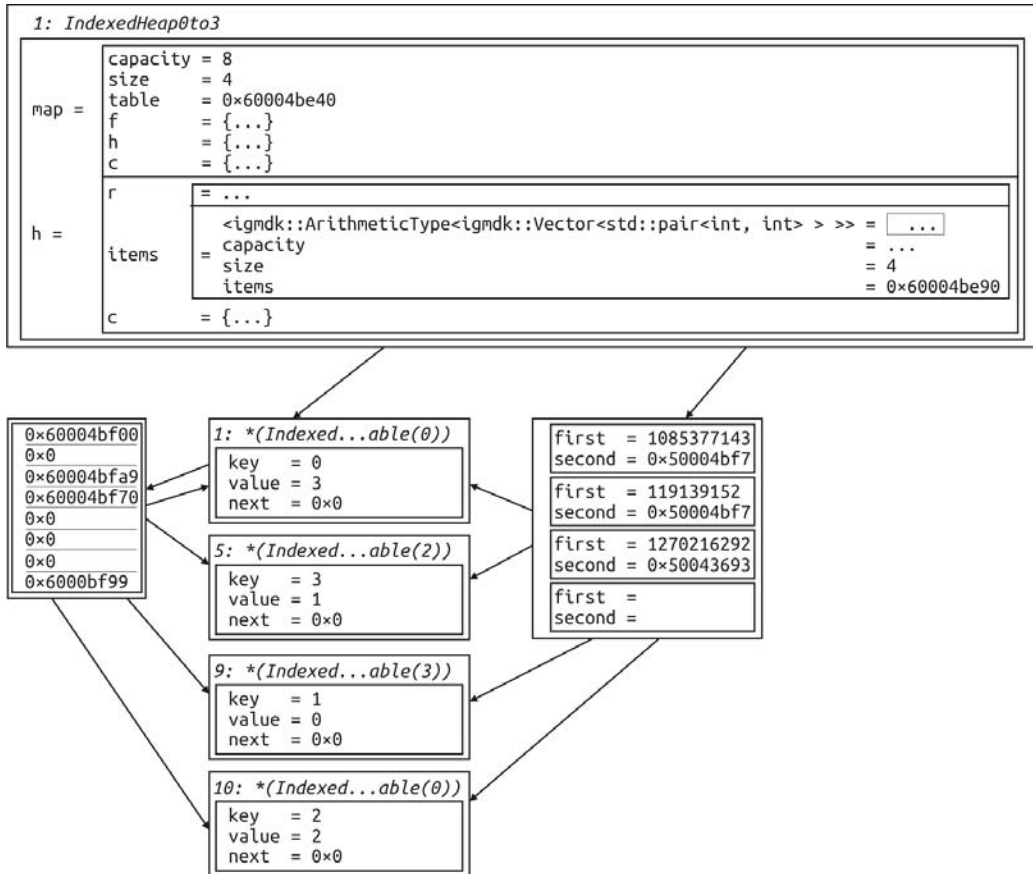


Рис. 10.2. Структура памяти индексированной кучи с целочисленными элементами от 0 до 3

Операции сводятся к вызову соответствующих операций над кучей и обновлению карты, когда составитель отчетов не может выполнить обновление:

```
template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM>,
typename HANDLE = int, typename HASHER = EHash<BUHash> > class IndexedHeap
{
    typedef ChainingHashTable<HANDLE, int, HASHER> MAP;
    MAP map;
    typedef typename MAP::NodeType* POINTER;
    typedef pair<ITEM, POINTER> Item;
    typedef PairFirstComparator<ITEM, POINTER, COMPARATOR> Comparator;
```

```

    struct Reporter
    {void operator()(Item& item, int i){item.second->value = i;}};
    Heap<Item, Comparator, Reporter> h;
public:
    IndexedHeap(COMPARATOR const& theC = COMPARATOR()): h(Comparator(theC)){}
    int getSize()const{return h.getSize();}
    ITEM const* find(HANDLE handle)
    {
        int* index = map.find(handle);
        return index ? &h[*index].first : 0;
    }
    bool isEmpty()const{return h.isEmpty();}
    void insert(ITEM const& item, HANDLE handle)
    {
        assert(!find(handle)); // в противном случае в карте
                               // появится дубликат
        h.insert(Item(item, map.insert(handle, h.getSize())));
    }
    pair<ITEM, HANDLE> getMin()const
    {
        Item temp = h.getMin();
        return make_pair(temp.first, temp.second->key);
    }
    pair<ITEM, HANDLE> deleteMin()
    {
        Item temp = h.deleteMin();
        pair<ITEM, HANDLE> result = make_pair(temp.first, temp.second->key);
        map.remove(temp.second->key);
        return result;
    }
    void changeKey(ITEM const& item, HANDLE handle)
    {
        POINTER p = map.findNode(handle);
        if(p) h.changeKey(p->value, Item(item, p));
        else insert(item, handle);
    }
    void deleteKey(HANDLE handle)
    {
        int* index = map.find(handle);
        assert(index);
        h.remove(*index);
        map.remove(handle);
    }
};

```

Использование вектора в качестве карты более эффективно для дескрипторов с небольшими целыми числами. Его размер динамически увеличивается до максимального значения дескриптора. Индексы в таком векторе — это дескрипторы, а элементы — это места в куче. Значение  $-1$  означает отсутствие данных по этому индексу. Структура памяти векторной индексированной кучи с целочисленными элементами от 0 до 3 представлена на рис. 10.3.



Рис. 10.3. Структура памяти векторной индексированной кучи с целочисленными элементами от 0 до 3

```
template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM> >
class IndexedArrayHeap
{
    Vector<int> map;
    typedef pair<ITEM, int> Item;
    typedef PairFirstComparator<ITEM, int, COMPARATOR> Comparator;
    struct Reporter
    {
        Vector<int>& pmap;
        Reporter(Vector<int>& theMap): pmap(theMap) {}
        void operator()(Item& item, int i){pmap[item.second] = i;}
    };
    Heap<Item, Comparator, Reporter> h;
public:
    typedef Item ITEM_TYPE;
    IndexedArrayHeap(COMPARATOR const& theC = COMPARATOR()):
        h(Comparator(theC), Reporter(map)) {}
    int getSize()const{return h.getSize();}
    ITEM const* find(int handle)
    {
        assert(handle >= 0);
        return handle >= map.getSize() || map[handle] == -1 ? 0 :
            &h[map[handle]].first;
    }
    bool isEmpty()const{return h.isEmpty();}
    void insert(ITEM const& item, int handle)
    {
        assert(handle >= 0);
```

```

    if(handle >= map.getSize())
        for(int i = map.getSize(); i <= handle; ++i) map.append(-1);
    h.insert(Item(item, handle));
}
pair<ITEM, int> const& getMin() const{return h.getMin();}
pair<ITEM, int> deleteMin()
{
    Item result = h.deleteMin();
    map[result.second] = -1;
    return result;
}
void changeKey(ITEM const& item, int handle)
{
    assert(handle >= 0);
    if(handle >= map.getSize() || map[handle] == -1) insert(item, handle);
    else h.changeKey(map[handle], Item(item, handle));
}
void deleteKey(int handle)
{
    assert(handle >= 0 && handle < map.getSize());
    int pointer = map[handle];
    assert(pointer != -1);
    h.remove(pointer);
    map[handle] = -1;
}
};

```

Индексированная куча (в любой реализации) позволяет получить чистое решение для следующей задачи, которую задают на собеседованиях. Есть марафонская дорожка с  $m$  датчиками и  $n$  бегунами, которые стартуют одновременно (предположим, что дорожка достаточно широка для этого). Датчик отправляет событие, состоящее из его идентификатора и идентификатора бегуна, когда бегун пробегает мимо. Система ведет список текущих  $k$  ведущих бегунов по ходу забега и может выводить его по требованию.

Простое решение состоит в том, чтобы вести упорядоченный список бегунов, прошедших определенный датчик (предположив, что несколько бегунов не могут подойти к одному и тому же датчику в одно и то же время). Когда нужно будет вывести список лидеров, начните с последнего датчика на дорожке, соберите бегунов из его списка, если они есть, и перейдите к следующему датчику, пока не получите нужное количество. Когда бегуны удаляются из всех списков, кроме последнего, печать таблицы лидеров занимает время и память  $O(n + m)$ .

Лучшее решение состоит в том, чтобы предположить, что в любой момент нам интересна информация только о  $k$  лучших бегунах ( $k$  выбирается на старте). Приоритетом бегуна является сочетание первого датчика и времени прибытия. Используйте глобальный счетчик для подсчета времени прибытия. Поместите  $k$  лучших на текущий момент бегунов в индексированную кучу. Когда датчик отправляет событие, бегун либо присутствует в числе лучших, либо нет. Потом:

- ◆ если бегун в настоящее время находится в числе лучших, обновите его приоритет;
- ◆ если он только вошел в число лучших, поместите его в кучу и удалите текущего  $k$ -го бегуна.

Печать занимает  $O(k \lg(k))$  за счет создания копии кучи и повторного удаления вершины. Обработка события занимает время  $O(\lg(k))$  и память  $O(n)$ . Можно отбросить коэффициент  $\lg(k)$ , используя структуру данных, состоящую из связанного списка и двух хеш-таблиц, но это слишком сложно для собеседования и непрактично.

## 10.5. Примечания по реализации

Идея индексированной кучи упоминается в работе [10.5], где приводится ограниченная реализация кучи индексированного массива и не упоминается, что можно использовать хеш-таблицы. Функция создания отчетов является оригинальной и позволяет чисто разделить компоненты. Учитывая, что для использования ключей нужно сопоставление, это очевидная, но полезная концепция, хотя и не единственный способ реализации. Я протестировал несколько разных реализаций, прежде чем выбрал решение с отчетами.

## 10.6. Комментарии

Существует множество других приоритетных очередей с различными свойствами. Все они бесполезны на практике, потому что бинарная куча реализуется проще, конкурентоспособна во всех случаях и использует меньше памяти. Основные альтернативы бинарной куче:

- ♦ *Очередь с корзиной* (Bucket queue) — для элементов с приоритетами в диапазоне  $[0, N - 1]$  это массив из  $N$  связанных списков, где список  $i$  содержит элементы с приоритетом  $i$  и целое число, содержащее наименьший индекс непустого списка. Например, вы можете планировать задачи, присваивая им приоритеты в диапазоне  $[1, 10]$ . Для малых  $N$  эта очередь работает быстрее, чем бинарная куча, но тратит гораздо больше памяти. Многие реальные очереди работают аналогичным образом;
- ♦ *Спаренная куча* (Pairing heap) — набор упорядоченных в куче деревьев на основе указателей. Операции уменьшения ключа и слияния занимают время  $O(1)$  — подробности реализации приведены в работе [10.4]. К недостаткам спаренной кучи можно отнести ее неуклюжую реализацию, повышенное использование памяти и большие постоянные коэффициенты (см. [10.2]). Эксперименты с бинарными кучами говорят о том, что ожидаемая стоимость вставки составляет  $O(1)$ , ожидаемая стоимость уменьшения ключа для случайных данных составляет  $O(1)$ , эффекты кеширования и использования меньшего объема памяти значительны, и разница между  $\lg(n)$  и константой невелика, особенно в коротком цикле движения вверх.
- ♦ Все это еще более справедливо для кучи *Фибоначчи* и других сложных куч, построенных на указателях. Они были разработаны как индексированные кучи с легкой операцией изменения ключа. *Очередь Бродала* (см. [10.1]) кажется теоретически лучшим вариантом из этих «слабых куч», которые пытаются сократить амортизированное число операций сравнения.

Для реализации *двусторонней кучи*, поддерживающей вставку и удаление максимального и минимального элемента, используется список с пропусками и неуникальными элементами. Куча *min-max* (см. [10.3]) специально разработана с учетом этих требований и может быть немного более эффективной, но ее сложно реализовать и поддержи-



вать. Еще одним обобщением является *многомерная куча* с элементами, ранжированными по каждому измерению (см. [10.1]). Ее можно реализовать с помощью индексированной кучи для любой координаты, помещая каждый элемент в каждую кучу и используя индексы для удаления из других куч после удаления в одной конкретной.

## 10.7. Список рекомендуемой литературы

- 10.1. Brass P. (2008). Advanced Data Structures. Cambridge University Press.
- 10.2. Bruun A., Edelkamp S., Katajainen J., & Rasmussen J. (2010). Policy-based benchmarking of weak heaps and their relatives. In International Symposium on Experimental Algorithms (pp. 424–435). Springer, Berlin, Heidelberg.
- 10.3. Mehta D. P. & Sahni S. (Eds). (2018). Handbook of Data Structures and Applications. CRC.
- 10.4. Mehlhorn K., & Sanders P., Dietzfelbinger M., & Dementiev R. (2019). Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox. Springer.
- 10.5. Sedgewick R., & Wayne K. (2011). Algorithms. Addison-Wesley.

# 11. Алгоритмы графов

## 11.1. Введение

На основе алгоритмов графов весьма удобно придумывать вопросы для собеседований, особенно, если речь идет о поиске в глубину и в ширину. В этой главе мы обсудим стандартные для поднятой темы вопросы. Также я приведу эффективные реализации нескольких алгоритмов, в которых используются индексированные кучи. Кроме того, будут кратко представлены более продвинутые алгоритмы, основанные на сетевых потоках.

## 11.2. Основы

*Граф* состоит из  $V$  вершин, соединенных  $E$  ребрами. Между любыми двумя вершинами существует хотя бы одно ребро, и ни одна вершина не имеет ребер сама с собой. Последовательность ребер называется *путем*. Граф является:

- ♦ *разреженным*, если в нем менее  $V^2/2$  ребер, и *плотным* в противном случае. У самых полезных графов  $E = O(V)$ ;
- ♦ *неориентированным*, если его ребра двунаправлены, и *ориентированным* в противном случае;
- ♦ *сильносвязанным*, если является *ориентированным* и из любой вершины существует путь к любой другой вершине;
- ♦ *связанным*, если является *неориентированным*, но сильносвязанным. Несвязный граф состоит из нескольких связных компонентов;
- ♦ *ациклическим*, если является *ориентированным*, и из любой вершины нет пути в эту же вершину. Такой тип графов распространен и называется сокращенно DAG;
- ♦ *деревом*, если ациклический, и каждая его вершина имеет не более одного входящего ребра;
- ♦ *двудольным*, если вершины образуют две группы и между вершинами одной группы нет ребер;
- ♦ *неявным*, если не представлен структурой данных, а вершины, ребра и данные ребер определяет некоторая функция.

Варианты различных графов представлены на рис. 11.1 и 11.2.

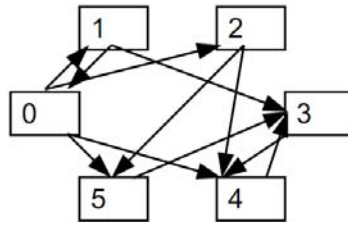


Рис. 11.1. Ориентированный циклический слабосвязанный разреженный граф

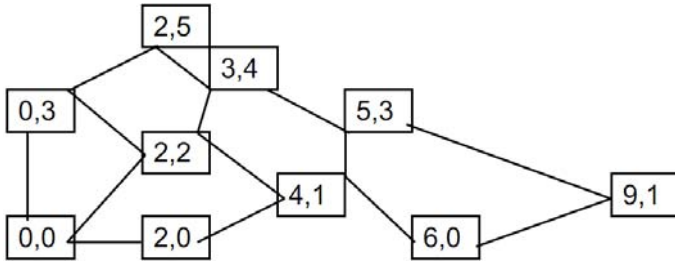


Рис. 11.2. Неориентированный разреженный связный граф, где данные в ребрах равны евклидову расстоянию между координатами вершин

## 11.3. Представление графов

Графы поддерживают:

- ◆ строительство;
- ◆ итерацию по вершинам;
- ◆ итерацию по ребрам вершины.

Некоторым алгоритмам требуются операции отображения с ребрами или вершинами, особенно если нужно найти исходящие ребра для некоторой вершины. Любое представление занимает пространство  $O(V + E)$ , если граф не является хотя бы частично неявным. Для хранения плотного графа требуется  $O(V^2)$  битов. Неориентированные графы представляются как ориентированные, но с двумя ребрами. У многих графов нет данных о вершинах, поэтому они более эффективно представлены вызывающим кодом, а данные ребер — графом.

Представьте динамически сортируемую последовательность вершин, у каждой из которых есть свой номер, и динамически отсортированную последовательность исходящих номеров вершин с соответствующими данными ребер. Такая структура данных эффективно поддерживает все операции, но вектор векторов (также называемый *массивом смежности*) влечет за собой меньшие накладные расходы, поддерживает пошаговое построение и итерацию, а также используется в большинстве алгоритмов (рис. 11.3 и 11.4).

Итерация по массиву смежности — это перебор всех исходящих ребер вершины:

```
template<typename EDGE_DATA> class GraphAA
{
    struct Edge
```

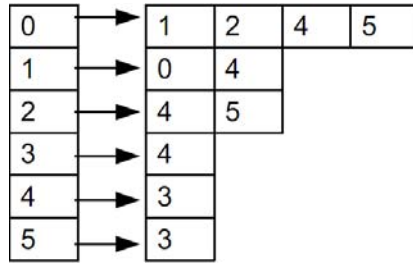


Рис. 11.3. Структура массива смежности для графа, показанного на рис. 11.1

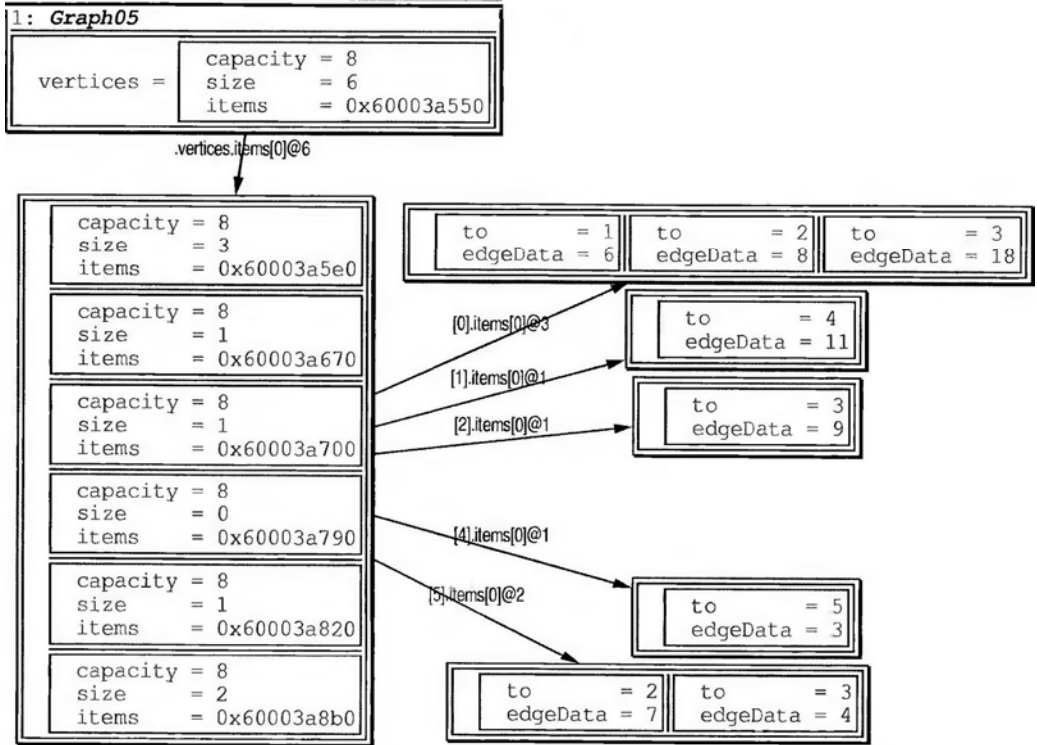


Рис. 11.4. Структура памяти массива смежности для графа, показанного на рис. 11.1, с краевыми данными

```

{
    int to;
    EDGE_DATA edgeData;
    Edge(int theTo, EDGE_DATA const& theEdgeData): to(theTo),
        edgeData(theEdgeData) {}
};
Vector<Vector<Edge> > vertices;
public:
    GraphAA(int initialSize = 0): vertices(initialSize) {}
    int nVertices()const{return vertices.getSize();}
    int nEdges(int v)const{return vertices[v].getSize();}

```

```

void addVertex(){vertices.append(Vector<Edge>());}
void addEdge(int from, int to, EDGE_DATA const& edgeData = EDGE_DATA())
{
    assert(to >= 0 && to < vertices.getSize());
    vertices[from].append(Edge(to, edgeData));
}
void addUndirectedEdge(int from, int to,
    EDGE_DATA const& edgeData = EDGE_DATA())
{
    addEdge(from, to, edgeData);
    addEdge(to, from, edgeData);
}
class AdjacencyIterator
{
    Vector<Edge> const* edges;
    int j; // текущее ребро
public:
    AdjacencyIterator(GraphAA const& g, int v, int theJ):
        edges(&g.vertices[v]), j(theJ){}
    AdjacencyIterator& operator++()
    {
        assert(j < edges->getSize());
        ++j;
        return *this;
    }
    int to(){return (*edges)[j].to;}
    EDGE_DATA const& data(){return (*edges)[j].edgeData;}
    bool operator!=(AdjacencyIterator const& rhs){return j != rhs.j;}
};
AdjacencyIterator begin(int v)const
{return AdjacencyIterator(*this, v, 0);}
AdjacencyIterator end(int v)const
{return AdjacencyIterator(*this, v, nEdges(v));}
};

```

В ориентированном графе легко перевернуть ребра:

```

template<typename GRAPH> GRAPH reverse(GRAPH const& g)
{
    GRAPH result(g.nVertices());
    for(int i = 0; i < g.nVertices(); ++i)
        for(typename GRAPH::AdjacencyIterator j = g.begin(i);
            j != g.end(i); ++j) result.addEdge(j.to(), i, j.data());
    return result;
}

```

Для статического графа более компактным представлением является объединение всех массивов ребер. В массиве хранятся ребра из вершины 0, затем из вершины 1 и т. д., а массив вершин индексируется в массив ребер. Байт-код или какой-либо другой механизм (см. главу 15. *Сжатие*) позволяет выполнить сжатие еще сильнее.

## 11.4. Поиск

Итерация вершин и ребер позволяет выполнять обход графа, но специальные порядки — например, генерируемые после выполнения *поиска в глубину* (depth-first search, DFS), обладают полезными свойствами. Алгоритм DFS, вызванный в исходной вершине, перебирает ее ребра, рекурсивно вызывая себя в каждой не посещенной целевой вершине (рис. 11.5). В результате обхода формируется дерево со следующими ребрами:

- ◆ *ребра дерева* — при посещении ранее не посещенной вершины;
- ◆ *обратные ребра* — при возврате из края дерева;
- ◆ *передние ребра* — при прыжке в ранее посещенный потомок текущей вершины;
- ◆ *поперечные ребра* — при прыжке в ранее посещенный не потомок текущей вершины.

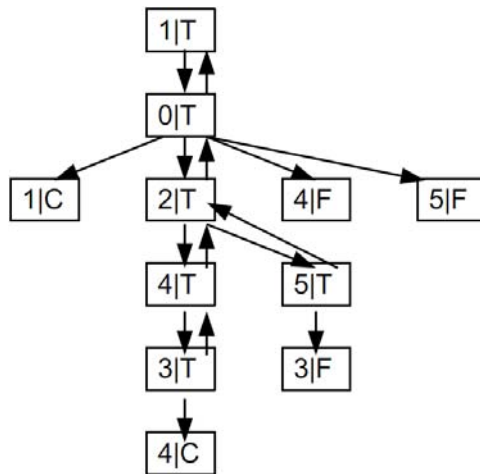


Рис. 11.5. Логика перехода DFS на примере графа, показанного на рис. 11.1: цифры обозначают индексы вершин, а буквы — типы ребер. Стрелки, направленные вверх, — это обратные ребра

Чтобы не исчерпать память при работе с большими графами, используйте стек. Вы должны вталкивать в него текущую вершину и следующий дочерний элемент, который нужно посетить, выталкивать элемент при переходе по обратному ребру, а также вызывать функтор действия при переходе в исходную вершину или по любому ребру. Функтор по умолчанию предоставляется для повторного использования кода. Передние и поперечные ребра выделены в отдельные группы, потому что отличить их от ребер дерева невозможно. Помощник, представленный в следующем коде, исследует подключенный компонент:

```

struct DefaultDFSAction
{
    void source(int v) {}
    void treeEdge(int v) {}
    void nonTreeEdge(int v) {}
    void backwardEdge(int v) {}
};

template<typename GRAPH, typename ACTION> void DFSComponent(GRAPH const& g,
    int source, Vector<bool>& visited, ACTION& a = ACTION())

```

```

{
    typedef typename GRAPH::AdjacencyIterator ITER;
    Stack<pair<ITER, int> > s;
    s.push(make_pair(g.begin(source), source));
    while(!s.isEmpty())
    {
        ITER& j = s.getTop().first;
        if(j != g.end(s.getTop().second))
        {
            int to = j.to();
            if(visited[to]) a.nonTreeEdge(to);
            else
            {
                a.treeEdge(to);
                visited[to] = true;
                s.push(make_pair(g.begin(to), to));
            }
            ++j;
        }
        else
        {
            s.pop();
            if(!s.isEmpty()) a.backwardEdge(s.getTop().second);
        }
    }
}

```

Поскольку граф может быть рассоединенным, мы вызываем DFS на каждой вершине. Это не влияет на правильность и асимптотическое время выполнения, поскольку мы помечаем посещенные вершины и не вызываем действий при входе в источник:

```

template<typename GRAPH, typename ACTION> void DFS(GRAPH const& g,
    ACTION& a = ACTION())
{
    Vector<bool> visited(g.nVertices(), false);
    for(int i = 0; i < g.nVertices(); ++i) if(!visited[i])
    {
        a.source(i);
        visited[i] = true;
        DFSComponent(g, i, visited, a);
    }
}

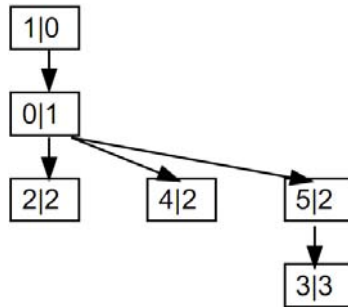
```

*Поиск в ширину* (breadth-first search, BFS) — это еще один не менее полезный способ изучения графов:

1. Ставим в очередь исходную вершину.
2. Пока очередь не опустеет:
3. Удаляем вершину из очереди.
4. Выполняем итерацию по ребрам, ставя в очередь каждое ребро.

Здесь действия не выполняются, потому что, по сути, единственным результатом работы алгоритма BFS является количество ребер, по которым нужно перейти для достиже-

ния каждой вершины из источника. Корректность алгоритма обеспечивается обходом вершин от ближних к дальним с постановкой впервые посещенных вершин в очередь (рис. 11.6).



**Рис. 11.6.** Логика перехода BFS на примере графа, показанного на рис. 11.1: первое число в узле — это вершина, а второе — расстояние от источника, выраженное количеством ребер

```

template<typename GRAPH> Vector<int> BFS(GRAPH& g, int source)
{
    Vector<int> distances(g.nVertices(), -1);
    Queue<int> q(g.nVertices());
    distances[source] = 0;
    q.push(source);
    while(!q.isEmpty())
    {
        int i = q.pop();
        for(typename GRAPH::AdjacencyIterator j = g.begin(i); j != g.end(i);
            ++j) if(distances[j.to()] == -1)
        {
            distances[j.to()] = distances[i] + 1;
            q.push(j.to());
        }
    }
    return distances;
}
  
```

Алгоритм BFS полезен только для изучения одного компонента. Алгоритм DFS проходит непрерывный путь, а BFS перескакивает через вершины. Оба выполняются за время  $O(V + E)$ , предполагая, что действия занимают время  $O(1)$ .

## 11.5. Примеры задач поиска

Алгоритм DFS находит *связанные компоненты* графа, если вызвать его с функтором, который добавляет новый компонент при входе в источник и добавляет любую вновь посещенную вершину к текущему компоненту:

```

struct ConnectedComponentAction: public DefaultDFSAction
{
    Vector<Vector<int> > components;
  
```



```

void source(int v)
{
    components.append(Vector<int>());
    treeEdge(v);
}

void treeEdge(int v) {components.lastItem().append(v);}
};

template<typename GRAPH>
Vector<Vector<int>> connectedComponents(GRAPH const& g)
{
    ConnectedComponentAction a;
    DFS(g, a);
    return a.components;
}

```

*Топологическая сортировка DAG* — это такой порядок вершин, что для любой вершины не существует пути к любой другой вершине, находящейся в отсортированном порядке раньше. Например, это может быть порядок, в котором нужно проходить учебные курсы. Когда DFS возвращается к вершине и входит в следующее исходящее ребро, он может посетить вершины на обратном пути последними в топологическом порядке. Таким образом, присвоение вершинам рангов, начиная с  $V - 1$ , при проходе по обратным ребрам дает топологическую сортировку. Если взять ребро, не являющееся ребром дерева, которое ведет вершине без ранга, это должно быть поперечное ребро, потому что, если бы это было прямое ребро, вершина имела бы ранг.

Когда у графа есть поперечное ребро, он имеет *цикл*, поэтому его нельзя топологически отсортировать. Например, на рис. 11.3 поперечные ребра (0,1) и (3,4) соответствуют циклам. Если их удалить, когда DFS возвращается в вершину 2 из 3 и переходит в 5, вершине 3 присвоится ранг 5, а вершине 4 — ранг 4. Затем, возвращаясь из вершины 5 в вершину 1, алгоритм выставит ранг 3 вершине 5, 2 — вершине 2, 1 — вершине 0 и 0 — вершине 1:

```

struct TopologicalSortAction
{
    int currentRank, leaf; // текущий лист дерева DFS
    Vector<int> ranks;
    bool hasCycle;
    TopologicalSortAction(int nVertices): currentRank(nVertices), leaf(-1),
        ranks(nVertices, -1), hasCycle(false) {}
    void source(int v) {treeEdge(v);}
    void treeEdge(int v) {leaf = v;} // потенциальный лист
    // если вершина не имеет ранга, мы нашли поперечное ребро
    void nonTreeEdge(int v) {if(ranks[v] == -1) hasCycle = true;}
    void backwardEdge(int v)
    {
        if(leaf != -1)
        { // присваиваем листу дерева DFS ранг
            ranks[leaf] = --currentRank;
            leaf = -1;
        }
        ranks[v] = --currentRank;
    }
};

```

```
template<typename GRAPH> Vector<int> topologicalSort (GRAPH const& g)
{
    TopologicalSortAction a(g.nVertices());
    DFS(g, a);
    if(a.hasCycle) a.ranks = Vector<int>(); // пустое значение ранга
                                           // говорит о наличии цикла
    return a.ranks;
}
```

DFS может генерировать случайный лабиринт:

1. Пусть есть с прямоугольная сетка ячеек единичного размера со стенками между любыми двумя соседними ячейками и ограничивающей рамкой вокруг всех ячеек.
2. Сетка представляется как граф, где вершины соответствуют ячейкам, а ребра соответствуют стенкам.
3. Случайным образом переставляем каждый массив ребер.
4. Начиная с любой вершины, запускаем алгоритм DFS, который стирает стену, образуя ребро дерева.
5. Выбираем любой начальный и конечный квадрат и удаляем его стены.

## 11.6. Минимальное остовное дерево

Когда в ребрах дерева содержится расстояние между вершинами, которые они соединяют, *минимальное остовное дерево* (minimum spanning Tree, MST) соединяет все вершины  $V - 1$  ребрами, такими что  $\sum$  расстояния между ребрами оказываются минимальными (рис. 11.7).

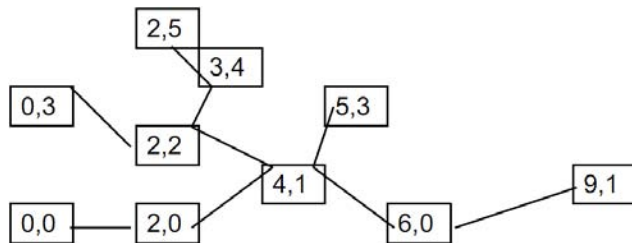


Рис. 11.7. MST графа евклидовых данных

Алгоритм *Прима* итеративно добавляет вершины, длина ребер до которых из любой вершины, уже находящейся в дереве, минимальна. Полученное дерево представляется в виде последовательности вершин:

1. Создаем очередь с индексированным приоритетом, упорядоченную по расстоянию от любой вершины в дереве.
2. Ставим в очередь все вершины с приоритетом  $\infty$ .
3. Пока очередь не пуста:
4. Удаляем вершину из очереди.
5. Уменьшаем известные расстояния до дочерних элементов.

6. Любая дочерняя вершина обновляет родительскую, если расстояние уменьшилось.

```
template<typename GRAPH> Vector<int> MST(GRAPH& g)
{ // представляем MST как ребра к родительским вершинам (у первого
  // узла их не будет)
  Vector<int> parents(g.nVertices(), -1);
  typedef pair<double, int> QNode;
  IndexedArrayHeap<QNode, PairFirstComparator<double, int> > pQ;
  for(int i = 0; i < g.nVertices(); ++i)
    pQ.insert(QNode(numeric_limits<double>::max(), i), i);
  while(!pQ.isEmpty())
  {
    int i = pQ.deleteMin().first.second;
    for(typename GRAPH::AdjacencyIterator j = g.begin(i); j != g.end(i);
      ++j)
    { // регулируем наиболее известные расстояния
      // до дочерних вершин, еще не входящих в дерево
      QNode const* child = pQ.find(j.to()); // потомок больше
      // не может находиться в очереди

      if(child && j.data() < child->first)
      {
        pQ.changeKey(QNode(j.data(), j.to()), j.to());
        parents[j.to()] = i; // обновляем ближайшего родителя
      }
    }
  }
  return parents;
}
```

Время выполнения равно  $O(E \ln(V))$  из-за  $O(E)$  операций смены ключа. Затраты памяти равны  $O(V)$ .

## 11.7. Кратчайшие пути

Когда в ребрах хранятся расстояния между вершинами, которые они соединяют, можно найти кратчайший путь от любой вершины к любой другой (рис. 11.8). Расстояние от любой вершины до нее самой считается равным нулю.

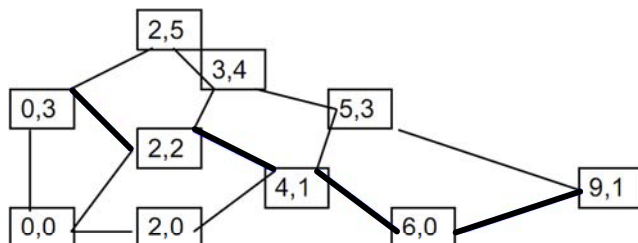


Рис. 11.8. Кратчайший путь из (0, 3) в (9, 1)

Когда все расстояния  $\geq 0$ , алгоритм Дейкстры находит расстояния от исходной вершины до всех остальных и останавливается, когда будет найдено расстояние до конечной вершины:

1. Создаем очередь с индексированным приоритетом, которая упорядочивает по расстоянию от источника.
2. Всем вершинам задаем расстояние  $\infty$ , а источнику — расстояние 0.
3. Пока очередь не опустела или не достигнуто целевое состояние, если таковое существует:
4. Удаляем из очереди следующую вершину.
5. Для любой дочерней вершины:
6. Расстояние = расстояние до вершины + расстояние (вершина, пункт назначения), если оно меньше.

В результате для любой вершины можно получить следующую вершину на обратном пути к источнику:

```
template<typename GRAPH>
Vector<int> ShortestPath(GRAPH& g, int from, int dest = -1)
{ // по умолчанию целевого состояния нет
    assert(from >= 0 && from < g.nVertices());
    Vector<int> pred(g.nVertices(), -1);
    typedef pair<double, int> QNode;
    IndexedArrayHeap<QNode, PairFirstComparator<double, int> > pQ;
    for(int i = 0; i < g.nVertices(); ++i) pQ.insert(
        QNode(i == from ? 0 : numeric_limits<double>::infinity(), i));
    while(!pQ.isEmpty() && pQ.getMin().first.second != dest)
    {
        int i = pQ.getMin().first.second;
        double dj = pQ.deleteMin().first.first; // расстояние до текущего узла
        for(typename GRAPH::AdjacencyIterator j = g.begin(i); j != g.end(i);
            ++j)
        { // потомок может больше не быть в очереди
            double newChildDistance = dj + j.data();
            QNode const* child = pQ.find(j.to());
            if(child && newChildDistance < child->first)
            {
                pQ.changeKey(QNode(newChildDistance, j.to()), j.to());
                pred[j.to()] = i; // новый лучший родитель
            }
        }
    }
    return pred;
}
```

Время выполнения равно  $O(E \ln(V))$  из-за  $O(E)$  операций смены ключа. Затраты памяти равны  $O(V)$ .

Когда некоторые расстояния *меньше нуля*, в графе могут получиться отрицательные циклы с расстоянием меньше 0, а их обход делает общее расстояние сколь угодно малым. Это происходит, например, если мы моделируем валютный рынок, а обменный

курс задается ребрами. Если в графе существуют отрицательные циклы, кратчайший путь в нем найти нельзя.

Когда некоторые расстояния меньше нуля, алгоритм Дейкстры работает неверно и не может обнаружить отрицательные циклы. *Алгоритм Беллмана — Форда* предполагает, что наилучшее расстояние до вершины становится известно после ее обработки:

1. Задаем расстояния до всех вершин равными  $\infty$ .
2. Задаем расстояние до источника равным 0.
3. Пока очередь не опустеет или не будет обнаружен отрицательный цикл:
  4. Удаляем вершину из очереди.
  5. Для любой дочерней вершины:
    6. Ставим в очередь, если она еще там не находится.
7. Расстояние = расстояние до вершины + расстояние (вершина, пункт назначения), если оно меньше.

В результате для любой вершины мы определяем следующую вершину на обратном пути к источнику:

```
template<typename GRAPH> struct BellmanFord
{
    int v; // вершина должна быть первой
    Vector<double> distances;
    Vector<int> pred;
    bool hasNegativeCycle;
    BellmanFord(GRAPH& g, int from): v(g.nVertices()), pred(v, -1),
        distances(v, numeric_limits<double>::infinity()),
        hasNegativeCycle(false)
    {
        assert(from >= 0 && from < v);
        Queue<int> queue;
        Vector<bool> onQ(v, false);
        distances[from] = 0;
        queue.push(from);
        onQ[from] = true;
        for(int nIterations = 0; !queue.isEmpty() && !hasNegativeCycle;)
        {
            int i = queue.pop();
            onQ[i] = false;
            for(typename GRAPH::AdjacencyIterator j = g.begin(i);
                j != g.end(i); ++j)
            {
                double newChildDistance = distances[i] + j.data();
                if(newChildDistance < distances[j.to()])
                {
                    distances[j.to()] = newChildDistance;
                    pred[j.to()] = i; // новый лучший родитель
                    if(!onQ[j.to()])
                    {
                        queue.push(j.to());
                    }
                }
            }
        }
    }
};
```

```

        onQ[j.to()] = true;
    }
} // через каждые v внутренних итераций проверяем
// наличие отрицательного цикла
if(++nIterations % v == 0)
    hasNegativeCycle = findNegativeCycle();
}
}
};

```

Проверка отрицательного цикла гарантирует, что ни одна вершина не является своим собственным предком, и выполняется через каждые  $V$  обновлений расстояния, чтобы амортизировать ее стоимость. Путем поиска по объединению мы соединяем каждую вершину с ее родителем. Цикла в графе не будет, если ни одна вершина не находится в том же подмножестве, что и ее родитель, до соединения. В противном случае объединение создает этот самый цикл:

```

bool findNegativeCycle()
{
    UnionFind uf(v);
    for(int i = 0; i < v; ++i)
    {
        int parent = pred[i];
        if(parent != -1)
        { // не может находиться в одном подмножестве с родителем
            if(uf.areEquivalent(i, parent)) return true;
            uf.join(i, parent);
        }
    }
    return false;
}

```

Время выполнения равно  $O(VE)$ , потому что каждое ребро ставится в очередь  $\leq V$  раз, если отрицательных циклов нет. В противном случае он за это время обнаруживается (см. [11.1]). На практике алгоритм работает намного быстрее из-за меньшего количества очередей.

## 11.8. Алгоритмы потока

Поток — это полезная обобщенная модель для задач вроде распределения и доставки товаров. Имея связный неориентированный граф, где вершины обозначают *источники* и *приемники*, нужно назначить потоки ребрам таким образом, чтобы:

- ◆ за исключением источника и конечного приемника,  $\sum$  входящих ребер =  $\sum$  выходящих;
- ◆ для любой вершины выходной поток  $> 0$ .

В задаче о *максимальном потоке* каждому ребру назначается пропускная способность, которая больше или равна значению потока, и вам требуется значение потока, максимизирующее суммарный поток из источника (рис. 11.9).

Пусть граничные мощности равны 2 для вершин от (5,3) до (9,1) и 1 для остальных

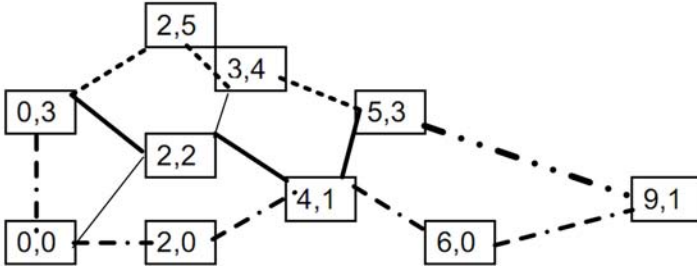


Рис. 11.9. Потоки проходят по путям, показанным линиями —, --- и - · - ·. Пути — и --- сливаются в - · - ·

В задаче о потоке с минимальной стоимостью каждому ребру назначается мощность  $\geq 0$  и стоимость, и от вас требуется найти поток со значением, превышающим необходимое количество или равным ему, такой, что суммарная стоимость потока окажется минимальна (рис. 11.10). Если необходимая сумма слишком велика, у вас нет подходящего решения.

Пусть граничные мощности равны 1, а стоимости заданы графом

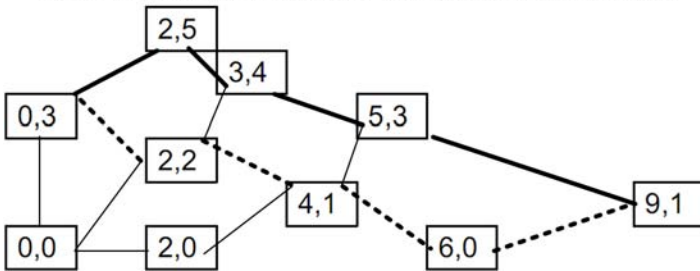


Рис. 11.10. Полученные потоки величиной 2 проходят по путям, показанным линиями — и ---

В данных ребра указано направление потока, позволяющее определить оставшуюся мощность. Для прямого потока можно передавать значение *мощность-поток*, а для обратного — *поток*. Каждое ребро связано со своей исходной вершиной. Вы можете увеличивать или уменьшать значение потока вдоль ребра края:

```
struct FlowData
{
    int from;
    double flow, capacity, cost; // стоимость, используемая
                                // для минимального потока
    FlowData(int theFrom, double theCapacity, double theCost = 0):
        from(theFrom), capacity(theCapacity), flow(0), cost(theCost) {}
    double capacityTo(int v) const { return v == from ? flow : capacity - flow; }
    // значение потока может выйти за диапазон (0, мощность),
    // но это не страшно
    void addFlowTo(int v, double change)
        { flow += change * (v == from ? -1 : 1); }
};
```

*Алгоритм увеличивающегося пути* для максимального потока:

1. Пока можно найти путь от источника к приемнику, по которому есть возможность отправить поток больше нуля:
2. Отправляем по пути максимальную возможную сумму

Это применимо для потока как с максимальной, так и с минимальной стоимостью, в зависимости от того, как алгоритм ищет пути. От вызывающего абонента ожидается, что `needFlow` равняется нулю для максимального потока, либо требуемой сумме, если стоимость может быть отрицательной. Для экономии памяти, которая тратится на хранение неориентированных ребер, данные потока вынесены в отдельный массив, где данные ребер графа являются индексами. Массив также будет содержать окончательные назначения потока. В структурах данных `pred` и `path` хранятся последовательность вершин и индексы соответствующих ребер потока:

```
template<typename GRAPH> class ShortestAugmentingPath
{
    int v;
    Vector<int> path, pred;
    double totalFlow;
public:
    double getTotalFlow()const{return totalFlow;}
    ShortestAugmentingPath(GRAPH const& g, Vector<FlowData>& fedges, int from,
        int to, double neededFlow = 0): v(g.nVertices()), totalFlow(0),
        path(v, -1), pred(v, -1)
    { // итеративно ищем путь
        assert(from >= 0 && from < v && to >= 0 && to < v);
        while(neededFlow == 0 ? hasAugmentingPath(g, fedges, from, to) :
            hasMinCostAugmentingPath(g, fedges, from, to, neededFlow))
        { // из него находится значение потока, которое нужно прибавить
            double increment = numeric_limits<double>::max();
            for(int j = to; j != from; j = pred[j])
                increment = min(increment, fedges[path[j]].capacityTo(j));
            if(neededFlow != 0) // этот случай касается только
                // самого малозатратного потока
                increment = min(increment, abs(neededFlow) - totalFlow);
            // прибавление ко всем ребрам
            for(int j = to; j != from; j = pred[j])
                fedges[path[j]].addFlowTo(j, increment);
            totalFlow += increment;
        }
    }
};
```

В результате поиск пути для максимального потока использует расширенный BFS, чтобы игнорировать края, заполненные до отказа или уже являющиеся частью существующего пути:

```
bool hasAugmentingPath(GRAPH const& g, Vector<FlowData>& fedges, int from,
    int to)
{
    for(int i = 0; i < v; ++i) pred[i] = -1;
    Queue<int> queue;
```



```

queue.push(pred[from] = from);
while(!queue.isEmpty())
{
    int i = queue.pop();
    for(typename GRAPH::AdjacencyIterator j = g.begin(i);
        j != g.end(i); ++j)
        if(pred[j.to()] == -1 && // не посещенная вершина
            // с заданной мощностью
            fedges[j.data()].capacityTo(j.to()) > 0)
        {
            pred[j.to()] = i;
            path[j.to()] = j.data();
            queue.push(j.to());
        }
}
return pred[to] != -1;
}

```

Время выполнения равно  $O(VE^2)$ , потому алгоритм может быть расширен сложностью  $O(VE)$  с BFS (см. [11.1]). На практике алгоритм намного быстрее.

Для потока с минимальной стоимостью алгоритм находит кратчайший путь относительно стоимости и игнорирует полные ребра. В этой реализации строится еще один граф для повторного использования алгоритмов кратчайшего пути. Затем мы находим соответствующие ребра и сохраняем индексы данных потока:

```

bool hasMinCostAugmentingPath(GRAPH const& g, Vector<FlowData>& fedges,
    int from, int to, double neededFlow)
{ // если нужно большее значение потока, строим граф
  // из ребер с имеющейся мощностью
  if(totalFlow >= abs(neededFlow)) return false;
  GraphAA<double> costGraph(v);
  for(int i = 0; i < v; ++i)
      for(typename GRAPH::AdjacencyIterator j = g.begin(i);
          j != g.end(i); ++j)
          if(fedges[j.data()].capacityTo(j.to()) > 0)
              costGraph.addEdge(i, j.to(), fedges[j.data()].cost);
  if(neededFlow > 0) pred = ShortestPath(costGraph, from, to);
  else
  { // отрицательные значения стоимости
      BellmanFord<GraphAA<double>> bf(costGraph, from);
      if(bf.hasNegativeCycle())
      {
          totalFlow = numeric_limits<double>::infinity();
          return false;
      }
      pred = bf.pred;
  };
  // извлекаем ребра пути
  for(int i = to; pred[i] != -1; i = pred[i])
      for(typename GRAPH::AdjacencyIterator j = g.begin(pred[i]);
          j != g.end(pred[i]); ++j)

```

```

    if(j.to() == i)
    {
        path[i] = j.data();
        break;
    }
    return pred[to] != -1;
}

```

Алгоритм быстр на практике, но находит кратчайшие пути за время  $O(VU)$ , где  $U \geq$  выходному потоку любой вершины (см. [11.1]).

## 11.9. Двудольное сопоставление

Даны две группы вершин и множество разрешенных ребер. Требуется найти подмножество ребер такое, чтобы у как можно большего числа вершин было только одно ребро. Сопоставление *идеально*, если у каждой вершины будет свое ребро (рис. 11.11).

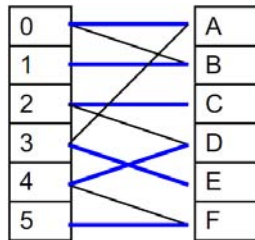


Рис. 11.11. Ребра идеального сопоставления

Самое простое решение сводится к поиску максимального потока. Присоедините источник ко всем вершинам в первой группе, сток ко всем вершинам во второй группе и присвойте всем ребрам пропускную способность 1. Время выполнения равно  $O(VE)$  (см. [11.1]):

```

Vector<pair<int, int> > bipartiteMatching(int n, int m,
    Vector<pair<int, int> > const& allowedMatches)
{ // v = n + m + 2, e = n + m + allowedMatches.getSize(),
  // время равно O(VE)
  GraphAA<int> sp(n + m + 2); // настраиваем граф и ребра потока
  Vector<FlowData> data;
  for(int i = 0; i < allowedMatches.getSize(); ++i)
  {
      data.append(FlowData(allowedMatches[i].first, 1));
      sp.addUndirectedEdge(allowedMatches[i].first,
          allowedMatches[i].second, i);
  }
  int source = n + m, sink = source + 1; // настраиваем группы
                                          // источников и стоков

  for(int i = 0; i < source; ++i)
  {
      int from = i, to = sink;

```

```
    if(i < n)
    {
        from = source;
        to = i;
    }
    data.append(FlowData(from, 1));
    sp.addUndirectedEdge(from, to, i + allowedMatches.getSize());
} // вычисляем соответствие
ShortestAugmentingPath<GraphAA<int> > dk(sp, data, source, sink);
// возвращаем ребра с положительным потоком
Vector<pair<int, int> > result;
for(int i = 0; i < allowedMatches.getSize(); ++i)
    if(data[i].flow > 0) result.append(allowedMatches[i]);
return result;
}
```

## 11.10. Устойчивое сопоставление

Когда у каждого из  $m$  мужчин и  $n$  женщин есть предпочтения, их сопоставление *устойчиво*, если никакие две пары не могут распасться из-за того, что мужчина в одной паре и женщина в другой предпочитают друг друга своим имеющимся партнерам. Если  $n \leq m$ , мужчины выбирают женщин, иначе женщины выбирают мужчин. Для примера предположим, что  $n \leq m$ . Для удобства предпочтения мужчин представим в виде упорядоченного списка женщин, а предпочтения женщин — в виде рангов мужчин (рис. 11.12).



Рис. 11.12. Ранги предпочтения и полученные сопоставления (мужчины начинаются с Z, а женщины — с A)

Алгоритм Гейла — Шепли итеративно вычисляет устойчивое сопоставление:

1. Изначально предпочтения не назначены.
2. Пока мужчинам не назначены предпочтения:
3. Любой незанятый мужчина делает предложение лучшей женщине, которой он еще не сделал предложение.
4. Она принимает его, если его ранг выше ранга ее нынешнего партнера, и отвергает в противном случае.

```

Vector<int> stableMatching(Vector<Vector<int> > const& womenOrders,
    Vector<Vector<int> > const& menScores)
{
    int n = womenOrders.getSize(), m = menScores.getSize();
    assert(n <= m);
    Stack<int> unassignedMen; // Любой тип списка подойдет
    for(int i = 0; i < n; ++i) unassignedMen.push(i);
    Vector<int> currentMan(m, -1), nextWoman(n, 0);
    while(!unassignedMen.isEmpty())
    {
        int man = unassignedMen.pop(), woman, currentM;
        do
        { // выхода за пределы не будет, т. к. n <= m
            woman = nextWoman[man]++;
            currentM = currentMan[woman];
        } while(currentM != -1 && // сопоставление найдено
            menScores[woman][man] <= menScores[woman][currentM]);
        currentMan[woman] = man; // поиск сопоставления
        // либо развод, либо поиск продолжается
        if(currentM != -1) unassignedMen.push(currentM);
    }
    return currentMan;
}

```

Если предпочтения всех женщин одинаковы, алгоритм работает быстрее, когда мужчины отсортированы в порядке убывания ранга, и в этом случае ни одна женщина не выбирает другого партнера. Поскольку каждая женщина и каждый мужчина встречаются в паре не более одного раза, время выполнения составляет  $O(nm)$ .

## 11.11. Задача назначения

Задача назначения аналогична двудольному сопоставлению, но у ребер есть веса, а сопоставление минимизирует суммарную стоимость ребер (рис. 11.13).

Простое решение сводится к поиску минимального потока и аналогично задаче двудольного сопоставления, за исключением использования весов, отличных от 1.

Пусть стоимость прямых ребер равна 1,  
а диагональных – 0

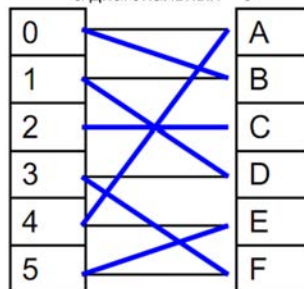


Рис. 11.13. Изменение в сопоставлениях в сравнении с рис. 11.2, если диагонали свободны

```

Vector<pair<int, int> > assignmentProblem(int n, int m,
    Vector<pair<pair<int, int>, double> > const& allowedMatches)
{ // v= n + m + 2, e = n + m + allowedMatches.getSize(), время равно O(VE)
  GraphAA<int> sp(n + m + 2); // настраиваем граф и ребра потока
  Vector<FlowData> data;
  for(int i = 0; i < allowedMatches.getSize(); ++i)
  {
    data.append(FlowData(allowedMatches[i].first.first, 1,
        allowedMatches[i].second));
    sp.addUndirectedEdge(allowedMatches[i].first.first,
        allowedMatches[i].first.second, i);
  }
  int source = n + m, sink = source + 1; // настраиваем группы источников и стоков
  for(int i = 0; i < source; ++i)
  {
    int from = i, to = sink;
    if(i < n)
    {
      from = source;
      to = i;
    }
    data.append(FlowData(from, 1, 0));
    sp.addUndirectedEdge(from, to, i + allowedMatches.getSize());
  } // вычисляем соответствие
  ShortestAugmentingPath<GraphAA<int> > dummy(sp, data, source, sink,
      min(n, m));
  // возвращаем ребра с положительным потоком
  Vector<pair<int, int> > result;
  for(int i = 0; i < allowedMatches.getSize(); ++i)
    if(data[i].flow > 0) result.append(allowedMatches[i].first);
  return result;
}

```

При использовании кратчайшего метода расширения пути время выполнения по методу Беллмана — Форда составляет  $O(V^2E)$  (см. [11.1]).

## 11.12. Генерация случайных графов

Для правильного тестирования графовых алгоритмов нужны большие графы. Сгенерированный граф должен быть разреженным и удовлетворять другим требованиям алгоритма.

Проще всего создавать ребро между двумя вершинами с вероятностью  $p$ . Но  $E[\text{пространственное использование результата}] = O(pV^2)$ . Более полезная модель — получить ориентированный граф с  $k$  исходящими ребрами на вершину:

```

template<typename GRAPH>
GRAPH randomDirectedGraph(int vertices, int edgesPerVertex)
{
  assert(edgesPerVertex <= vertices);
  GRAPH g(vertices);

```

```
for(int i = 0; i < vertices; ++i)
{
    Vector<int> edges = GlobalRNG().sortedSample(edgesPerVertex, vertices);
    for(int j = 0; j < edgesPerVertex; ++j) g.addEdge(i, edges[j]);
}
return g;
}
```

Еще одна полезная модель генерирует  $n$  точек в единичном квадрате и создает граф, который соединяет каждую вершину с  $k$  ближайшими соседями или со всеми точками на некотором расстоянии. Однако такие модели менее эффективны. В работе [11.2] предлагаются эффективные генераторы для различных моделей.

## 11.13. Примечания по реализации

Единственное приведенное здесь нововведение — это использование индексированной кучи в вычислениях кратчайшего пути и MST. Все остальные алгоритмы стандартны.

Алгоритмы сетевого потока найти труднее всего, потому что мало где о них пишут. Мои простые реализации, особенно для задачи о потоке с минимальной стоимостью, больше нигде не встречаются, но в них используются не самые лучшие алгоритмы (см. разд. «Комментарии»).

## 11.14. Комментарии

В задаче нахождения MST алгоритм Крускала сортирует ребра по расстоянию и итеративно прибавляет к пути следующее кратчайшее ребро, которое соединяет два еще не соединенных компонента, найденных с помощью объединения-поиска. Алгоритму требуется память  $O(E)$ , и он не работает напрямую с представлением массива смежности.

Алгоритм Дейкстры также позволяет найти кратчайший путь с кратчайшим длиннейшим ребром, используя функцию  $\max(\text{расстояние до вершины}, \text{расстояние(вершина, дочерний элемент)})$  вместо суммы расстояния до вершины и расстояния(вершина, дочерний элемент), но это редко бывает полезно.

В задачах поиска кратчайших путей в больших графах огромный выигрыш по времени дает предварительная обработка. Почитайте в Google про *contraction hierarchies*, если любопытно. Другая модель заключается в поиске нескольких кратчайших путей хорошего качества, чтобы пользователи могли выбрать лучший в соответствии со своими потребностями. Поиск нескольких различных кратчайших путей бесполезен, поскольку они, как правило, очень похожи. В более совершенных моделях задаются альтернативные значения для ребер или определяются несколько функций расстояния. Это может быть, например, время в пути, затраты на топливо, количество пересадок и т. п. Каждая из этих задач решается отдельно.

В самых быстрых из известных алгоритмов потока используются более сложные методы *push-relabel* (см. [11.1, 11.8]). В работах [11.3] и [11.11] приведен их краткий обзор и ссылки на недавние исследования. Есть также современные экспериментальные работы с потоками с минимальными затратами (см. [11.5] и [11.7]). Большинство других алгоритмов графов и их практических реализаций были изобретены уже к 1990 году. Мы не

коснулись темы сопоставлений. Эта и другие, менее популярные связанные с алгоритмами темы, такие как проверка планарности, а также касающаяся их обширная теория, обсуждаются в работах [11.6] и [11.11].

## 11.15. Советы по дополнительной подготовке

Изучите и реализуйте основные алгоритмы push-relabel для потоков с максимальной и минимальной стоимостью. Работают ли они на больших тестовых задачах быстрее, чем алгоритмы расширения пути?

## 11.16. Список рекомендуемой литературы

- 11.1. Ahuja R. K., Magnanti T. L., & Orlin J. B. (1993). Network Flows: Theory, Algorithms, and Applications. MIT Press.
- 11.2. Batagelj V., & Brandes U. (2005). Efficient generation of large random networks. *Physical Review E*, 71(3), 036113.
- 11.3. Goldberg A. V., & Tarjan R. E. (2014). Efficient maximum flow algorithms. *Communications of the ACM*, 57(8), 82–89.
- 11.4. Heineman G. T., Pollice G., & Selkow S. (2016). Algorithms in a Nutshell. O'Reilly.
- 11.5. Kiraly Z., & Kovacs P. (2012). Efficient implementations of minimum-cost flow algorithms. *Acta Univ. Sapientiae*, 4(1), 67–118.
- 11.6. Kocay W., & Kreher D. L. (2016). Graphs, Algorithms, and Optimization. CRC.
- 11.7. Kovács P. (2015). Minimum-cost flow algorithms: an experimental evaluation. *Optimization Methods and Software*, 30(1), 94–127.
- 11.8. Mehlhorn K., & Näher S. (1999). LEDA: a Platform for Combinatorial and Geometric Computing. Cambridge University Press.
- 11.9. Mehlhorn K., & Sanders P., Dietzfelbinger M., & Dementiev R. (2019). Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox. Springer.
- 11.10. Sedgewick R., & Wayne K. (2011). Algorithms. Addison-Wesley.
- 11.11. Thulasiraman K., Arumugam S., Nishizeki T., & Brandstädt A. (2015). Handbook of Graph Theory, Combinatorial Optimization, and Algorithms. CRC.

## 12. Разные алгоритмы и методы

### 12.1. Введение

Это первая глава, в которой вы познакомитесь с материалом, выходящим за рамки стандартной учебной программы бакалавриата. Мы обсудим здесь несколько недостаточно обширных для создания отдельных глав тем. В основном это будут общие методы структурирования данных, алгоритмы кеширования и комбинаторная генерация.

### 12.2. Превращение статических структур данных в динамические

Структура данных является *динамической*, если она поддерживает обновления, *полудинамической*, если она не поддерживает удаление, и *статической*, если она не поддерживает обновление вообще. Когда параметры, определенные при построении структуры, не позволяют задать некоторую высоту, применяется *перестроение*, если для него имеется достаточно информации. В частности, это относится к вектору, но не к фильтру Блума.

*Частичное перестроение* восстанавливает часть структуры данных после последовательности обновлений или события. Например, чтобы сбалансировать дерево без вращения, выполняется рандомизированное перестроение, после которого дерево остается случайным, и каждый узел в каждом поддереве равновероятно может стать корнем дерева. Количество узлов сохраняется, а вставка в поддерево с  $n$  узлами перестраивает его с вероятностью  $1/n$ , используя инкрементное построение, обрабатывающее новый узел и узлы в поддереве, которое ему назначается. С вероятностью  $(n-1)/n$  вставка выполнится в несбалансированное дерево. Поскольку для случайного дерева  $\Pr(\text{высота} > c \lg(n))$  для  $c > 2$  экспоненциально мала, у любого узла есть в  $1/b$  раз больше потомков по сравнению с его соседом, где  $0,5 < b < 1$ . Если затраты на перестроение

равны  $n \lg(n)$ ,  $E[\text{стоимость вставки}] = C(n) \leq \frac{1}{n} n \lg(n) + \frac{n-1}{n} C(nb) \approx \lg(n) + C(nb)$ . Со-

гласно основной теореме  $C(n) = O(\lg(n)^2)$ . Это верно, если число  $b$  отдалено от 0 на  $O(1)$ , и ожидаемое поведение лучше, чем в худшем случае.

Еще один способ удовлетворить неравенству  $0,5 < b < 1$  — это создать идеально сбалансированное поддерево из самого высокого поддерева, ставшего несбалансированным в результате вставки. Амортизированная стоимость вставки составляет  $O(\lg(n)^2)$ , если перестроение занимает время  $O(n \lg(n))$ , хотя в среднем оно меньше. Баланс веса гарантирует высоту  $O(\lg(n))$  (см. [12.2]).

Операция *полного перестроения* перестраивает всю структуру данных. При удвоении массива выполняется перестроение вектора. Операция обновления называется *слабой*,



если после  $O(n)$  обновлений использование ресурсов всеми операциями увеличивается на  $O(1)$ . В этом случае перестроение после каждых  $O(n)$  обновлений снижает амортизированную стоимость перестроения в  $O(n)$  раз.

Возможность перестроения и слабого удаления позволяет сделать любую полудинамическую структуру данных динамической. При удалении элемента его нужно пометить логическим *надгробием*. После удаления достаточного количества нужно выполнить перестроение, чтобы окончательно очиститься от них. Стоимость слабого удаления амортизируется как  $O(\text{перестроение}/\text{количество удаленных элементов})$ . Перестроение для операций удаления выполняется реже, чем для вставок, чтобы последовательность вставок и удалений не вызывала частых перестроений.

Задача является *разложимой по порядку*, если существует способ получить результат запроса к множеству данных путем объединения результатов запросов к любому из разделов множества. Разложимость по порядку позволяет представить структуру данных набором блоков. В частности, для структуры данных с  $n$  элементами блоки имеют размер  $2^i$ , где выполняется неравенство  $0 \leq i \leq \lg(n)$ ,  $\sum$  размеров  $\geq n$ , и каждое значение  $i$  уникально. В итоге получается  $\leq \lg(n)$  блоков. Операция вставки создает структуру размера 1 и, если она уже есть, перестраивает обе структуры в структуру размера 2, и т. д. Механизм похож на работу двоичного счетчика и позволяет амортизировать вставку  $O(\lg(n))$  при времени восстановления  $O(n)$ . Операция удаления является слабой.

Если блоки поддерживают вставку до заполнения, вставка в самый большой блок до его заполнения и выделение еще одного блока вдвое большего размера работают более эффективно.

## 12.3. Обеспечение устойчивости структур данных

После каждого обновления структура данных меняется, и иногда возникает задача сохранения прошлых версий. Например, после фиксации кода в системе контроля версий создается новая версия репозитория. Структура данных является *частично постоянной*, когда каждая ее версия доступна, а обновления затрагивают только последнюю версию. Клонирование структуры данных при каждом обновлении позволяет реализовать этот механизм.

Есть и более эффективный *метод толстых узлов*. Если структура данных представляет собой набор связанных узлов, каждый указатель узла можно заменить вектором пар (номер версии, указатель) и сохранить эту версию структуры данных:

- ◆ при выполнении обновления нужно скопировать и обновить каждый затронутый узел, добавить пару (новый номер версии, новый указатель) к нужным векторам и увеличить номер версии. Время обновления при этом увеличивается на время копирования затронутых узлов;
- ◆ для работы с последней версией можно использовать последнюю пару в каждом векторе;
- ◆ чтобы вернуться на  $k$  версий назад, следует воспользоваться бинарным поиском за дополнительное время  $O(\lg(k))$ .

В качестве сохраняемого узла может выступать вся структура данных, отдельные ее биты или что-то промежуточное — например, ячейки памяти, не связанные со специ-

фикой структуры данных. Выбирайте узлы так, чтобы обновление меняло лишь часть из них. Например, для сохранения версионности дерева можно сделать каждый узел постоянным и выполнять копирования, если изменяются указатели элементов или дочерних элементов (рис. 12.1).

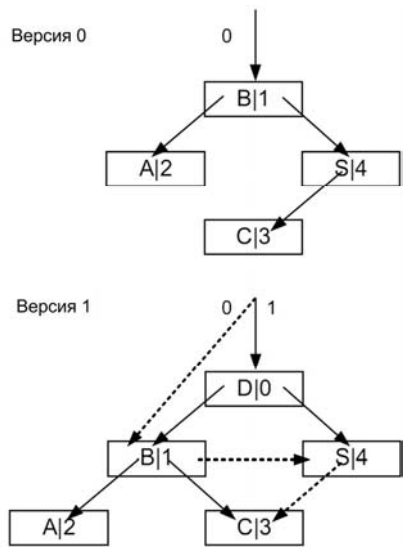


Рис. 12.1. Постоянное декартово дерево (treap) после вставки D с приоритетом 0

## 12.4. Хранение кеша

Поведение кеша определяется политиками размера и замены данных. При фиксированном размере идеальная политика определяет такое кеширование ресурсов, которое помогло бы для оптимизации доступа к ним в будущем. У политики *последнее недавно использованное* (Last Recently Used, LRU) коэффициент конкурентоспособности равен 2 (см. [12.4]). Когда кеш заполнен, и программа осуществляет доступ к ресурсу, который в кеше отсутствует, он вытесняет из кеша самый старый ресурс. Оптимальной реализацией является использование связанного списка, проиндексированного хеш-таблицей (рис. 12.2).

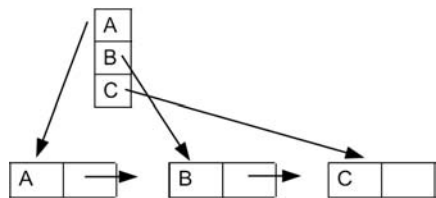


Рис. 12.2. Структура кеша LRU, реализованная в виде хеш-таблицы и связанного списка

Элементы в списке упорядочиваются по времени доступа, при этом недавно используемые элементы перемещаются на передний план, а хеш-таблица обеспечивает эффективный поиск, поэтому операции выполняются время  $O(\text{хеш-таблица})$ . Структура

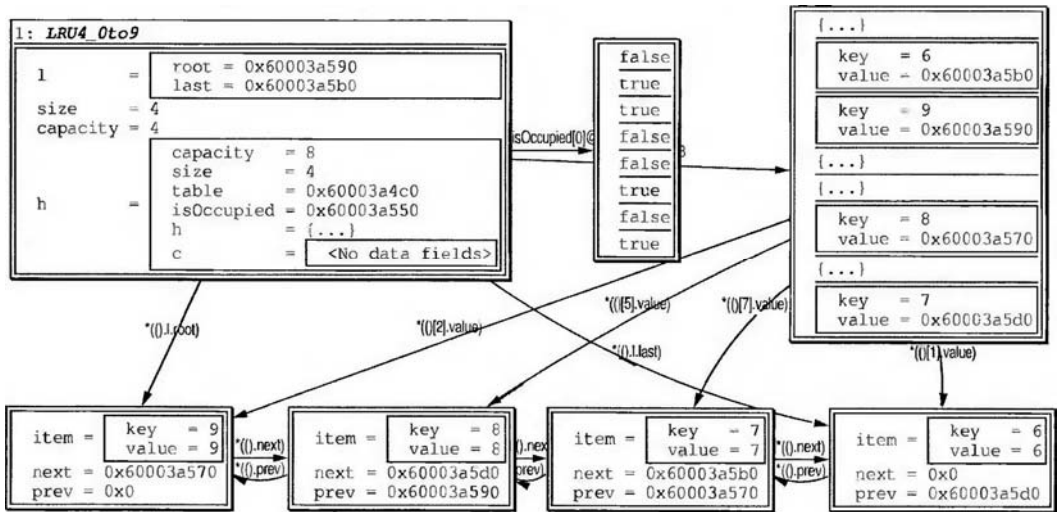


Рис. 12.3. Структура памяти кеша LRU со вставленными целочисленными элементами от 0 до 9 и удаленными элементами от 0 до 5

памяти кеша LRU со вставленными целочисленными элементами от 0 до 9 и удаленными элементами от 0 до 5 представлена на рис. 12.3.

```
template<typename KEY, typename VALUE,
        typename HASHER = EHash<BUHash> > class LRUCache
{
    typedef KVPair<KEY, VALUE> ITEM;
    typedef SimpleDoublyLinkedList<ITEM> LIST;
    typedef typename LIST::Iterator I;
    LIST l;
    int size, capacity;
    LinearProbingHashTable<KEY, I, HASHER> h;
public:
    LRUCache(int theCapacity): size(0), capacity(theCapacity)
    {assert(capacity > 0);}
    VALUE* read(KEY const& k)
    {
        I* np = h.find(k);
        if(np)
        { // помещается на передний план
            l.moveBefore(*np, l.begin());
            return &(*np)->value;
        }
        return 0;
    }
    typedef I Iterator;
    Iterator begin(){return l.begin();}
    Iterator end(){return l.end();}
    Iterator evicteeOnWrite(KEY const& k) // none, если кеш не заполнен
                                         // или элемент уже кеширован
    {return size < capacity || h.find(k) ? end() : l.rBegin();}
```

```

void write(KEY const& k, VALUE const& v)
{
    VALUE* oldV = read(k); // проверка, не был ли элемент уже вставлен
    if(oldV) *oldV = v;    // поиск, обновление
    else
    {
        Iterator evictee = evicteeOnWrite(k);
        if(evictee != end())
        {
            h.remove(evictee->key);
            l.moveBefore(evictee, l.begin()); // удаление evictee
            evictee->key = k;
            evictee->value = v;
        }
        else
        {
            ++size;
            l.prepend(ITEM(k, v));
        }
        h.insert(k, l.begin());
    }
}
};

```

Этот код можно применить для реализации фасада над различными ресурсами, чтобы пользователи API не знали о том, что кеш вообще задействован. Подобные механизмы часто используются при доступе к диску. Основное различие между различными фасадами заключается в том, нужно ли выполнять запись, и если да, то когда: немедленно или как можно позже, причем последний вариант полезнее. Надо также проверить, произошли ли изменения в каком-либо удаленном блоке, и очистить его:

```

template<typename KEY, typename VALUE, typename RESOURCE,
        typename HASHER = EHash<BUHash> > class DelayedCommitLRUCache
{
    RESOURCE& r;
    typedef LRUCache<KEY, pair<VALUE, bool>, HASHER> LRU;
    typedef typename LRU::Iterator I;
    LRU c;
    void commit(I i)
    {
        if(i->value.second) r.write(i->key, i->value.first);
        i->value.second = false;
    }
    void writeHelper(KEY const& k, VALUE const& v, bool fromWrite)
    { // коммит evictee, если есть значение
        I i = c.evicteeOnWrite(k);
        if(i != c.end()) commit(i);
        c.write(k, pair<VALUE, bool>(v, fromWrite));
    }
    DelayedCommitLRUCache(DelayedCommitLRUCache const&); // копирование не разрешено
    DelayedCommitLRUCache& operator=(DelayedCommitLRUCache const&);
};

```

```

public:
    DelayedCommitLRUCache(RESOURCE& theR, int capacity): r(theR), c(capacity)
    {assert(capacity > 0);}
    VALUE const& read(KEY const& k)
    { // проверка значения в кеше
        pair<VALUE, bool>* mv = c.read(k);
        if(!mv)
        { // если его там нет, считываем ресурс и помещаем его в кеш
            writeHelper(k, r.read(k), false);
            mv = c.read(k);
        }
        return mv->first;
    }
    void write(KEY const& k, VALUE const& v){writeHelper(k, v, true);}
    void flush(){for(I i = c.begin(); i != c.end(); ++i) commit(i);}
    ~DelayedCommitLRUCache(){flush();}
};

```

## 12.5. Вектор с $k$ -битным размером слова

Чтобы сэкономить место при необходимости представления последовательностей перечислений — например, данных ДНК, можно использовать вектор  $k$ -битных целых чисел, представляющих значения  $\in [0, 2^k - 1]$ . Это фасад, реализованный поверх набора битов. Произвольный доступ выполняется за время  $O(1)$ , а амортизированная скорость добавления равна  $O(1)$ . Но в целом это не особо полезно, поэтому в реализации выполняется лишь минимум (рис. 4.12).

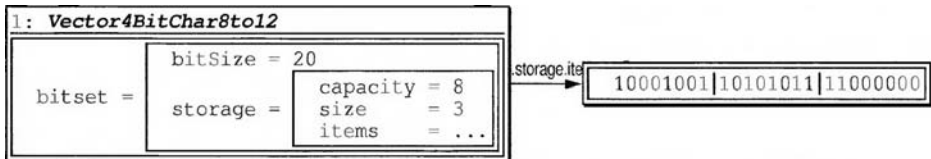


Рис. 12.4. Структура памяти вектора из 4-битных слов с байтовым хранилищем, содержащим целые числа от 8 до 12

```

template<int N, typename WORD = unsigned long long> class KBitWordVector
{
    Bitset<WORD> b;
public:
    unsigned long long getSize()const{return b.getSize()/N;}
    KBitWordVector(){};
    KBitWordVector(int n, WORD item = 0): b(n * N)
    {
        if(Bits::getValue(item, 0, N) == 0) b.setAll(0);
        else for(unsigned long long i = 0; i < getSize(); ++i) set(item, i);
    }
    WORD operator[](unsigned long long i)const
    {assert(i < getSize()); return b.getValue(i * N, N);}
};

```

```

void set(WORD value, unsigned long long i)
{assert(i < getSize()); b.setValue(value, i * N, N);}
void append(WORD value){b.appendValue(value, N);}
};

```

## 12.6. Объединение множества на интервалах

Имея разбиение интервала  $[0, n]$ , можно найти подынтервал, содержащий  $0 \leq i \leq n$ , объединить два соседних подынтервала и разделить подынтервал. Простое решение — сохранить точки разделения в динамической отсортированной последовательности, причем точкой разделения будет крайняя правая точка в интервале. Затем выполняется поиск, слияние и разделение, реализованные через преемника, удаление и вставку, соответственно:

```

class IntervalSetUnion
{
    Treap<int, bool> treap;
public:
    int find(int i){return treap.getSize() == 0 ? -1: treap.successor(i)->key;}
    void merge(int i){return treap.remove(i);}
    void split(int i){treap.insert(i, 0);}
};

```

Подобные задачи встречаются редко.

## 12.7. Создание первых $N$ простых чисел

*Решето Эратосфена* содержит список первых  $n$  чисел, где любое простое число  $\leq \sqrt{n}$  удаляет все кратные ему (см. [12.5]). Этот список при просмотре и удалении в порядке возрастания содержит только простые числа. Для повышения эффективности использования памяти список организуется как набор битов, представляющий нечетные числа  $\geq 3$ :

```

class PrimeTable
{
    long long maxN;
    Bitset<> table; // помечаются нечетные числа, начиная с 3
    long long nToI(long long n){return (n - 3)/2;}
public:
    PrimeTable(long long primesUpto): maxN(primesUpto - 1),
    table(nToI(maxN) + 1)
    {
        assert(primesUpto > 1);
        table.setAll(true);
        for(long long i = 3; i <= sqrt(maxN); i += 2)
            if(isPrime(i)) // для каждого нечетного,
                // кратного i <= k <= maxN/i, ставим пометку false
                for(long long k = i; i * k <= maxN; k += 2)
                    table.set(nToI(i * k), false);
    }
};

```

```

bool isPrime(long long n) const
{
    assert(n > 0 && n <= maxN);
    return n == 2 || (n > 2 && n % 2 && table[nToI(n)]);
}
};

```

Алгоритм выполняется за время  $O(n)$  и занимает примерно  $n/16$  байтов памяти.

## 12.8. Генерация всех возможных перестановок

Алгоритм вычисления *лексикографического преемника* перестановки:

1. Найти последний элемент, который меньше следующего элемента.
2. Поменять его местами с его преемником из элементов справа.
3. Повторить пункт 2 наоборот.

Например, в числе 045837621 меняются местами 3 и 6, а после обратного обмена получается 045861237. Элементы справа после замены располагаются в порядке убывания, поэтому алгоритм работает.

Чтобы пропустить некоторый класс перестановок, нужно отсортировать оставшиеся элементы в порядке убывания и перейти к следующей перестановке. Если не нужны перестановки, начинающиеся с 04586, пропустите их, создав 045867321 и перейдя к 045871236:

```

struct Permutator
{
    Vector<int> p;
    Permutator(int size){for(int i = 0; i < size; ++i) p.append(i);}
    bool next()
    { // поиск наибольшего i такого, что p[i] < p[i + 1]
        int j = p.getSize() - 1, i = j - 1; // начинаем с предпоследнего
        while(i >= 0 && p[i] >= p[i + 1]) --i;
        bool backToIdentity = i == -1;
        if(!backToIdentity)
        { // поиск j такого, что p[j] является следующим по величине
            // элементом после p[i]
            while(i < j && p[i] >= p[j]) --j;
            swap(p[i], p[j]);
        }
        p.reverse(i + 1, p.getSize() - 1);
        return backToIdentity; // true, если возвращаемся к наименьшей перестановке
    }
    bool advance(int i)
    {
        assert(i >= 0 && i < p.getSize());
        quickSort(p.getArray(), i + 1, p.getSize() - 1,
            ReverseComparator<int>());
        return next();
    }
};

```

Переход к следующей перестановке в худшем случае занимает время  $O(n)$ , а перестановка  $n$  элементов — амортизированное время  $O(1)$ , потому что  $(n - m)!$  из  $n!$  для перестановок требуют поменять местами  $m$  элементов, а общая средняя работа равняется:

$$\sum_{0 \leq m < n} \frac{m(n - m)!}{n!} = O(1).$$

Посещение всех перестановок занимает время  $O(nm!)$ , и это узкое место алгоритма. Алгоритм также работает с повторяющимися элементами и пригоден для работы с любыми копируемыми и поддерживающими сравнение элементами.

## 12.9. Генерация всех возможных сочетаний

Поскольку порядок в сочетании не имеет значения, для упрощения алгоритма можно работать с отсортированной последовательностью. Алгоритм вычисления лексикографического преемника комбинации  $c$  из  $m$  из  $n$  чисел  $\in [0, n - 1]$ :

1. Найти последний индекс  $i$ , для которого  $c[i] < n - m + i$ .
2. Увеличить  $c[i]$ .
3. Для любого  $j > i$  сбросить  $c[j]$  до  $c[j - 1] + 1$ .

Например, для сочетания (4, 6) вида 0145,  $i = 1$ , а следующее сочетание 0234. Пропуск в  $i$  сбрасывает число по индексу  $i$  до максимально возможных значений. Алгоритм выполняется за  $O(m)$  времени на комбинацию:

```
struct Combinator
{
    int n;
    Vector<int> c;
    Combinator(int m, int theN): n(theN), c(m, -1)
    {
        assert(m <= n && m > 0);
        skipAfter(0);
    }
    void skipAfter(int i)
    { // увеличение c[i] и сброс всех c[j], у которых j>i
        assert(i >= 0 && i < c.getSize());
        ++c[i];
        for(int j = i + 1; j < c.getSize(); ++j) c[j] = c[j - 1] + 1;
    }
    bool next()
    { // поиск самого правого c[i], которое можно увеличить
        int i = c.getSize() - 1;
        while(i >= 0 && c[i] == n - c.getSize() + i) --i;
        bool finished = i == -1;
        if(!finished) skipAfter(i);
        return finished;
    }
};
```



## 12.10. Генерация всех подмножеств

Подмножество множества размером  $m$  наиболее экономично представляется в виде битовой строки размером  $m$ , в которой присутствующие в подмножестве элементы помечены единицами. Чтобы сгенерировать все подмножества в обратном лексикографическом порядке, установите битовую строку со значением  $2^m - 1$  и уменьшайте ее до 0. Чтобы пропустить подмножество, достаточно обнулить нужные младшие биты. Операции занимают время  $O(1)$ , если в качестве битовой строки используется одно слово.

Этот метод является иллюстрацией известной методики *ранжирования/деранжирования*. Идея ее состоит в том, чтобы определить отображение интересных объектов в целые числа. Затем можно сгенерировать все объекты путем подсчета и удаления текущего целого числа.

## 12.11. Создание всех разделов

*Раздел* — это отношение эквивалентности, в котором каждый элемент принадлежит группе. Для  $n$  элементов наиболее экономичным представлением является массив  $p$  номеров групп, начиная с 0. Поскольку помещение каждого элемента в группу 0 эквивалентно помещению каждого элемента в любую другую группу, максимально допустимое значение  $p[i]$  равно:

$$m_i = \begin{cases} \max(m_{i-1}, p[i-1] + 1), & i > 0 \\ 0, & i = 0 \end{cases}.$$

Чтобы сгенерировать лексикографического преемника, нужно увеличить крайний правый поддерживающий увеличение элемент, а остальные задать равными нулю. Например, для последовательности 010123 второй 0 — самый правый увеличиваемый элемент, а следующий раздел — 011000. Чтобы пропустить  $i$ , сделайте значения справа от него максимально возможными и сгенерируйте преемник. Алгоритм выполняется за  $O(m)$  времени на раздел:

```
struct Partitioner
```

```
{
    Vector<int> p;
    Partitioner(int n): p(n, 0) {assert(n > 0);}
    bool skipAfter(int k)
    { // установка максимальных значений завершающих элементов и переход дальше
      assert(k >= 0 && k < p.getSize());
      for(int i = k; i < p.getSize(); ++i) p[i] = i;
      return next();
    }
    bool next()
    { // поиск крайнего правого p[j], которое можно увеличить
      int m = 0, j = -1;
      for(int i = 0; i < p.getSize(); ++i)
      {
          if(p[i] < m) j = i;
          m = max(m, p[i] + 1);
      }
    }
}
```

```
bool finished = j == -1;
if(!finished)
{ // увеличение и сброс хвоста
    ++p[j];
    for(int i = j + 1; i < p.getSize(); ++i) p[i] = 0;
}
return finished;
}
};
```

## 12.12. Генерация всех зависимых объектов

Пусть нам необходимо сгенерировать все синтаксические деревья для функции пяти переменных, используя только три уровня операторов «+» и «\*». Принцип генерации всех значений любого типа таков:

1. Проверить, изоморфен ли тип другому легко генерируемому типу.
2. Искать самую экономичную форму представления.
3. Использовать лексикографический порядок.
4. Не сохранять состояние.

Если ни один метод не помогает, можно сгенерировать менее ограниченный класс объектов и принять/отклонить каждый из них, в конечном итоге получить все двоичные последовательности некоторой большой длины и вывести те из них, которые соответствуют нужным объектам. Но с какого-то момента этот метод оказывается весьма неэффективным.

## 12.13. Примечания по реализации

Использование кеша LRU с отложенной отправкой — это оригинальное решение, которое оказывается удобным для алгоритмов работы с дисками (см. главу 13. *Алгоритмы для работы с внешней памятью*). Несмотря на простую алгоритмическую идею и доступность компонентов, правильная реализация достаточно сложна, и за годы работы я обнаружил в ней несколько ошибок.

Другие реализации довольно просты, и о них можно почитать в любом источнике.

## 12.14. Комментарии

Удвоение массива можно сделать эффективным в худшем случае, но это некрасиво, неэффективно с точки зрения памяти и бесполезно. Когда массив заполнен до половины, нужно создать еще один, в два раза больший. При добавлении нового элемента он добавляется в оба, а имеющийся в старом массиве элемент копируется из него в новый. По мере заполнения старого массива в новый скопируются все его элементы, и он будет заполнен наполовину. Удалите старый массив и повторите процесс.

Идея, лежащая в основе реализации вектора с  $k$ -битной длиной слова, оригинальна, хотя и вполне очевидна.

## 12.15. Советы по дополнительной подготовке

Внедрите автоматизированные тесты для алгоритмов генерации. Нужно использовать малое  $n$  и проверить, что все возможные числа генерируются в правильном порядке.

## 12.16. Список рекомендуемой литературы

- 12.1. Brass P. (2008). Advanced Data Structures. Cambridge University Press.
- 12.2. Overmars M. H. (1983). The Design of Dynamic Data Structures. Springer.
- 12.3. Nayak A., & Stojmenovic I. (Eds). (2007). Handbook of Applied Algorithms: Solving Scientific, Engineering, and Practical Problems. Wiley-IEEE Press.
- 12.4. Wikipedia (2013). Cache algorithms. [http://en.wikipedia.org/wiki/Cache\\_algorithms](http://en.wikipedia.org/wiki/Cache_algorithms). Accessed May 18, 2013.
- 12.5. Wikipedia (2016). Sieve of Eratosthenes. [https://en.wikipedia.org/wiki/Sieve\\_of\\_Eratosthenes](https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes). Accessed October 30, 2016.

## 13. Алгоритмы для работы с внешней памятью

### 13.1. Введение

Важно понимать принципы работы с файлами на диске, особенно с последовательностью их расположения и таблицами размещения. Именно этого я и коснусь в деталях реализации.

### 13.2. Диски и файлы

*Жесткий диск* — это постоянная память с медленным доступом. Технологии их создания различны, но всегда справедливо следующее:

- ◆ поддерживается произвольный доступ — т. е. *поиск*;
- ◆ запись работает незначительно или существенно медленнее, чем чтение;
- ◆ чтение и запись блоков смежных байтов работает на несколько порядков быстрее, чем доступ к данным побайтово;
- ◆ и поиск, и *передача данных* выполняются долго, причем передача — дольше;
- ◆ для работы с данными необходимо скопировать их во внутреннюю память.

С логической точки зрения *файлом* называют массив байтов. В C++ файл — это динамический массив, который запоминает последний доступный индекс и увеличивает его после операции ввода/вывода. ОС сама обрабатывает динамические особенности этой структуры. Приведенная далее реализация предназначена для доступа к блокам байтов. В ней используется подмножество файлового API C++, реализующего нужную абстракцию. Принятые решения описаны в комментариях к коду. Обратите внимание на следующие моменты:

- ◆ ОС ограничивает количество файлов, с которыми может работать один исполняемый файл, поэтому закрывайте ненужные файлы;
- ◆ функция C++ `fstream` не может открыть файл для записи, если разрешения файла этого не позволяют;
- ◆ указатель произвольного доступа потока чтения называется *указателем получения*, а потока записи — *указателем размещения*. В функции `fstream` они эквивалентны, поскольку используется один буфер. Поэтому может использоваться любой набор методов доступа. Указатели сбрасываются после каждой операции записи, поскольку в противном случае поведение чтения после записи может быть непредсказуемо;
- ◆ Функция `fstream` может (это не гарантировано стандартом, но реализовано) использовать буфер размера `BUFSIZ` (макрос из `<stdio>`). Таким образом, чтение нескольких

смежных блоков меньшего размера по-прежнему выполняется быстро, но запись выполняется медленно из-за необходимости очистки, о которой говорилось ранее. Также вероятно, что у блоков, превышающих размер буфера, ввод/вывод выполняется напрямую, а не разбивается на фрагменты `BUFSIZ`.

```
class File
{
    fstream f; // объект ввода/вывода C++
    long long size; // кеширование для эффективности
    void create(const char* filename){ofstream dummy(filename, ios::trunc);}
    File(File const&); // копирование не выполняется
    File& operator=(File const&);
    void goToEnd()
    {
        f.seekg(0, ios::end);
        assert(f); // проверка внешних проблем
    }
public:
    static bool exists(const char* filename){return bool(ifstream(filename));}
    static void remove(const char* filename)
    {
        if(exists(filename))
        {
            int returnCode = std::remove(filename);
            assert(returnCode == 0); // проверка внешних проблем
        }
    }
    File(const char* filename, bool truncate)
    {
        if(truncate || !exists(filename)) create(filename);
        f.open(filename, ios::binary | ios::in | ios::out);
        assert(f); // проверяем, что файл заблокирован и т. д.
        // вычисление размера
        goToEnd();
        size = getPosition();
        setPosition(0); // возврат к началу
    }
    long long getPosition(){return f.tellg();}
    long long getSize()const{return size;}
    long long bytesToEnd(){return getSize() - getPosition();}
    void setPosition(long long position)
    {
        assert(0 <= position && position <= getSize());
        f.seekg(position);
        assert(f); // проверка внешних проблем
    }
    void read(unsigned char* buffer, long long n)
    {
        assert(n > 0 && n <= bytesToEnd());
        f.read((char*)buffer, n);
        assert(f); // проверка внешних проблем
    }
}
```

```
void write(unsigned char* buffer, long long n)
{
    assert(n > 0 && n <= bytesToEnd()); // защита от ошибок
    f.write((char*)buffer, n);
    f.flush();
    assert(f); // проверка внешних проблем
}

void append(unsigned char* buffer, long long n)
{
    goToEnd();
    size += n; // это делается в начале, чтобы избежать
              // ошибок при проверке
    write(buffer, n); // запись предпоследнего значения
}

};
```

На моем компьютере размер блока файловой системы равен 4096 байтов, а логического блока — 512 байтов, что равно `BUFSIZ`. По всей видимости, компилятор пытается установить для `BUFSIZ` максимально возможное значение — другие разработчики на разных форумах говорят, что `BUFSIZ` ∈ [512, 4096]. Похоже, что из-за увеличения объемов в современных дисках и файловых системах используется размер 4096 вместо старого 512, с которым размер файлов становился меньше и экономилось место (рис. 13.1)<sup>1</sup>.

Байты	Секунды	Замедление по сравнению с предыдущим
4	0,66	
8	0,65	0,98
16	0,64	0,98
32	0,65	1,01
64	0,66	1,03
128	0,68	1,02
256	0,71	1,05
512	0,74	1,04
1024	0,83	1,12
2048	1,06	1,28
4096	1,47	1,39
8192	2,28	1,55
16384	3,88	1,70
32768	7,15	1,84
65536	13,78	1,93

Рис. 13.1. Время выполнения  $10^5$  чтений случайных блоков размера  $B$  из файла размером  $10^7$

<sup>1</sup> См., например, <https://support.microsoft.com/en-us/help/140365/default-cluster-size-for-ntfs-fat-and-exfat> и <https://unix.stackexchange.com/questions/178899/optimizing-logical-sector-size-for-physical-sector-size-4096-hdd>).

Эксперименты показывают:

- ♦ оптимальный размер блока  $B$  может превышать `BUFSIZ`;
- ♦ предельная стоимость постепенного удвоения  $B$  стремится к 2. Значение  $B$  не кажется важным, если оно не слишком сильно превышает размер блока файловой системы, но при больших размерах блока ситуация не столь радужная. У более новых твердотельных дисков (SSD) фрагментация не является проблемой из-за отсутствия операции поиска<sup>2</sup>.

На современных компьютерах используется безопасный вариант  $B = \max(\text{BUFSIZ}, 4096)$ . Размер блока получается не слишком большим и не слишком маленьким и используется по умолчанию для всех дисковых алгоритмов и структур данных. Можно просто довериться значению `BUFSIZ`. Если вы готовы пожертвовать переносимостью, ознакомьтесь с API конкретной используемой ОС.

Использование внешней памяти дают следующие преимущества:

- ♦ *надежность хранения* — любая структура данных на диске внешней памяти сохраняется после вычисления;
- ♦ увеличенный объем памяти — для приложений, работающих с большими объемами, внутренней памяти всегда мало.

Алгоритмы работы с внешней памятью заточены на эффективность, ведь если не организована работа с непрерывными данными, программа замедляется на порядки. Чаще всего реализуются задачи вроде чтения файла, выполнения вычислений и записи результата в другой файл. Это наиболее эффективный способ работы с файлами. Но иногда — например, при работе с базами данных, вам нужно работать только с небольшой частью большого файла.

### 13.3. Структура файла

*Сериализация* — это превращение структуры данных в последовательность байтов. Если предположить, что вся структура данных хранится в одном файле, для моделирования любой структуры данных можно использовать следующие подходы:

- ♦ целые числа, которые обозначают смещение от начала файла, могут использоваться как указатели;
- ♦ элементы и массивы могут иметь известную фиксированную или сохраненную переменную длину;
- ♦ файл разбивается на часть переменной длины и часть массива элементов фиксированной длины для обеспечения быстрых непрерывных операций. Первая часть дополняется, чтобы элементы массива не попадали на границы блоков файловой системы;
- ♦ использование заголовка фиксированной длины для указания формата файла, контрольных сумм, разрешений и других полезных данных;
- ♦ использование файлов со значениями, разделенными запятыми (comma-separated value, CSV), для хранения простых матричных данных;

---

<sup>2</sup> См., например, <https://www.pcgamer.com/should-i-defrag-my-ssd>.

- ◆ использование JSON- или XML-подобного кодирования, чтобы сделать представление сложной структуры данных удобочитаемым для человека. Двоичные и текстовые файлы обрабатываются так же, если в них используются только печатные символы;
- ◆ чтобы избежать проблем с переносимостью между системами с прямым и обратным порядком байтов, в которых байты целого числа просматриваются в разном порядке, целые числа можно представлять в байтовом или поддерживающем переинтерпретацию коде (см. главу 13. *Сжатие*). При разработке формата файла каждый байт определяется логически. В алгоритмах этой главы такое представление игнорируется для простоты.

Например, файл может иметь следующую структуру:

```
|HEADER|LENGTH|{JSON_VAR: VAL1, JSON_ARR: [VAL2, VAL3]}|.
```

## 13.4. Работа с файлами CSV

Файлы CSV легко создавать из строковых матриц:

```
void createCSV(Vector<Vector<string> > const& matrix, const char* filename)
{
    ofstream file(filename);
    assert(file);
    for(int i = 0; i < matrix.getSize(); ++i)
    {
        for(int j = 0; j < matrix[i].getSize(); ++j)
        {
            if(j > 0) file << ",";
            file << matrix[i][j];
        }
        file << endl;
    }
}
```

Такие файлы особенно полезны для записи и анализа любых экспериментальных измерений. Довольно часто стоит задача запуска нескольких алгоритмов для решения нескольких задач с последующим сбором каких-то метрик (значений показателей) — таких как оценка качества решения, время выполнения и т. п. Для чтения и записи таких файлов полезно иметь возможность преобразовывать число с плавающей запятой в строку с предсказуемым форматированием:

```
string toStringDouble(double x)
{
    stringstream s;
    s << setprecision(17) << x;
    string result;
    s >> result;
    return result;
}
```

Допустим, нам нужна возможность разделить комбинированную матрицу данных, получив одну матрицу для каждой метрики, чтобы их впоследствии можно было анализировать. В этом случае схема матрицы принимает вид:



Задача 1	Алгоритм 1	Метрика 1	Метрика 2	Алгоритм 2	Метрика 1	Метрика 2
Задача 2	Алгоритм 1	Метрика 1	Метрика 2	Алгоритм 2	Метрика 1	Метрика 2

Подобные матрицы удобно записывать построчно, одновременно запуская все алгоритмы для одной задачи. Обычно это работает более эффективно, чем запускать алгоритмы по отдельности, т. к. ввод данных в этом случае нагружается меньше. Вызывающая сторона должна знать названия метрик. Индивидуальный макет метрики тогда выглядит следующим образом:

	Алгоритм 1	Алгоритм 2
Задача 1	Метрика	Метрика
Задача 2	Метрика	Метрика

```
Vector<Vector<Vector<string>>> >> splitRegularMatrix(
    Vector<Vector<string>> > const& matrix, int nMetrics)
{ // должно быть правильное количество столбцов в каждой строке
  assert(nMetrics > 0); // вычисление количества строк, столбцов и метрик
  for(int i = 0; i < matrix.getSize(); ++i)
    assert((matrix[i].getSize() - 1) % (nMetrics + 1) == 0);
  int nNewRows = matrix.getSize() + 1,
      nNewColumns = 1 + (matrix[0].getSize() - 1)/(nMetrics + 1);
  // разделение
  Vector<Vector<Vector<string>>> >> result(nMetrics,
      Vector<Vector<string>> (nNewRows, Vector<string>(nNewColumns)));
  for(int i = 0; i < nMetrics; ++i)
  { // копирование имен алгоритмов из первой строки
    for(int c = 1; c < nNewColumns; ++c) result[i][0][c] =
        matrix[1][1 + (c - 1) * (nMetrics + 1)];
    for(int r = 1; r < nNewRows; ++r)
    { // копирование задач metricNames
      result[i][r][0] = matrix[r - 1][0];
      // копирование соответствующих столбцов
      for(int c = 1; c < nNewColumns; ++c) result[i][r][c] =
          matrix[r - 1][2 + i + (c - 1) * (nMetrics + 1)];
    }
  }
  return result;
}
```

Для целей анализа можно создать файл (например, в OpenOffice Calc) с одной метрикой и формулой сравнения по рангам (см. главу 21. *Вычислительная статистика*):

1. Преобразуем баллы в ранги (чем он меньше, тем лучше).
2. Усредняем.
3. Выполняем ранжирование по средним рангам.

Потребуется несколько вспомогательных функций в частности, чтобы преобразовать номер столбца индекса в имя ячейки. Чтобы понять это, обратитесь к документации по

Calc. Имейте в виду, что формулы электронных таблиц для связей дают более низкий, а не средний ранг, но это мало что меняет для статистического анализа:

```
string cellValue(int r, int c)
{ // конвертирование ячейки и ссылки
    return "INDIRECT(ADDRESS(" + to_string(r + 1) + ";" +
        to_string(c + 1) + ";4))";
}
string fixNumber(int r, int c)
{ // конвертирование ячейки и ссылки и преобразование в число
    // из научной записи
    return "VALUE(TRIM(" + cellValue(r, c) + "))";
}
string rankFormula(int r, int c, int c0, int cLast)
{ // классификация столбца c в диапазоне [c0, cLast] в этой строке
    return "RANK(" + cellValue(r, c) + ";" + cellValue(r, c0) +
        ":" + cellValue(r, cLast) + ";1)";
}
string minFormula(int r, int c0, int cLast)
{ return "MIN(" + cellValue(r, c0) + ":" + cellValue(r, cLast) + ")"; }
string aveFormula(int r0, int c0, int rLast, int cLast)
{ // вычисление среднего по строке в столбце
    assert(r0 == rLast || c0 == cLast);
    return "AVERAGE(" + cellValue(r0, c0) + ":" +
        cellValue(rLast, cLast) + ")";
}
```

Запустите алгоритм несколько раз, меняя размер выборки таким образом, чтобы не создавать искусственной статистической достоверности (рис. 13.2):

```
void augmentComparableMatrix(Vector<Vector<string> >& matrix, int nRepeats = 1)
{ // пусть первая строка - это имя алгоритма + имя задачи
    // в первом столбце
    int nDataRows = matrix.getSize() - 1, nColumns = matrix[0].getSize();
    // добавление пустой строки в качестве разделителя
    matrix.append(Vector<string>(nColumns, ""));
    // извлечение численных значений из всех столбцов
    int fixedStart = matrix.getSize();
    for(int r = 0; r < nDataRows; ++r)
    {
        Vector<string> newRow(nColumns, "");
        for(int c = 1; c < nColumns; c++)
            newRow[c] = string("=") + fixNumber(1 + r, c);
        matrix.append(newRow);
    }
    // добавление пустой строки в качестве разделителя
    matrix.append(Vector<string>(nColumns, ""));
    // задание формул ранжирования для всех точек данных
    for(int r = 0; r < nDataRows; ++r)
    {
        Vector<string> newRow(nColumns, "");
        for(int c = 1; c < nColumns; c++) newRow[c] = string("=") +
            rankFormula(fixedStart + r, c, 1, nColumns - 1);
    }
}
```

```
        matrix.append(newRow);
    }
    // усреднение рангов в каждом столбце
    Vector<string> newRow2(nColumns, "Ave Ranks");
    for(int c = 1; c < nColumns; c++) newRow2[c] = string("=") + aveFormula(
        matrix.getSize() - nDataRows, c, matrix.getSize() - 1, c);
    matrix.append(newRow2);
    // ранжирование средних значений
    Vector<string> newRow(nColumns, "Total Rank");
    for(int c = 1; c < nColumns; c++) newRow[c] = string("=") +
        rankFormula(matrix.getSize() - 1, c, 1, nColumns - 1);
    matrix.append(newRow);
    // поиск достаточно большой rankDifference
    double maxDiff = findNemenyiSignificantAveRankDiff(nColumns - 1,
        nDataRows/nRepeats);
    Vector<string> newRow3(nColumns, toStringDouble(maxDiff));
    newRow3[0] = "Significant Diff";
    matrix.append(newRow3);
    Vector<string> newRow4(nColumns, "Cutoff Rank");
    for(int c = 1; c < nColumns; c++) newRow4[c] = string("=") +
        minFormula(matrix.getSize() - 3, 1, nColumns - 1) + "+" +
        cellValue(matrix.getSize() - 1, c);
    matrix.append(newRow4);
    Vector<string> newRow5(nColumns, "Same as Best");
    for(int c = 1; c < nColumns; c++) newRow5[c] = string("=") + "IF(" +
        cellValue(matrix.getSize() - 1, c) + ">" +
        cellValue(matrix.getSize() - 4, c) + ";1;0)";
    matrix.append(newRow5);
}
```

	Child	Computer
Animal Recognition	1	5
Simple Multiplication	2	1
	1	5
	2	1
	1	2
	2	1
Ave Ranks	1.5	1.5
Total Rank	1	1
Significant Diff	1.6928008593	1.6928008593
Cutoff Rank	3.1928008593	3.1928008593
Same as Best	1	1

Рис. 13.2. Образец выходного файла. Обратите внимание на форматирование чисел

Объедините все в функцию. В результирующих файлах имена метрик представляют собой префиксы, а имена файлов — суффиксы:

```
void createAugmentedCSVFiles(Vector<Vector<string> > const& matrix,
    Vector<string> const& metricNames, string filename, int nRepeats = 1)
```

```

{
    Vector<Vector<Vector<string> > > pieces (
        splitRegularMatrix(matrix, metricNames.getSize()));
    for(int i = 0; i < pieces.getSize(); ++i)
    {
        augmentComparableMatrix(pieces[i], nRepeats);
        createCSV(pieces[i], (metricNames[i] + "_" + filename).c_str());
    }
}

```

## 13.5. Модель ввода/вывода

Быстрее всего работать со смежными данными. Это всё равно что везти из супермаркета тележку с продуктами, а не носить по одному товару за раз. Распишем подробнее:

- ◆ стоимость поиска равна нулю, потому что он выполняется только перед более медленной передачей данных;
- ◆ стоимость чтения равна стоимости записи. Это справедливо для старых магнитных дисков, но для флеш-памяти или SSD-дисков — нет;
- ◆ данные передаются блоками размером  $\leq B$ , при этом  $B$  выбирается таким образом, что его увеличение не приводит к дальнейшему ускорению. Каждая передача есть операция *ввода/вывода*;
- ◆ размер диска не ограничен, а объем памяти равен  $M$  и значительно превышает  $B$ ;
- ◆ стоимость операций с внутренней памятью равна нулю, потому что операции ввода/вывода работают на несколько порядков медленнее;

Эта модель справедлива для внутренней и кеш-памяти, за исключением постоянных коэффициентов. Можно предположить, что количество операций ввода/вывода наименьшее. Количество операций (см. [13.3]):

- ◆ сканирование —  $O\left(\frac{n}{B}\right)$ ;
- ◆ операции с отображением —  $O(\log_B(n))$ ;
- ◆ сортировка —  $O\left(\frac{n}{B} \log_{M/B}\left(\frac{n}{B}\right)\right)$ ;
- ◆ приоритетные очереди — амортизированная стоимость вставки/удаления  $O\left(\frac{1}{B} \log_{M/B}\left(\frac{n}{B}\right)\right)$ .

В целом эта модель ввода/вывода точна, но у нее есть недостатки:

- ◆ у дисков бывают большие быстрые области кеша;
- ◆ операция поиска занимает большую часть времени выполнения, если операций поиска выполняется намного больше, чем операций ввода/вывода;
- ◆ ОС может размещать большие файлы в блоках размером  $< B$ . Размер может составлять всего 512 байтов, а соседние блоки не обязательно должны быть непрерывны-

ми, что фактически уменьшает  $B$  до размера блока файловой системы. Размер этого блока обычно равен размеру односимвольного текстового файла. Его увеличение ускоряет доступ, но приводит к ненужным затратам места, потому что файлы в общей своей массе крошечные;

♦ вычисления внутренней памяти могут быть узким местом.

Многие алгоритмы и структуры данных, например  $B$ -дерево (о нем далее в этой главе) занимают  $O(B)$  внутренней памяти, поэтому  $B$  не может быть слишком большим. Структура файла, поддерживающего модель ввода/вывода, представляет собой массив блоков переменной длины большого фиксированного размера. Структуры данных, работающие с такими файлами, могут кодировать любые данные конфигурации в заголовок фиксированного размера или использовать для этого отдельный файл. То есть эта реализация обеспечивает возможность использования заголовков. Чтобы избежать смещения блоков, мы кодируем первые несколько — при необходимости с отступами.

Общее требование состоит в том, чтобы сохранить все необходимое для воссоздания наблюдаемого состояния структуры данных, включая любые параметры. Сам файл блока кодирует размер блока на случай, если файл считывается структурой данных, которой нужен другой размер блока. Предполагается, что подойдет и желаемый размер, и текущий. У вектора оба эти размера должны быть кратны размеру элемента. Это обеспечивает переносимость файлов, когда, например, изменяется размер блока файловой системы.

Любая операция, которая должна считывать данные, выполняет более одного ввода/вывода, поэтому важно эффективно использовать внутреннюю память, не занимая ее слишком много. Поэтому вы можете использовать кеш часто используемых блоков, чтобы уменьшить число обращений к диску. Асимптотически оптимальная стратегия заключается в хранении в буфере нескольких наименее использовавшихся блоков, используя кеш LRU (см. главу 12. *Разные алгоритмы и методы*):

```
class BlockFile
{
    File f;
    long long size; // количество блоков, не считая заголовочных
    enum{SELF_HEADER_SIZE = 4};
    int headerSize, blockSize;
    int getNHeaderBlocks() const
    {
        return ceiling(SELF_HEADER_SIZE + headerSize, blockSize);
    }
    void setBlock(long long blockId) {f.setPosition(blockId * blockSize);}
    void write(long long blockId, Vector<unsigned char> const& block)
    {
        assert(block.getSize() == blockSize);
        setBlock(blockId);
        f.write(block.getArray(), blockSize);
    }
    Vector<unsigned char> read(long long blockId)
    {
        Vector<unsigned char> block(blockSize);
        setBlock(blockId);
        f.read(block.getArray(), blockSize);
        return block;
    }
}
```

```

typedef DelayedCommitLRUCache<long long, Vector<unsigned char>, BlockFile>
    CACHE;
friend CACHE; // чтобы разрешить доступ для чтения и записи
CACHE cache; // объявленный последним уничтожается первым
Vector<unsigned char> getHelper(long long blockId, int start, int n)
{
    Vector<unsigned char> data(n);
    Vector<unsigned char> const& block = cache.read(blockId);
    for(int i = 0; i < n; ++i) data[i] = block[start + i];
    return data;
}
void setHelper(Vector<unsigned char> const& data, long long blockId,
    int start)
{
    Vector<unsigned char> block = cache.read(blockId);
    for(int i = 0; i < data.getSize(); ++i) block[start + i] = data[i];
    cache.write(blockId, block);
}
public:
constexpr static int targetBlockSize(){return max<int>(BUFSIZ, 4096);}
int getBlockSize()const{return blockSize;}
// блоки заголовка в размер не входят
long long getSize()const{return size - getNHeaderBlocks();}
BlockFile(string const& filename, int theBlockSize, int cacheSize,
    int theHeaderSize = 0): f(filename.c_str(), false), size(0),
    headerSize(theHeaderSize), blockSize(theBlockSize),
    cache(*this, cacheSize)
{
    assert(blockSize > 0);
    long long fileSize = f.getSize();
    if(fileSize > 0)
    { // файл уже существует, и нельзя использовать getHelper,
      // потому что размер блока еще не известен
        Vector<unsigned char> selfHeader(SELF_HEADER_SIZE);
        f.setPosition(0);
        f.read(selfHeader.getArray(), SELF_HEADER_SIZE);
        blockSize = ReinterpretDecode(selfHeader);
        assert(blockSize > 0 && fileSize % blockSize == 0); // простая проверка
        size = fileSize/blockSize;
    }
    else
    { // добавляем блоки заголовков и записываем blockSize
      // в собственный заголовок
        for(int i = 0; i < getNHeaderBlocks(); ++i) appendEmptyBlock();
        setHelper(ReinterpretEncode(blockSize, SELF_HEADER_SIZE), 0, 0);
    }
}
void appendEmptyBlock()
{
    ++size;
    Vector<unsigned char> block(blockSize, 0);

```

```

        f.append(block.getArray(), blockSize);
    }
    Vector<unsigned char> get(long long blockId, int start, int n)
    { // для блоков без заголовка
        assert(0 <= blockId && blockId < getSize() && n > 0 && start >= 0 &&
            start + n <= blockSize);
        return getHelper(blockId + getNHeaderBlocks(), start, n);
    }
    void set(Vector<unsigned char> const& data, long long blockId, int start)
    { // для блоков без заголовка
        assert(0 <= blockId && blockId < getSize() && start >= 0 &&
            start + data.getSize() <= blockSize);
        setHelper(data, blockId + getNHeaderBlocks(), start);
    }
    void writeHeader(Vector<unsigned char> const& header)
    { // вызывающий заголовок может занимать несколько блоков
        assert(header.getSize() == headerSize);
        for(int i = 0, toWrite = headerSize; i < getNHeaderBlocks(); ++i)
        {
            int start = i == 0 ? SELF_HEADER_SIZE : 0,
                n = min(toWrite, blockSize - start);
            Vector<unsigned char> headerBlockData(n);
            for(int j = 0; j < n; ++j) headerBlockData[j] =
                header[(headerSize - toWrite) + j];
            setHelper(headerBlockData, i, start);
            toWrite -= n;
        }
    }
    Vector<unsigned char> readHeader()
    { // вызывающий заголовок может занимать несколько блоков
        assert(headerSize > 0);
        Vector<unsigned char> header;
        for(int i = 0, toRead = headerSize; i < getNHeaderBlocks(); ++i)
        {
            int start = i == 0 ? SELF_HEADER_SIZE : 0,
                n = min(toRead, blockSize - start);
            header.appendVector(getHelper(i, start, n));
            toRead -= n;
        }
        return header;
    }
};

```

При чтении и записи используется менее одного ввода/вывода на операцию. Обратите внимание:

- ◆ структуры данных внешней памяти могут содержать только элементы POD или, в более общем случае, элементы, для которых пользователь предоставляет сериализатор, выполняющий преобразование в последовательность байтов известного фиксированного размера;
- ◆ ссылки на элементы кеша становятся недействительными при замене соответствующих блоков, и это может произойти в одной и той же строке кода. Например,

для вектора внешней памяти  $v$  с размером кеша 1,  $v[i] = v[j]$  присваивается мертвой ссылке, если  $v[i]$  получает текущую ссылку на блок, а  $v[j]$  вызывает замену блока. То есть методы доступа не могут безопасно возвращать элементы по ссылке. Это также может произойти в рекурсии, когда дочерние вызовы делают недействительными ссылки родительских вызовов.

Чтобы использовать  $D$  одновременно доступных дисков, нужно создать суперблок размера  $DB$  и работать с  $D$  подблоками параллельно (здесь это не реализовано). Обычно какой-то контроллер или другой слой обрабатывает эту конфигурацию (она называется *RAID0*), и вся структура отображается для приложения как один диск. Некоторые алгоритмы могут быть более эффективными при прямом доступе ко всем дискам, но это ускорение обычно незначительно и не стоит увеличения сложности.

Когда значение  $B$  неизвестно, модель ввода/вывода называется *кеш-независимой*. Алгоритм игнорирует кеш, если его замедление составляет  $O(1)$  по сравнению с оптимальным алгоритмом, который работает с известным  $B$ . Чтобы это работало, модель предполагает, что ОС передает данные блоками оптимального размера. Основные строительные блоки:

- ◆ при последовательном сканировании массива выбираются оптимальные  $n/B$  операций ввода/вывода;
- ◆ разделяй и властвуй: разделение в конечном итоге доходит до блоков размером меньше  $B$ , т. е. сортировка слиянием и быстрая сортировка *кеш-независимы* и выполняются за  $O\left(\frac{n}{B} \lg\left(\frac{n}{B}\right)\right)$  операций ввода/вывода.

Несмотря на важность удобства кеширования, сложные алгоритмы, использующие кеш, реализовывать не стоит:

- ◆ постоянные коэффициенты теряются из-за отсутствия информации о большом значении  $B$ ;
- ◆ в случае использования диска встроенной памяти, выигрыш незначителен, потому что в ней кеши большие, постоянные различия коэффициентов малы, а планировщик задач ОС удаляет кешированные данные.

## 13.6. Вектор внешней памяти

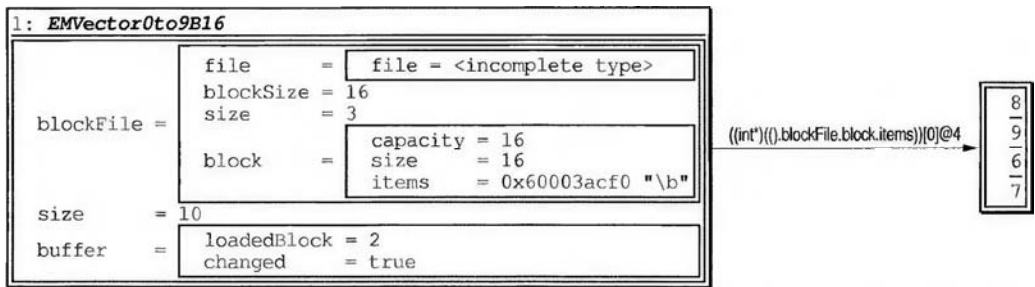
Допустим, нам нужно реализовать векторные операции, но с сохранением элементов на диске. Для повышения эффективности элементы надо разделять на блоки и при необходимости загружать весь блок в память, используя указанную пользователем стратегию буферизации. Для обработки незаполненных блоков в конце файла нужно сохранить в заголовке количество дополнительных элементов, которые могут поместиться в файл без увеличения его длины.

Элементы преобразуются в байты с помощью предоставленного пользователем сериализатора. По умолчанию выполняется операция приведения, которая предотвращает обмен файлами с компьютерами с другим порядком байтов, но это редкий вариант использования.



Вот некоторые различия по сравнению с вектором в памяти:

- ◆ операция добавления не выполняет удвоение, а выделяет еще один блок, когда это необходимо;
- ◆ используется метод `set`, потому что значение по ссылке не возвращается;
- ◆ операция удаления уменьшает внутренний размер, но не уменьшает использование памяти, т. к. в стандартной C++ API подобный подход не поддерживается. Пользователи могут сделать это самостоятельно путем перестроения;
- ◆ для работы нужно имя файла. Это способствует надежности хранения данных, но запрещает создание копий.



**Рис. 13.3.** Структура памяти вектора внешней памяти (EMVector) с  $B = 16$  байтов (4 числа `int`) после вставки целых чисел от 0 до 9. Последние два числа (8 и 9) находятся в последнем блоке, а остальные его элементы — мусор

Удобный сериализатор по умолчанию — это приведение, для которого требуется тип элемента POD. Если не менять архитектуру, то оно работает нормально, но в архитектуры с другим порядком следования байтов решение не переносится. Для улучшения переносимости нужно разрешить пользовательские сериализаторы — например, преобразующие элемент в последовательность байтов, используя код переинтерпретации (см. главу 15. Сжатие):

```
template<typename POD> struct CastSerializer // код не является переносимым
{ // следующую строку нужно раскомментировать, когда через несколько лет
  // она будет поддерживаться
  // CastSerializer(){assert(is_trivially_copyable<POD>::value);}
  constexpr static int byteSize(){return sizeof(POD);}
  POD operator()(Vector<unsigned char> const& bytes)
  {
    assert(bytes.getSize() == byteSize());
    POD item;
    for(int i = 0; i < byteSize(); ++i)
      ((unsigned char*)&item)[i] = bytes[i];
    return item;
  }
  Vector<unsigned char> operator()(POD const& item)
  {
    Vector<unsigned char> bytes(byteSize());
    for(int i = 0; i < byteSize(); ++i)
      bytes[i] = ((unsigned char*)&item)[i];
  }
};
```

```

        return bytes;
    }
};

template<typename POD> struct IntegralSerializer
{ // типичная задача
    IntegralSerializer() {assert(is_integral<POD>::value);}
    constexpr static int byteSize() {return sizeof(POD);}
    POD operator()(Vector<unsigned char> const& bytes)
    {
        assert(bytes.getSize() == byteSize());
        return ReinterpretDecode(bytes);
    }
    Vector<unsigned char> operator()(POD const& item)
    {return ReinterpretEncode(item, byteSize());}
};

template<typename POD, typename SERIALIZER = CastSerializer<POD> >
class EMVector
{
    BlockFile blockFile; // должен быть первым
    long long size;
    int itemsPerBlock() const {return blockFile.getBlockSize()/sizeof(POD);}
    long long block(long long i) {return i/itemsPerBlock();}
    long long index(long long i) {return i % itemsPerBlock();}
    SERIALIZER s;
    enum {HEADER_SIZE = 4};
    int extraItems() const {return blockFile.getSize() * itemsPerBlock() - size;}
    static int calculateBlockSize()
    { // если строгое равенство не выполняется, спуск вниз,
      // иначе переход через
        int result = BlockFile::targetBlockSize()/SERIALIZER::byteSize();
        if(result == 0) ++result; // вниз двигаться нельзя
        return result * SERIALIZER::byteSize();
    }
    EMVector(EMVector const&); // копирование не разрешено
    EMVector& operator=(EMVector const&);
public:
    long long getSize() {return size;}
    EMVector(string const& filename, int cacheSize = 2): size(0),
        blockFile(filename, calculateBlockSize(), cacheSize, HEADER_SIZE)
    {
        assert(blockFile.getBlockSize() % SERIALIZER::byteSize() == 0);
        // проверка, существует ли файл, - заголовок
        // определяет количество дополнительных элементов
        if(blockFile.getSize() > 0) size = blockFile.getSize() *
            itemsPerBlock() - ReinterpretDecode(blockFile.readHeader());
    }
    ~EMVector()
    { // записываем количество дополнительных элементов в заголовок
        Vector<unsigned char> header =
            ReinterpretEncode(extraItems(), HEADER_SIZE);

```

```

        blockFile.writeHeader(header);
    }
    void append(POD const& item)
    {
        ++size;
        if(extraItems() < 0) blockFile.appendEmptyBlock();
        set(item, size - 1);
    }
    void set(POD const& item, long long i)
    {
        assert(i >= 0 && i < size);
        blockFile.set(s(item), block(i), index(i) * SERIALIZER::byteSize());
    }
    POD operator[](long long i)
    {
        assert(i >= 0 && i < size);
        return s(blockFile.get(block(i), index(i) * SERIALIZER::byteSize(),
            SERIALIZER::byteSize()));
    }
    void removeLast()
    {
        assert(size > 0);
        --size;
    }
};

```

За счет кеширования произвольный доступ занимает время  $1/B$  и один ввод/вывод для смежных и случайных местоположений соответственно.

## 13.7. Сортировка

Вот простой способ эффективной сортировки (рис. 13.4):

1. Разделить вектор на  $Q + 1$  фрагментов по  $C$  элементов каждый (в последнем окажется остаток).
2. Отсортировать фрагменты во внутренней памяти и записать результат во временный файл.
3. Создать  $Q + 1$  буфер размером  $B$  и приоритетную очередь, содержащую первый элемент из каждого фрагмента и его номер.
4. Пока очередь не пуста:
5.     Записать исключенный из очереди элемент в вектор.
6. Поместить следующий элемент из буфера записанного элемента в очередь, перезаполнив буфер, если он пуст и если во фрагменте больше элементов.

При наличии в блоке  $m$  элементов должно быть достаточно внутренней памяти для  $C$  элементов во время сортировки и  $mQ$  элементов во время слияния. Таким образом, параметры  $C = \sqrt{n}m$  и  $Q = n/C$  минимизируют использование внутренней памяти:

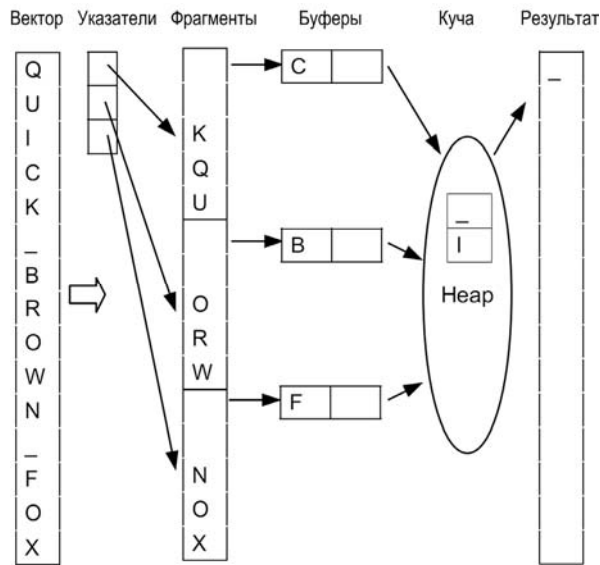


Рис. 13.4. Логика сортировки внешней памяти

```

friend void IOSort(EMVector& vector)
{
    { // удаление временного вектора перед удалением файла
        long long n = vector.getSize(), C = sqrt(n *
            vector.itemsPerBlock()), Q = n/C, lastQSize = n % C;
        File::remove("IOSortTempFile.igmdk"); // если уже существует
        EMVector temp("IOSortTempFile.igmdk"); // потенциально
        // значения могут быть разными
        // размер блока старого файла и значение BUFSIZ могут
        // измениться, но это не страшно
        typedef pair<POD, long long> HeapItem;
        Heap<HeapItem, PairFirstComparator<POD, long long> > merger;
        for(long long q = 0, i = 0; q < Q + 1; ++q)
        {
            long long m = q == Q ? lastQSize : C;
            if(m > 0)
            { // сортировка блоков
                Vector<POD> buffer;
                for(long long j = 0; j < m; ++j)buffer.append(vector[i++]);
                quickSort(buffer.getArray(), buffer.getSize());
                // помещение наименьшего элемента каждого
                // блока в кучу
                merger.insert(HeapItem(buffer[0], q));
                // а остальное – во временный вектор
                for(long long j = 1; j < m; ++j) temp.append(buffer[j]);
            }
        }
        Vector<Queue<POD> > buffers(Q + 1);
        Vector<long long> pointers(Q + 1, 0);
    }
}

```

```

for(long long i = 0; i < n; ++i)
{ // слияние. Помните, что временные блоки на 1 меньше
  long long q = merger.getMin().second;
  vector.set(merger.deleteMin().first, i);
  if(buffers[q].isEmpty()) // заполнение при необходимости
    while(pointers[q] < (q == Q ? lastQSize : C) - 1)
      buffers[q].push(temp[q * (C - 1) + pointers[q]++]);
  if(!buffers[q].isEmpty()) // проверка, что работа с блоком выполнена
    merger.insert(HeapItem(buffers[q].pop(), q));
}
}
File::remove("IOSortTempFile.igmdk");
}

```

Алгоритму требуется  $O(n/B)$  операций ввода/вывода, если на это достаточно внутренней памяти. Если нет, нужно объединить отсортированные фрагменты подобно сортировке слиянием, пока  $Q$  не станет достаточно маленьким, чтобы начать слияние кучи.

## 13.8. Векторные структуры данных

Множество структур данных можно реализовать с оптимальным вводом/выводом с помощью вектора:

- ◆ стек — обратите внимание, что при размере кеша, равном 1, последовательность push/pop на границе блока требует одного ввода/вывода на операцию. Размера 2 достаточно для оптимального амортизированного количества  $1/B$  на операцию;
- ◆ свободный список — один вектор содержит узлы, а другой — индексы возвращаемых узлов;
- ◆ очередь — используйте удвоение в циклической очереди, избегая создания временных данных после удвоения путем копирования первой половины во вторую.

Свободный список полезен в реализации многих других структур данных:

```

template<typename POD, typename SERIALIZER = CastSerializer<POD> >
class EMFreelist
{
  EMVector<POD, SERIALIZER> nodes;
  EMVector<long long, IntegralSerializer<long long> > returned;
  // запрет на копирование
  EMFreelist(EMFreelist const&);
  EMFreelist& operator=(EMFreelist const&);
public:
  EMFreelist(string const& filenameSuffix, int cacheSize = 2):
    nodes("Nodes" + filenameSuffix, cacheSize),
    returned("Returned" + filenameSuffix){}
  long long allocate(POD const& item = POD())
  {
    if(returned.getSize() > 0)
    { // повторное использование последнего освобожденного узла
      long long result = returned[returned.getSize() - 1];
      returned.removeLast();
    }
  }

```

```

        nodes.set(item, result);
        return result;
    }
    else
    {
        nodes.append(item);
        return nodes.getSize() - 1;
    }
}
void deallocate(long long i)
{ // не существует эффективного способа проверить,
  // освобождена ли память
  assert(i >= 0 && i < nodes.getSize());
  returned.append(i);
}
POD operator[](long long i)
{ // не существует эффективного способа проверить,
  // освобождена ли память
  assert(i >= 0 && i < nodes.getSize());
  return nodes[i];
}
void set(POD const& item, long long i)
{ // не существует эффективного способа проверить,
  // освобождена ли память
  assert(i >= 0 && i < nodes.getSize());
  nodes.set(item, i);
}
};

```

## 13.9. Дерево B+

Дерево B+ представляет собой динамическую отсортированную последовательность, в которой листья содержат значения, внутренние узлы — ключи. Во многих задачах при работе с базами данных ключи малы, а значения огромны, поэтому разделение повышает эффективность и позволяет связывать листья для перебора в цикле. Ключи некоторых узлов появляются в дереве несколько раз, но в сумме у внутренних узлов ключей меньше, чем листьев.

Внутренний узел имеет размер и  $M$  пар (ключ, указатель ввода/вывода).  $M$  — четное, больше или равно 4, и такое, что его размер в байтах  $\approx B$ . Узел размера  $k$  обобщает узел бинарного дерева и имеет  $k$  указателей и  $k - 1$  ключей, где указатель <sub>$i$</sub>  указывает на узел с ключами, которые меньше ключа <sub>$i$</sub> , а указатель <sub>$i + 1$</sub>  — на узел ключами, большими или равными ключу <sub>$i$</sub> . У пары  $k - 1$  есть только указатель. Поиск внутри внутреннего узла находит указатель на следующий узел в соответствии с этим определением.

В листе хранится отсортированный массив элементов размера  $L$  — такого, что размер узла в байтах  $\approx B$  больше или равен двум и является четным (рис. 13.5).

Для экономии места можно ввести соглашение о том, что  $-1$  является нулевым указателем, числа  $\geq 0$  являются внутренними указателями узлов, а числа  $< -1$  являются указателями листьев. Внутренние узлы поддерживают поиск дочернего элемента, соответст-

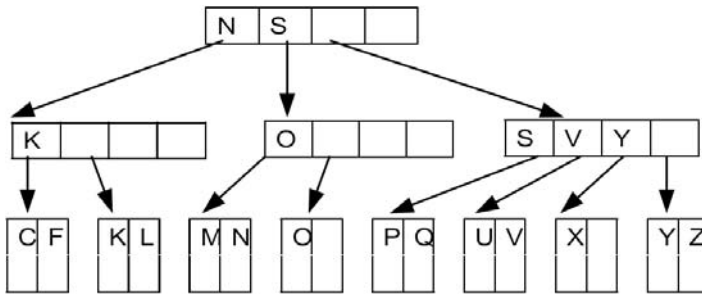


Рис. 13.5. Возможная древовидная структура дерева B+ для буквенных ключей и неуказанных значений

вующего ключу, и листьев, включающих преемника ключа. Используйте линейный поиск для обоих. Листья распределяются с использованием свободного списка, а внутренние узлы — вектора, потому что удаление не затрагивает последний. Для переносимости оба используют пользовательские сериализаторы, основанные на сериализаторах, предоставленных пользователем для ключей и значений:

```

template<typename KEY, typename VALUE, typename KEY_SERIALIZER =
    CastSerializer<KEY>, typename VALUE_SERIALIZER = CastSerializer<VALUE> >
class EMBPlusTree
{
    typedef KVPair<KEY, long long> Key;
    typedef KVPair<KEY, VALUE> Record;
    // выполнение ограничений на значения M и L,
    // но сначала от внутреннего узла
    enum{NULL_IO_POINTER = -1, NODE_SIZE_BYTES = 4, POINTER_SIZE = 8,
        KEY_SIZE = KEY_SERIALIZER::byteSize() + POINTER_SIZE, RECORD_SIZE =
        KEY_SERIALIZER::byteSize() + VALUE_SERIALIZER::byteSize(), B =
        BlockFile::targetBlockSize() - NODE_SIZE_BYTES, M = 2 * min<int>(2,
        B/2/KEY_SIZE), L = 2 * min<int>(1, (B - POINTER_SIZE)/2/RECORD_SIZE)
    };
    struct Node
    {
        int size;
        Key next[M];
        Node(): size(1) {next[0].value = NULL_IO_POINTER;}
        int findChild(KEY const& key)
        { // прошлое дочернее значение не превышает ключ
            int i = 0;
            while(i < size - 1 && key >= next[i].key) ++i;
            return i;
        }
    };
    struct Serializer
    {
        KEY_SERIALIZER ks;
        constexpr static int byteSize()
        {return NODE_SIZE_BYTES + int(M) * int(KEY_SIZE);}
        Node operator()(Vector<unsigned char> const& bytes)
        {
            assert(bytes.getSize() == byteSize()); // базовая проверка файла
        }
    };
};
  
```

```

    Node node;
    BitStream bs(bytes); // сперва декодируем размер
    node.size = ReinterpretDecode(bs.readBytes(NODE_SIZE_BYTES));
    for(int i = 0; i < M; ++i) // затем указатели
    {
        node.next[i].key =
            ks(bs.readBytes(KEY_SERIALIZER::byteSize()));
        node.next[i].value =
            ReinterpretDecode(bs.readBytes(POINTER_SIZE));
    }
    return node;
}
Vector<unsigned char> operator()(Node const& node)
{
    BitStream bs; // сперва декодируем размер
    bs.writeBytes(ReinterpretEncode(node.size, NODE_SIZE_BYTES));
    for(int i = 0; i < M; ++i) // затем указатели
    {
        bs.writeBytes(ks(node.next[i].key));
        bs.writeBytes(ReinterpretEncode(node.next[i].value,
            POINTER_SIZE));
    }
    return bs.bitset.getStorage();
}
};

struct Leaf
{
    int size;
    long long next;
    Record records[L];
    Leaf(): size(0), next(NULL_IO_POINTER) {}
    int inclusiveSuccessorRecord(KEY const& key)
    { // прошлая запись меньше ключа
        int i = 0;
        while(i < size && key > records[i].key) ++i;
        return i;
    }
}

struct Serializer
{
    KEY_SERIALIZER ks;
    VALUE_SERIALIZER vs;
    constexpr static int byteSize()
    {
        return int(NODE_SIZE_BYTES) + int(POINTER_SIZE) +
            int(L) * int(RECORD_SIZE);
    }
}

Leaf operator()(Vector<unsigned char> const& bytes)
{
    assert(bytes.getSize() == byteSize()); // базовая проверка файла
    Leaf leaf;

```



```

        BitStream bs(bytes); // сперва декодируем размер
        leaf.size = ReinterpretDecode(bs.readBytes(NODE_SIZE_BYTES));
        // затем следующий указатель
        leaf.next = ReinterpretDecode(bs.readBytes(POINTER_SIZE));
        for(int i = 0; i < L; ++i) // а потом записи
        {
            leaf.records[i].key =
                ks(bs.readBytes(KEY_SERIALIZER::byteSize()));
            leaf.records[i].value =
                vs(bs.readBytes(VALUE_SERIALIZER::byteSize()));
        }
        return leaf;
    }
    Vector<unsigned char> operator() (Leaf const& leaf)
    {
        BitStream bs; // сперва декодируем размер
        bs.writeBytes(ReinterpretEncode(leaf.size, NODE_SIZE_BYTES));
        // затем следующий указатель
        bs.writeBytes(ReinterpretEncode(leaf.next, POINTER_SIZE));
        for(int i = 0; i < L; ++i) // а потом записи
        {
            bs.writeBytes(ks(leaf.records[i].key));
            bs.writeBytes(vs(leaf.records[i].value));
        }
        return bs.bitset.getStorage();
    }
};
// индексы листов начинаются с -2
long long leafIndex(long long index){return -(index + 2);}
long long inverseLeafIndex(long long lIndex){return -lIndex - 2;}
long long root;
File header; // для корневого указателя доступ без кеширования
EMVector<Node, typename Node::Serializer> nodes;
EMFreelist<Leaf, typename Leaf::Serializer> leaves;
pair<long long, long long> findLeaf(KEY const& key)
{
    long long current = root, parent = NULL_IO_POINTER;
    while(current >= 0) // остановка по достижении листа или null
    {
        parent = current;
        Node node = nodes[current];
        current = node.next[node.findChild(key)].value;
    }
    return make_pair(current, parent);
}
EMBPlusTree(EMBPlusTree const&); // копирование не разрешено
EMBPlusTree& operator=(EMBPlusTree const&);
public:
    EMBPlusTree(string const& filenameSuffix): root(NULL_IO_POINTER),
        header(("Header" + filenameSuffix).c_str(), false),
        nodes("Keys" + filenameSuffix, 8), leaves("Records" + filenameSuffix)

```

```

{
    Vector<unsigned char> temp(POINTER_SIZE);
    if(header.getSize() > 0)
    {
        header.read(temp.getArray(), POINTER_SIZE);
        root = ReinterpretDecode(temp);
    }
    else header.append(temp.getArray(), POINTER_SIZE); // место для корня
}
~EMBPlusTree()
{ // записываем корень в заголовок
    header.setPosition(0);
    header.write(ReinterpretEncode(root, POINTER_SIZE).getArray(),
        POINTER_SIZE);
}
};

```

В операции поиска используется внутренний инвариант узла, чтобы найти лист, содержащий значение:

```

pair<long long, long long> findLeaf(KEY const& key)
{
    long long current = root, parent = NULL_IO_POINTER;
    while(current >= 0) // остановка по достижении листа или null
    {
        parent = current;
        Node node = nodes[current];
        current = node.next[node.findChild(key)].value;
    }
    return make_pair(current, parent);
}
ITEM find(KEY const& key, bool& status)
{
    status = true;
    long long current = findLeaf(key).first;
    if(current != NULL_IO_POINTER)
    {
        Leaf leaf = leaves[leafIndex(current)];
        int i = leaf.inclusiveSuccessorRecord(key);
        if(i < leaf.size && key == leaf.records[i].key)
            return leaf.records[i].value;
    }
    status = false;
    return VALUE();
}

```

Вставьте балансы, используя сплиты. Когда узел заполнен, а его родитель — нет:

1. Поместите правую половину ключей в новый узел.
2. Вставьте указатель на новый узел в родителя после указателя на узел.
3. Скопируйте средний ключ в родителя и переместите его в новый узел (рис. 13.6).

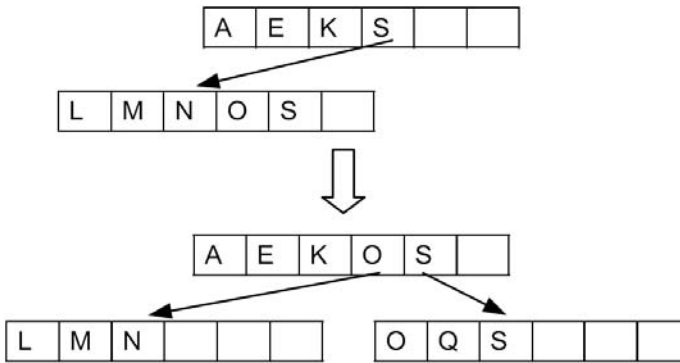


Рис. 13.6. Разделение узла дерева B+  
(напомним, что последняя ячейка ключа не используется)

```
void splitInternal(long long index, int child)
{
    Node parent = nodes[index];
    long long childIndex = parent.next[child].value;
    Node left = nodes[childIndex], right;
    // копируем ключ среднего элемента в родительский,
    // сдвигая его ключ
    for(int i = parent.size++; i > child; --i)
        parent.next[i] = parent.next[i - 1];
    parent.next[child].key = left.next[M/2 - 1].key;
    parent.next[child + 1].value = nodes.getSize();
    // перемещение элементов, начиная с середины, вправо
    right.size = M/2 + 1;
    for(int i = 0; i < right.size; ++i)
        right.next[i] = left.next[i + M/2 - 1];
    left.size = M/2;
    nodes.append(right); // запись узлов
    nodes.set(left, childIndex);
    nodes.set(parent, index);
}
```

Разделение листа следует той же логике с небольшими отличиями, потому что лист не содержит фиктивных ключей, поддерживает указатель на следующий лист, а элементы хранятся иначе:

```
void splitLeaf(long long index, int child)
{
    Node parent = nodes[index];
    long long childIndex = parent.next[child].value,
        newChildIndex = inverseLeafIndex(leaves.allocate());
    Leaf left = leaves[leafIndex(childIndex)], right;
    // копируем ключ среднего элемента в родительский,
    // сдвигая его ключ
    for(int i = parent.size++; i > child; --i)
        parent.next[i] = parent.next[i - 1];
    parent.next[child].key = left.records[L/2].key;
    parent.next[child + 1].value = newChildIndex;
```

```

// перемещение элементов, начиная с середины, вправо
left.size = right.size = L/2;
for(int i = 0; i < right.size; ++i)
    right.records[i] = left.records[i + L/2];
right.next = left.next;
left.next = newChildIndex;
leaves.set(right, leafIndex(newChildIndex)); // запись узлов
leaves.set(left, leafIndex(childIndex));
nodes.set(parent, index);
}

```

Чтобы выполнить вставку:

1. Создайте корень, если он отсутствует, и разделите его, если он заполнен.
2. Найдите нужный лист, вставляя ключ в каждый узел и разбивая полные узлы по пути вниз.
3. Вставьте элемент в соответствующий лист.

```

bool shouldSplit(long long node)
{
    return node < NULL_IO_POINTER ?
        leaves[leafIndex(node)].size == L : nodes[node].size == M;
}

void insert(KEY const& key, ITEM const& value)
{ // первый узел является корневым как лист
    if(root == NULL_IO_POINTER) root = inverseLeafIndex(leaves.allocate());
    else if(shouldSplit(root))
    { // разделяем корень, если нужно
        Node newRoot;
        newRoot.next[0].value = root;
        bool wasLeaf = root < NULL_IO_POINTER;
        root = nodes.getSize();
        nodes.append(newRoot);
        wasLeaf ? splitLeaf(root, 0) : splitInternal(root, 0);
    }
    long long index = root;
    while(index > NULL_IO_POINTER) // работа с внутренним узлом
    { // движение вниз, вставка и разделение
        Node node = nodes[index];
        int childI = node.findChild(key), child = node.next[childI].value;
        if(shouldSplit(child))
        { // разделение дочернего элемента, если нужно
            child < NULL_IO_POINTER ? splitLeaf(index, childI) :
                splitInternal(index, childI);
            if(key > nodes[index].next[childI].key) // переход к следующему потомку
                child = nodes[index].next[childI + 1].value;
        }
        index = child;
    }
    // вставка элемента в лист
    Leaf leaf = leaves[leafIndex(index)];
    int i = leaf.inclusiveSuccessorRecord(key);
}

```

```

    if(i < leaf.size && key == leaf.records[i].key)
        leaf.records[i].value = value;
    else
    {
        for(int j = leaf.size++; j > i; --j)
            leaf.records[j] = leaf.records[j - 1];
        leaf.records[i] = Record(key, value);
    }
    leaves.set(leaf, leafIndex(index));
}

```

В реализации с параллельным выполнением узел блокируется, когда алгоритм просматривает его, и разблокируется по завершении. Нисходящий характер всех операций позволяет это правильно реализовать. Операция удаления удаляет узел из листа без объединения. Такая простая стратегия используется в базах данных, потому что она эффективна для параллельного выполнения. Кроме того, реальные базы данных время от времени перестраиваются, особенно при добавлении новых полей данных, что обеспечивает слияние. С помощью дополнительной логики можно объединить наполовину заполненные узлы. Пустые листы также удаляются:

```

void remove(KEY const& key)
{
    pair<long long, long long> pointerAndParent = findLeaf(key);
    long long pointer = pointerAndParent.first;
    if(pointer != NULL_IO_POINTER)
    {
        Leaf leaf = leaves[leafIndex(pointer)];
        int i = leaf.inclusiveSuccessorRecord(key);
        if(i < leaf.size && key == leaf.records[i].key)
        {
            --leaf.size;
            for(int j = i; j < leaf.size; ++j)
                leaf.records[j] = leaf.records[j + 1];
        }
        if(leaf.size > 0) leaves.set(leaf, leafIndex(pointer));
        else
        { // Удаление листа
            leaves.deallocate(leafIndex(pointer));
            Node parent = nodes[pointerAndParent.second];
            parent.next[parent.findChild(key)].value = NULL_IO_POINTER;
            nodes.set(parent, pointerAndParent.second);
        }
    }
}

```

Дерево B+ является оптимальным по вводу/выводу при выполнении операций над динамическими отсортированными последовательностями. Без удалений каждый узел заполняется более чем наполовину, поэтому при наличии  $n$  элементов высота дерева равна  $O(\log_B(n))$ , что на практике  $< 4$ . В кеше LRU всегда будет храниться корень. Извлечение большой записи выполняется за оптимальное время  $O(\text{высота} + \text{чтение записи})$ .

При выполнении задач вроде добавления поля в базу данных необходимо перестраивать дерево. Вот алгоритм построения дерева  $B^+$  из отсортированного файла:

1. Выполнить сортировку.
2. Скопировать каждый  $B$ -й узел на следующий уровень.
3. Рекурсивно повторять алгоритм на  $B$  разделителях, пока все не уместится в один блок, который станет корнем.

Алгоритм выполняется за  $O(\text{сортировка}) < O(n \text{ вставок})$ .

## 13.10. Комментарии

Если значение  $\text{BUFSIZ} < B$ , лучше настроить размер буфера равным `fstream`. Для этого перед открытием файла можно использовать функцию `f.rdbuf()->pubsetbuf(array, size)`. Но здесь это не делается и далее рассматриваться не будет, поскольку:

- ◆ стандарт C++ не гарантирует, что эта функция сработает (кроме случаев, когда оба аргумента не равны 0, а это бесполезно);
- ◆ в моих тестах размер буфера, равный 4096 или даже 0, вообще не влиял на время чтения блоков. При чтении нескольких смежных блоков малого размера  $B$  время близко к оптимальному размеру буфера уже при 128 и не улучшается после 512;
- ◆ слишком большой размер буфера (в моем случае  $\geq 8192$ ) замедляет чтение блоков. Видимо, реализация пытается заполнить его даже для меньших  $B$ .

Вы можете эффективно реализовать связанный список, хотя реализация эта будет не столь полезна. Если блок размером  $IO$  вмещает  $k$  узлов, любые два последовательных блока должны содержать  $k/2$  элементов. Это приводит к замедлению сканирования всего в 2 раза. При вставке в блок помещается новый элемент. Если места в нем нет, один из граничных элементов вытаскивается в соседний блок. Если и в нем нет места, блок делится на два перед новой вставкой, что является наихудшим случаем ограничения емкости. Удаление выполняется просто, если не достигнут предел емкости, иначе блоки-нарушители объединяются.

У  $B$ -дерева (для простоты *дерево  $B^+$*  принято называть  $B$ -деревом) ключи и элементы не разделены, что негативно влияет на время выполнения.

Для операций с отображениями можно использовать несколько более быстрых хеш-таблиц — например, линейное зондирование (не учитывающее кеш) или линейное хеширование (подробности см. в работе [13.2]). Но хеш-таблицы не поддерживают упорядоченные операции и эффективное перестроение из отсортированных данных, к тому же у них имеются проблемные худшие случаи (см. [13.3]).

Многие алгоритмы трудно или невозможно сделать эффективными с точки зрения ввода/вывода. Например, у графовых алгоритмов скачки DFS занимают один ввод/вывод на вершину.

## 13.11. Советы по дополнительной подготовке

- ◆ Реализуйте сжатие вектора с заданным именем временного файла. Попробуйте выполнить то же самое с другими структурами данных.

- ◆ Реализуйте стек и очередь.
- ◆ Дополните сортировку векторов, добавив обработку случая нехватки памяти. Запустите тестирование на данных объемом не менее 16 Гбайт (или больше, если на диске хватит для этого места). Определите влияние параметра  $B$  на время выполнения.
- ◆ Реализуйте прямую итерацию для В-дерева.
- ◆ Реализуйте возможность эффективного создания В-дерева из вектора отсортированных записей.
- ◆ Исследуйте и реализуйте задачу увеличения количества узлов В-дерева.
- ◆ Реализуйте тесты надежности хранения для всех рассмотренных структур данных.

## 13.12. Список рекомендуемой литературы

- 13.1. Cormen T. H., Leiserson C. E., Rivest, R. L., & Stein C. (2009). Introduction to Algorithms. MIT Press.
- 13.2. Folk M., Zoellick B. and Riccardi G. (1998). File Structures: An Object-Oriented Approach with C++. Addison-Wesley.
- 13.3. Dementiev R. (2007). Algorithm Engineering for Large Data Sets. Springer.
- 13.4. Mehlhorn K., & Sanders P., Dietzfelbinger M., & Dementiev R. (2019). Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox. Springer.
- 13.5. Meyer U., Sanders P., & Sibeyn J. F. (2003). Algorithms for Memory Hierarchies. Springer.

# 14. Строковые алгоритмы

## 14.1. Введение

Если вы когда-либо задавались вопросом, как ваш текстовый редактор ищет шаблоны в текстовом файле, особенно, если шаблоны составлены из выражений, то ответ — строковые алгоритмы. Мы подробно рассмотрим многие полезные алгоритмы, но в целом это очень широкая тема, поэтому более детально вы можете ознакомиться с материалом в упоминаемой мною литературе.

*Строка* — это вектор символов из некоторого алфавита. Предполагается, что сравнение символов занимает время  $O(1)$ . *Алфавит* является:

- ♦ *индексированным*, если каждый символ соответствует числу, — например, в ASCII и Unicode, но не в UTF8;
- ♦ *ограниченным*, если количество возможных символов ограничено, — опять же, как в ASCII и Unicode, но не в UTF8;
- ♦ *упорядоченным*, если поддерживает осмысленные сравнения, — например, индексированные алфавиты поддерживают сравнения, а осмысленные — нет (за исключением подмножеств вроде букв или чисел в ASCII);
- ♦ *обобщенным*, если поддерживает сравнение на равенство.

Выделяют несколько часто используемых типов подстрок (также называемых *факторами*):

- ♦ *q-gram* — подстрока размером  $q$ ;
- ♦ *префикс* — начальная часть строки;
- ♦ *суффикс* — конечная часть строки.

## 14.2. Поиск по одному шаблону

Допустим, нам требуется найти все вхождения строки шаблона длиной  $m$  в большей текстовой строке длиной  $n$ . Для этого требуется хотя бы общий алфавит. Медленный алгоритм перебора просто проверяет совпадения в каждой позиции. Его время выполнения —  $O(mn)$ , а E[время выполнения] —  $O(n)$  (см. [14.4]). Из этого алгоритма наиболее интересна проверка появления подстроки в заданной точке текста. Предполагается, что в тексте достаточно символов, и все они один за другим сравниваются с образцом:

```
template<typename VECTOR, typename VECTOR2> bool matchesAt(int position,
    VECTOR2 text, VECTOR pattern, int patternSize)
{ // разрешение различных типов текста и шаблонов
    int i = 0;
```



```

while(i < patternSize && pattern[i] == text[i + position]) ++i;
return i == patternSize;
}

```

Нижние границы количества сравнения символов:

- ◆ худший случай  $O(n)$ ;
- ◆ ожидаемое время  $O\left(\frac{n \log_a(m)}{m}\right)$  при работе со случайным текстом и ограниченным алфавитом размера  $a$ .

*Алгоритм Хорспула* сдвигает шаблон слева направо. Например, если задан шаблон `apple` и текст `there_is_a_particularly_healthy_fruit_called_apple`, алгоритм грубой силы сравнивает `apple` с `there`, `here_`, `ere_i` и т. д. Вместо сдвига на одну позицию алгоритм Хорспула предварительно обрабатывает сам шаблон и вычисляет сдвиг любого символа, кроме последнего, т. е. максимально возможный сдвиг равен  $m - 1$ . Например, для слова `apple` предварительно вычисленные сдвиги имеют вид:

a	p	l	e	Любой другой
4	2	1	5	5

С учетом этой информации сдвиг определяется последним символом текущего фактора текста. Например, после сравнения `apple` с `there` выполняется сдвиг 5, чтобы выровнять шаблон с `_is_a`, потому что в слове `apple` нет других букв `e`. Затем на 4, чтобы выровнять с `a_par`, потому что `apple` начинается с `a`. Затем на 5, чтобы выровнять с `_appl`, и на 1, после чего будет найдено совпадение. Для случайного текста и шаблона  $E[\text{время выполнения}] = O\left(\frac{n}{\min(m, a)} + n_{\text{совпадений}}\right)$  (см. [14.12]).

У алгоритма *HashQ* оптимальное ожидаемое время выполнения (см. [14.10]) оказывается быстрее для небольших алфавитов или очень длинных шаблонов. Алгоритм применим для общих алфавитов (с соответствующей хеш-функцией, которую пользователь должен предоставить). В обоих вариантах время выполнения в худшем случае равно  $O(mn)$ . Алгоритм *HashQ* вычисляет сдвиги, используя  $q$ -граммы для  $q \geq 1$ , поэтому необходимо выполнение условия  $m \geq q$ . Оптимальное значение  $q = 2\log_d(m)$  (см. [14.13]). В алгоритме задействована хеш-таблица размера  $m/q$ , а в случае коллизий предпочтение отдается меньшим значениям сдвига. Чтобы найти величину сдвига во время сопоставления, найдите последнюю  $q$ -грамму текущего текстового фактора. Алгоритм предполагает, что значение  $h$  поставляется с компоновщиком.

Для маленьких алфавитов эффективнее задавать значение  $h$  явно. Например, при работе с ДНК удобно принять  $q = 4$ , где  $h$  = конкатенация всех битов. При  $q = 1$ , идентичности  $h$  и размере таблицы  $a$  алгоритм по скорости эквивалентен алгоритму Хорспула. Для  $q = 3$  в работе [14.10] предложено простое значение  $h$ , основанное на сложении и сдвиге, — в приведенном далее коде это реализовано без развертывания цикла для общего  $q$ :

```

struct LecroqHash // расчет при q = 1
{ // игнорируется параметр size - алгоритму сравнения будет достаточно размера таблицы
  LecroqHash(int dummy) {}
}

```

```

struct Builder
{
    unsigned char result;
    Builder(): result(0){}
    void add(unsigned char c){result = result << 1 + c;}
};

Builder makeBuilder(){return Builder();}
unsigned char operator() (Builder b){return b.result;}
};

template<typename VECTOR, typename HASHER = LecroqHash>
class HashQ
{
    enum{CHAR_ALPHABET_SIZE = 1 << numeric_limits<unsigned char>::digits};
    int patternSize, q;
    Vector<int> shifts; // размер для быстроты хеширования
                        // является степенью двойки
    VECTOR const &pattern;
    HASHER h;
    typedef typename HASHER::Builder B;
public:
    HashQ(VECTOR const& thePattern, int thePatternSize, int theQ = 1): q(theQ),
        pattern(thePattern), patternSize(thePatternSize), shifts(max<int>(
            CHAR_ALPHABET_SIZE, nextPowerOfTwo(ceiling(patternSize, q)))),
        h(shifts.getSize())
    { // предварительный расчет сдвигов
        assert(patternSize >= q);
        int temp = patternSize - q;
        for(int i = 0; i < shifts.getSize(); ++i) shifts[i] = temp + 1;
        for(int i = 0; i < temp; ++i)
        {
            B b(h.makeBuilder());
            for(int j = 0; j < q; ++j) b.add(pattern[i + j]);
            shifts[h(b)] = temp - i;
        }
    }
    // возврат позиции соответствия и следующей начальной позиции (-1, -1)
    template<typename VECTOR2> pair<int, int> findNext(VECTOR2 const& text,
        int textSize, int start = 0) // допускаются различные типы текста и шаблонов
    {
        while(start + patternSize <= textSize)
        {
            int result = start, hStart = start + patternSize - q;
            B b(h.makeBuilder());
            for(int j = 0; j < q; ++j) b.add(text[hStart + j]);
            start += shifts[h(b)];
            if(matchesAt(result, text, pattern, patternSize))
                return make_pair(result, start);
        }
        return make_pair(-1, -1);
    }
};

```

## 14.3. Поиск по нескольким шаблонам

Если есть  $k$  шаблонов разной длины, их можно просто сопоставлять поочередно один за другим, но алгоритм Ву — Манбера более эффективен. Он представляет собой обобщение алгоритма HashQ, при  $E[\text{время выполнения}] = O\left(\frac{n \log_a(mk)}{m}\right)$ ,  $q = \log_a(mk)$  и

размере таблицы  $\frac{mk}{q}$ , где  $m$  — это средняя длина (см. [14.12]). Можно сопоставлять

шаблоны длины  $< q$  по отдельности, алгоритмом Ву — Манбера с  $q = 1$  или один за другим. Алгоритм хеширует суффикс  $q$ -грам каждого шаблона, и для любого значения хеша создается список шаблонов, которые его хешируют. Хеш последней  $q$ -граммы текущего текстового фактора определяет сдвиг. Он равен минимально возможному сдвигу для любого шаблона и возможного совпадения, которые затем проверяются методом перебора:

```
template<typename VECTOR, typename HASHER = LecroqHash> class WuManber
{
    enum{CHAR_ALPHABET_SIZE = 1 << numeric_limits<unsigned char>::digits};
    int q, minPatternSize;
    Vector<pair<VECTOR, int> > const& patterns;
    Vector<int> shifts; // размер для быстроты хеширования
                        // является степенью двойки
    Vector<Vector<int> > candidates;
    HASHER h;
    typedef typename HASHER::Builder B;
public:
    WuManber(Vector<pair<VECTOR, int> > const& thePatterns, int theQ = 1,
              double avePatternSize = 1): q(theQ), patterns(thePatterns), shifts(
                max<int>(CHAR_ALPHABET_SIZE, nextPowerOfTwo(avePatternSize *
                    patterns.getSize()/q)), candidates(shifts.getSize()),
                h(shifts.getSize()), minPatternSize(numeric_limits<int>::max())
    { // предварительный расчет сдвигов
        for(int i = 0; i < patterns.getSize(); ++i)
            minPatternSize = min(patterns[i].second, minPatternSize);
        assert(patterns.getSize() > 0 && minPatternSize >= q);
        int temp = minPatternSize - q;
        for(int i = 0; i < shifts.getSize(); ++i) shifts[i] = temp + 1;
        for(int j = 0; j < patterns.getSize(); ++j)
            for(int i = 0; i < temp + 1; ++i)
            {
                B b(h.makeBuilder());
                for(int k = 0; k < q; ++k) b.add(patterns[j].first[i + k]);
                int hi = h(b);
                if(i == temp) candidates[hi].append(j);
                else shifts[hi] = min(temp - i, shifts[hi]);
            }
    } // возврат позиции соответствия, следующей начальной
    // позиции (-1, -1) и индексов шаблонов, в которых найдены
    // совпадения. Могут использоваться различные типы текста и шаблонов
```

```

template<typename VECTOR2> pair<Vector<int>, int> findNext(
    VECTOR2 const& text, int textSize, int start = 0)
{
    Vector<int> matches(patterns.getSize(), -1);
    while(start + minPatternSize <= textSize)
    {
        B b(h.makeBuilder());
        for(int j = 0; j < q; ++j)
            b.add(text[start + minPatternSize - q + j]);
        int hValue = h(b);
        bool foundAMatch = false;
        for(int i = 0; i < candidates[hValue].getSize(); ++i)
        {
            int j = candidates[hValue][i];
            if(start + patterns[j].second <= textSize && matchesAt(
                start, text, patterns[j].first, patterns[j].second))
            {
                foundAMatch = true;
                matches[j] = start;
            }
        }
        start += shifts[hValue];
        if(foundAMatch) return make_pair(matches, start);
    }
    return make_pair(matches, -1);
}
};

```

## 14.4. Регулярные выражения

*Регулярное выражение* — это шаблон текста, заданный правилами. Выражение образуется применением к предложениям операторов «\*» (левый операнд может встречаться ноль и более раз), «|» (левое или правое) и «&» (левое и правое) в указанном порядке. Оператор «&» явно не пишется. *Предложение* — это любое заключенное в скобки подвыражение, включая одиночные символы с неявной скобкой. Если операторы встречаются в качестве символов, используется escape-символ «/». Например, выражение `/(xxx/)/xxxx-xxx`, в котором  $0 \leq x \leq 9$ , соответствует номерам телефонов. Существуют и другие операторы — например, символы-джокеры классы символов. Чтобы узнать, содержит ли текст шаблон, соответствующий выражению, используйте синтаксис `*выражение*`.

*Алгоритм Глушкова* (см. [14.14, 14.12]) преобразует выражение в недетерминированный конечный автомат, где *эпсилон-переходы* не поглощают символы. Парсинг выражения из  $m$  символов создает граф, в котором вершины соответствуют конечным состояниям, а ребра — эпсилон-переходам, добавляемым:

- ◆ к следующему состоянию для символов «\*», «(» и «|»;
- ◆ между началом предложения и следующим состоянием, если есть символ «\*»;
- ◆ для символа «|» — от начала слева до начала справа и от начала справа до конца предложения (рис. 14.1).

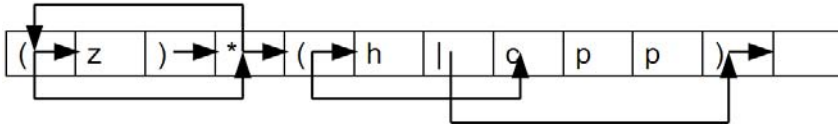


Рис 14.1. Регулярное выражение для «зудящего» файла C++ zzzzzz.cpp и его вариантов

В стеке хранятся открытые предложения, где символ «(» открывает предложение, «|» находится внутри предложения, а «)» закрывает его. Для простоты предположим, что символ «|» всегда находится внутри круглых скобок и escape-символов нет:

```
class RegularExpressionMatcher
{
    string re;
    int m;
    GraphAA<bool> g; // пробные данные
public:
    RegularExpressionMatcher(string const& theRe): re(theRe), m(re.length()),
        g(m + 1)
    {
        Stack<int> clauses;
        for(int i = 0; i < m; ++i)
        {
            int clauseStart = i;
            if(re[i] == '(' || re[i] == '|') clauses.push(i);
            else if(re[i] == ')')
            {
                int clauseOp = clauses.pop();
                if(re[clauseOp] == '|')
                {
                    clauseStart = clauses.pop();
                    g.addEdge(clauseStart, clauseOp + 1);
                    g.addEdge(clauseOp, i);
                }
                else clauseStart = clauseOp;
            }
            if(i < m - 1 && re[i + 1] == '*') // до следующей точки начала
                // от начала предложения
                g.addUndirectedEdge(clauseStart, i + 1);
            if(re[i] == '(' || re[i] == '*' || re[i] == ')')
                g.addEdge(i, i + 1); // от текущего состояния до следующего
        }
    }
};
```

*Активные состояния* изначально равны нулю. Алгоритм DFS вычисляет все возможные следующие состояния, используя эпсилон-переходы. Выполнение переходов чтения из текущих активных состояний дает следующий набор активных состояний. Совпадение считается найденным, если конечное состояние  $m$  остается активным после чтения всех символов:

```
Vector<int> findActiveStates(Vector<int> const& sources)
{
    Vector<bool> visited(g.nVertices(), false);
```

```

DefaultDFSAction a;
for(int i = 0; i < sources.getSize(); ++i) if(!visited[sources[i]])
{
    visited[sources[i]] = true;
    DFSComponent(g, sources[i], visited, a);
}
Vector<int> activeStates;
for(int i = 0; i < visited.getSize(); ++i)
    if(visited[i]) activeStates.append(i);
return activeStates;
}

bool matches(string const& text)
{
    Vector<int> activeStates = findActiveStates(Vector<int>(1, 0));
    for(int i = 0; i < text.length() && activeStates.getSize() > 0; ++i)
    { // для продолжения активных состояний должно быть
      // не менее одного
      Vector<int> stillActive;
      for(int j = 0; j < activeStates.getSize(); ++j)
          if(activeStates[j] < m && re[activeStates[j]] == text[i])
              stillActive.append(activeStates[j] + 1);
      activeStates = findActiveStates(stillActive);
    }
    for(int j = 0; j < activeStates.getSize(); ++j)
        if(activeStates[j] == m) return true;
    return false;
}

```

Время выполнения равняется  $O(nm)$  из-за необходимости обновлять активные состояния после каждого символа текста.

## 14.5. Расширенные шаблоны

*Расширенный шаблон* — это регулярное выражение, в котором операторы «\*» и «|» используются на односимвольных операндах, что позволяет ускорить поиск. Самым простым решением является расширение алгоритма поиска по одному шаблону *сдвиг-и* (shift-and) (см. [14.12]).

Если есть шаблон длины  $m$ , алгоритм поддерживает строку битов активного состояния  $s$  из  $m$  битов, такую что бит  $i$  установлен, если последние  $i + 1$  символов текста совпадают с префиксом шаблона длиной  $i + 1$  при объединении с символом чтения. При выполнении сравнения установленный бит остается в сдвинутой позиции, если его соответствующий символ совпал с шаблоном. Мы предполагаем, что  $m$  меньше размера слова  $w$ . Если нет, можно использовать набор битов. Поиск всех совпадений занимает время  $O(nm/w)$ :

1. Для любого возможного символа:
2.     Задать битовую строку  $c$  размером  $m$ , заполненную нулями.
3. Для любого символа шаблона в позиции  $i$ :
4.     Установить бит  $i$  в строке  $c$ .

5. Изначально  $s = 0$ .
6. Пока  $j < n$ :
7.     Прочитать символ текста  $t[j]$ .
8.     Установить  $s$  равным  $(s \ll 1 \mid 1) \& c[t[j]]$ .
9.     Сообщить о совпадении в позиции  $j - (m - 1)$ , если установлен бит  $m - 1$ .

Например, для шаблона `apple` и символа «р»  $c = 01100$ . Возможны расширения алгоритма:

- ◆ *символы-джокеры*, соответствующие любому символу, и операция « $\mid$ » из нескольких символов. В этом случае устанавливаются соответствующие биты каждого совпадающего символа;
- ◆ повторяющиеся символы, соответствующие регулярному выражению  $xx^*$ . Для них нужно запоминать совпадающие позиции в  $s$  после сдвига. Если информация о позициях хранится в битовой строке  $R$ , используйте выражение  $((s \ll 1 \mid 1) \mid (s \& R)) \& c$  (см. на веб-сайте онлайн-дополнения к работе [14.12]);
- ◆ некоторые необязательные символы могут быть опущены. Пусть в битовой строке  $O$  определены позиции таких символов. После чтения любого символа установите бит в  $s$ , а если следующие  $k$  позиций необязательны, установите следующие  $k$  битов.

На этапе предварительной обработки вычисляются значения  $L$  и  $P$  — последняя и предыдущая позиции блоков последовательных установленных битов в строке  $O$ . Например, если  $O = 01010110$  и  $s = 00100001$ ,  $s$  становится равным  $01100111$ ,  $F = 01010100$  и  $P = 00101001$ . Расчет  $P$  предполагает отсутствие в бите 0 необязательных символов, но это не проблема, поскольку необязательные символы в начале и в конце шаблона можно опускать. После обновления символа задаем  $s$  равным  $s \mid O \& ((s \mid L) \wedge \sim((s \mid L) - P))$  (пояснение приведено в работе [14.12]).

Выражение « $x^*$ » делает символ  $x$  необязательным и повторяемым. Реализация предполагает, что вызывающая сторона задает маску `charPos`, разрешая выполнение операции « $\mid$ ». Интерфейс несколько отличается от интерфейса алгоритмов точного сопоставления, потому что последние работают только с ограниченным алфавитом и сохраняют другие состояния, помимо текущей позиции:

```
class ShiftAndExtended
```

```
{ // обработка символов-джокеров для простоты пропущена
    enum{ALPHABET_SIZE = 1 << numeric_limits<unsigned char>::digits};
    unsigned char *pattern;
    int patternSize, position; // patternSize задается перед масками
    unsigned long long charPos[ALPHABET_SIZE], O, P, L, R, state;
    unsigned long long makeMask(Vector<int> const& positions)const
    {
        unsigned long long mask = 0;
        for(int i = 0; i < positions.getSize(); ++i)
        {
            assert(positions[i] >= 0 && positions[i] < patternSize);
            Bits::set(mask, positions[i], true);
        }
        return mask;
    }
}
```

```

public:
    ShiftAndExtended(unsigned char* thePattern, int thePatternSize,
        Vector<int> const& repeatedPositions = Vector<int>(),
        Vector<int> const& optionalPositions = Vector<int>()): position(0),
        state(0), patternSize(thePatternSize), pattern(thePattern),
        R(makeMask(repeatedPositions)), O(makeMask(optionalPositions))
    { // предварительное вычисление битовых строк
        assert(patternSize <= numeric_limits<unsigned long long>::digits &&
            !Bits::get(0, 0)); // символы в позиции 0 не могут быть необязательными
        for(int i = 0; i < ALPHABET_SIZE; ++i) charPos[i] = 0;
        for(int i = 0; i < patternSize; ++i)
            Bits::set(charPos[pattern[i]], i, true);
        // и маски для необязательных символов
        unsigned long long sides = 0 ^ (0 >> 1);
        P = (0 >> 1) & sides;
        L = 0 & sides;
    }
    int findNext(unsigned char* text, int textSize)
    {
        while(position < textSize)
        { // выполняем обновление для повторяющихся символов
            state = (((state << 1) | 1) | (state & R)) &
                charPos[text[position++]];
            // а затем обновление необязательных символов
            unsigned long long sL = state | L;
            state |= 0 & (sL ^ ~(sL - P));
            if(Bits::get(state, patternSize - 1)) return position - patternSize;
        }
        return -1;
    }
};

```

Время выполнения такое же, как и у обычного алгоритма *сдвиг-и*.

## 14.6. Алгоритмы поиска расстояния между строками

Основные расстояния между строками, основанные на количестве операций редактирования, применяемых к строкам, позволяют:

- ◆ *Hamming* — замену;
- ◆ *Indel* — вставку/удаление;
- ◆ *Lewenstein* — вставку/удаление/замену;
- ◆ *Transpose* — вставку/удаление/замену/перенос.

Первые три — это метрики с известной стоимостью любой операции. Обычно вычисляют *разницу между двумя строками*, в которых параметр расстояния отступа используется для создания сценария редактирования, который изменяет строку *a* на строку *b* с помощью последовательности команд:



- ♦ вставить  $i$  — вставка  $b[i]$  на место  $i$  в строку  $a$ ;
- ♦ удалить  $i$  — удалить  $a[i]$ .

Чтобы применить этот сценарий к строке  $a$ , не зная  $b$ , нужно сохранить вставленный  $b[i]$  в вектор  $c$  так, чтобы  $j$ -я вставка в позицию  $i$  вставляла  $c[j]$  в позицию  $i$  вектора  $a$ . Такой алгоритм полезен, например, для программного обеспечения контроля версий, где текстовые файлы представляют собой строки символов общего алфавита, а все прошлые версии файла хранятся как различия между последовательными версиями. Структура памяти последовательности редактирования, хранимой в векторе, представлена на рис. 14.2.

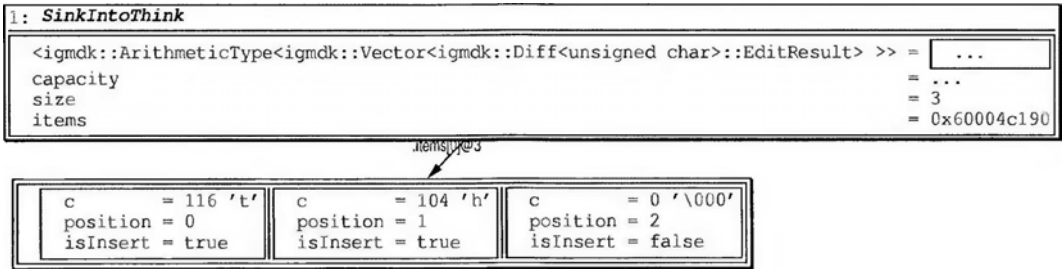


Рис. 14.2. Структура памяти последовательности редактирования, хранимой в векторе

```

template<typename CHAR> class Diff
{
public:
    struct EditResult
    {
        CHAR c;
        int position;
        bool isInsert;
    };
};

```

Вы можете определять расстояния рекурсивно по любой метрике  $d$  в удобной для вычислений форме. Пусть  $p(s, j)$  — первые  $j$  символов строки  $s$ . Тогда:

$$d(a, b) = \begin{cases} n, & \text{если } a \text{ пусто} \\ d(p(a, m-1), p(b, n-1)), & \text{если } a[m-1] = b[n-1] \\ \min(d(a, p(b, n-1)), d(p(a, m-1), b)) + 1 & \end{cases}$$

Алгоритм динамического программирования использует  $O(nm)$  времени и пространства, но оно может отличаться для других расстояний и неударных затрат.

Алгоритм *ВМММ* более эффективен. Пусть  $x$  соответствует  $p(b, x+1)$ , а  $y$  —  $p(a, y+1)$ . Тогда  $x$  и  $y$  определяют набор диагоналей с постоянным  $x - y$ . Диагональ  $m+1$  равна  $(-1, -1)$ , а диагональ  $n+1$  равна  $(n-1, m-1)$ . Если требуется выполнить  $p$  удалений:

- ♦ алгоритм динамического программирования позволяет достичь базового случая  $(-1, -1)$  из  $(n-1, m-1)$  путем вычисления внутри диагоналей от  $m+1-p$  до  $n+1-p$ ;
- ♦ сценарий выполняет  $p$  удалений и  $n-m$  вставок;
- ♦  $d = n - m + p$ .

Решение заключается в проходе кратчайшим путем из  $(-1, -1)$  в  $(n - 1, m - 1)$ , где из  $(x, y)$  можно перейти в:

- ◆  $(x + 1, y + 1)$  — бесплатно, если  $a[y] = b[x]$ ;
- ◆  $(x + 1, y)$  — ценой вставки;
- ◆  $(x, y + 1)$  — ценой удаления.

Схема превращения слова *sink* в *think* представлена на рис. 14.3.

5	6	7	8	9	10	11			
4			-1	0	1	2	3	4	
3				t	h	i	n	k	
2	-1		0	1	2	3	4	5	
1	0	s	1	2	3	4	5	6	
0	1	i	2	3	4	3	4	5	
	2	n	3	4	5	4	3	4	
	3	k	4	5	6	5	4	3	

Рис. 14.3. Превращение слова *sink* в *think*.

Работа с диагональными индексами от 0 до 11, кратчайший путь редактирования увеличивается до 3

Логика кратчайшего пути позволяет для любой диагонали получать текущий  $x$  и последовательность правок, которые приводят к  $(x, y)$ , вычисляя  $y = x - (d - 1 - m)$ . Значение  $x$  хранится как *граничный* вектор, равный  $-2$  (это удобный базовый случай). Поскольку значение  $p$  неизвестно, оно изначально равно 0 и многократно увеличивается, расширяя все диагонали, пока не будет достигнута точка  $(n - 1, m - 1)$ . Требуется  $\leq n + m + 3$  диагоналей. Расширение диагонали границы сначала требует время и пространства  $O(np)$  и ожидаемое  $O(n + dp)$ , где  $d$  — расстояние редактирования (см. [14.15]). Реализация предполагает, что  $m \leq n$ :

```
struct Edit
{
    Edit* prev; // используется только для промежуточных задач,
                // а не конечного результата
    int position;
    bool isInsert;
};

static Vector<EditResult> DiffInternal(Vector<CHAR> const& a,
    Vector<CHAR> const& b, CHAR const& nullC)
{
    int M = a.getSize(), N = b.getSize(), size = M + N + 3,
        mainDiagonal = N + 1;
    assert(M <= N); // строка a должна быть короче, чем b
    Vector<int> frontierX(size, -2);
    Vector<Edit*> edits(size, 0);
```

```

Freelist<Edit> f;
for(int p = 0; frontierX[mainDiagonal] < N - 1; ++p)
{ // от нижнего левого угла к главной диагонали
    for(int d = M + 1 - p; d < mainDiagonal; ++d)
        extendDiagonal(d, frontierX, edits, a, b, f);
    // от правого верхнего угла к главной диагонали
    for(int d = mainDiagonal + p; d >= mainDiagonal; --d)
        extendDiagonal(d, frontierX, edits, a, b, f);
} // вычисленный путь извлекается в обратном порядке
Vector<EditResult> result;
for(Edit* link = edits[mainDiagonal]; link; link = link->prev)
{
    EditResult er = {nullC, link->position, link->isInsert};
    result.append(er);
} // правильный порядок
result.reverse();
return result;
}

```

Алгоритм расширяет диагональ, разрешая еще одну операцию редактирования. Поскольку удаление граничной точки на диагонали  $i + 1$  или вставка из точки на диагонали  $i - 1$  ведут к самой дальней точке на диагонали  $i$ , можно вычислить  $x = \max(\text{граница}[i - 1] + 1, \text{граница}[i + 1])$ . Это стоит одной операции редактирования, за исключением базового случая при переходе в  $(-1, -1)$ . Затем можно расширить вычисленную самую дальнюю точку без затрат на редактирование, увеличив  $x$ , пока  $a[y + 1] = b[x + 1]$ :

```

static void extendDiagonal(int d, Vector<int>& frontierX, Vector<Edit*>&
    edits, Vector<CHAR> const& a, Vector<CHAR> const& b, Freelist<Edit>& f)
{ // выбор следующего лучшего варианта редактирования
    int x = max(frontierX[d - 1] + 1, frontierX[d + 1]),
        y = x - (d - 1 - a.getSize());
    if(x != -1 || y != -1)
    { // применение редактирования, если не имеет места базовый случай
        bool isInsert = x != frontierX[d + 1];
        edits[d] = new(f.allocate())Edit();
        edits[d]->isInsert = isInsert;
        edits[d]->prev = edits[d + (isInsert ? -1 : 1)];
        edits[d]->position = isInsert ? x : y;
    } // перемещение по диагонали как можно дальше
    while(y + 1 < a.getSize() && x + 1 < b.getSize() &&
        a[y + 1] == b[x + 1])
    {
        ++y;
        ++x;
    }
    frontierX[d] = x;
}

```

Превращение  $a$  в  $b$  выполняется так же, как  $b$  в  $a$ , но операции вставки и удаления меняются местами, поэтому условие  $m \leq n$  алгоритму не мешает. Число удалений увеличивается на (количество вставок – количество удалений), потому что каждое редактирование сдвигает оставшиеся позиции символов. Количество вставок остается преж-

ним, потому что каждый символ  $b$ , вставленный в  $a$ , находится в одной и той же позиции в обеих строках:

```
static Vector<EditResult> diff(Vector<CHAR> const& a, Vector<CHAR> const& b,
    CHAR const& nullC = CHAR()) // нулевой символ, который дальше будет удален
{ // редактирование нужно для превращения a в b - позиции
  // рассчитываются относительно b
  bool exchange = a.getSize() > b.getSize();
  Vector<EditResult> result = exchange ? DiffInternal(b, a, nullC) :
    DiffInternal(a, b, nullC);
  for(int i = 0, netInserted = 0; i < result.getSize(); ++i)
  { // выполнение обмена и при необходимости установка символов
    // и изменение позиций удаления
    if(exchange) result[i].isInsert = !result[i].isInsert;
    if(result[i].isInsert)
    {
      ++netInserted;
      result[i].c = b[result[i].position];
    }
    else result[i].position += netInserted--;
  }
  return result;
}
```

В сценарии редактирования «из  $a$  в  $b$  и обратно в  $a$ » строка  $b$  строится посимвольно. Построение выполняется итеративно, пока для операции редактирования не берутся символы из  $a$ :

```
static Vector<CHAR> applyDiff(Vector<CHAR> const& a,
    Vector<EditResult> const& script)
{
  Vector<CHAR> b;
  int nextA = 0;
  for(int i = 0; i < script.getSize(); ++i)
  { // отбор символов из a до следующей позиции
    while(b.getSize() < script[i].position)
    { // базовая проверка ввода, чтобы строка не закончилась
      // до проверки следующей позиции
      assert(nextA < a.getSize());
      b.append(a[nextA++]);
    }
    if(script[i].isInsert) b.append(script[i].c);
    else ++nextA; // пропуск одного символа при удалении
  } // остальное берется из a
  while(nextA < a.getSize()) b.append(a[nextA++]);
  return b;
}
```

## 14.7. Обратный индекс

Обратный индекс сопоставляет термин со списком всех документов, которые его содержат, и поддерживает логический запрос `содержит(термин)`. Например, предметный

указатель в конце книги — это инвертированный индекс, т. е. отсортированный вектор слов. Чтобы создать указатель, запустите синтаксический анализатор для каждого документа и вставьте его идентификатор в список всех терминов, найденных синтаксическим анализатором. У слов должны быть разделители — например, пробелы и знаки препинания.

Вы можете определять множество типов запросов, в частности булевы формулы, такие как `contains("inverted") & contains("index")`. Это позволит создать соответствующие списки идентификаторов и вычислить их пересечение. Например, пересечение двух отсортированных последовательностей размеров  $n$  и  $m$  может быть оптимально выполнено относительно простым алгоритмом за время  $O(n \lg(m/n))$ , где  $n$  — размер более короткой последовательности (см. [14.2]). При работе с большими наборами данных может использоваться много серверов, каждый из которых предназначен для определенного диапазона терминов. Для вычислений пересечения двух последовательностей нужно случайным образом выбрать один сервер и отправить его данные на другой сервер, который вычислит и вернет пересечение со своими собственными данными. Могут поддерживаться и другие операции — такие как объединение и исключение.

Если мы работаем с большим индексом, хранящимся на диске, — такие обычно используют поисковые системы, карта реализуется в виде В-дерева. Чтобы сжать списки, их нужно отсортировать и сохранить первое число в байтовом коде (см. главу 15. *Сжатие*), а любое другое число хранится как байтовое отличие от предыдущего числа. Чтобы сделать индекс параллельным, можно использовать хеш-таблицу и выделить отдельный сервер для каждого диапазона выходных данных хеш-функции.

## 14.8. Суффиксный индекс

*Массив суффиксов* строки размером  $n$  представляет собой массив позиций суффиксов строки, отсортированных в лексикографическом порядке (рис. 14.4). Алгоритм быстрой сортировки с несколькими ключами вычисляет его за время  $O(n^2 \lg(n))$  и ожидаемое время  $O(n \lg(n))$ .

Асимптотически оптимальный алгоритм для общего алфавита основан на *лемме удвоения* (см. [14.4]). Пусть  $r(i, k)$  — отсчитываемый с нуля ранг суффикса в позиции  $i$  в спи-

mississippi	10	i
	7	ippi
	4	issippi
	1	ississippi
	0	mississippi
	9	pi
	8	ppi
	6	sippi
	3	sissippi
	5	ssippi
	2	ssissippi

Рис. 14.4. Массив суффиксов для слова mississippi с соответствующими суффиксами



```

        ranks2[sa[i]] = r;
        if(r == n - 1) return sa; // ранги уже уникальны
    }
    ranks.swapWith(ranks2); // более эффективно, чем присвоение
}
}
return sa;
}

```

Массив *lcp* хранит *lcp* соседних суффиксов в массиве.  $lcp[i] = lcp(sa[i-1], sa[i])$  — при  $i = 0$  значение не определено. В ходе вычисления используется *переставленный массив lcp*, определенный как  $PLCP[sa[i]] = lcp[i]$ , а также временный массив, равный  $pred[i] = \begin{cases} sa[size-1], & \text{если } i = 0 \\ sa[i-1] \end{cases}$ . Значение  $pred[0]$  используется для удобства.  $PLCP[i] = lcp(i, pred[i])$ . Для  $i > 0$   $PLCP[i] \geq PLCP[i-1] - 1$  (в работе [14.9]  $pred[0]$  обозначено как  $\phi[0]$  и не определено). Итак, нужно вычислить  $PLCP$  путем линейного сканирования  $pred$  и для  $i > 0$ , не глядя на первые  $PLCP[i-1] - 1$  символов. Поскольку  $\sum$  различий между последовательными значениями  $PLCP$  равна  $O(n)$ , время выполнения тоже равно  $O(n)$ :

```

template<typename ITEM> Vector<int> LCPArray(ITEM* text, int size, int* sa)
{
    Vector<int> pred(size, 0), PLCP(size, 0);
    for(int i = 0; i < size; ++i) pred[sa[i]] = sa[(i ? i : size) - 1];
    for(int i = 0, p = 0; i < size; ++i)
    {
        while(text[i + p] == text[pred[i] + p]) ++p;
        PLCP[i] = p;
        p = max(p - 1, 0);
    } // значение pred становится массивом LCP, значению 0
    // соответствует lcp(sa[0], sa[n-1])
    for(int i = 0; i < size; ++i) pred[i] = PLCP[sa[i]];
    return pred;
}

```

Массивы суффиксов и *lcp* образуют суффиксный указатель, который эффективен для многих задач сопоставления с образцом. Например, поиск всех вхождений шаблона занимает  $O(m \lg(n))$  времени с двумя бинарными поисками путем вычисления интервала, в котором все суффиксы являются префиксами шаблона. Количество совпадений =  $\text{right} - \text{left} + 1$ . Структура памяти суффиксного указателя для слова mississippi приведена на рис. 14.5.

```

template<typename ITEM> struct SuffixIndex
{
    Vector<ITEM> const& text;
    Vector<int> sa, lcpa;
    SuffixIndex(Vector<ITEM> const& theText): text(theText),
        sa(suffixArray<SARank>(text.getArray(), text.getSize()),
        lcpa(LCPArray(text.getArray(), text.getSize(), sa.getArray())) {}
    bool isKLess(ITEM const* a, int aSize, ITEM const* b, int bSize, int k)
    const

```

```
{
    for(int i = 0; i < min(aSize, bSize); ++i)
        if(a[i] > b[i]) return false;
    return max(aSize, bSize) < k;
}
pair<int, int> interval(ITEM* pattern, int size)
{
    int left = 0, right = sa.getSize();
    while(left < right)
    {
        int i = (left + right)/2;
        if(isKLess(&text[sa[i]], sa.getSize() - sa[i], pattern, size,
            size)) left = i + 1;
        else right = i - 1;
    }
    int left2 = left - 1, right2 = sa.getSize() - 1;
    while(left2 < right2)
    {
        int i = (left2 + right2)/2;
        if(isKLess(pattern, size, &text[sa[i]], sa.getSize() - sa[i],
            size)) right2 = i - 1;
        else left2 = i + 1;
    }
    return make_pair(left, right2);
}
};
```

Вы можете найти самую длинную повторяющуюся подстроку за время  $O(n)$ , проверив, какие два суффикса имеют наибольший lcp.

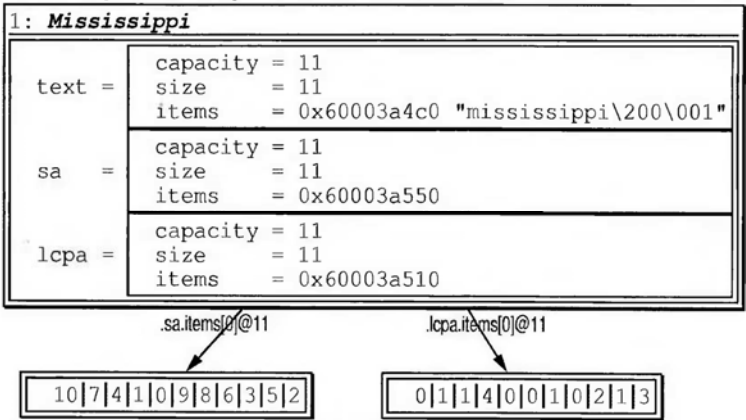


Рис. 14.5. Структура памяти суффиксного указателя для слова mississippi



## 14.9. Синтаксическое дерево

В *синтаксическом дереве* каждый узел имеет значение, которое является константой или функцией своих потомков. Значение корня = значению его функционального выражения. Функция может представлять любую связь дочерних узлов своего узла.

Синтаксическое дерево обычно является результатом преобразования текстовых команд. Например,  $5 + 4$  соответствует дереву, представленному массивом  $[(+, 1, 2), (5, -1, -1), (4, -1, -1)]$  (последние два числа в кортеже — это указатели на дочерние элементы). *Лексический анализ* разделяет последовательности символов на символы и использует для создания дерева определенные правила грамматики — такие как порядок старшинства.

Если нужно получить читаемую и редактируемую спецификацию чего-либо, существуют форматы вроде XML и JSON, у которых более чем достаточно функций и имеется хорошая поддержка API на многих языках.

## 14.10. Введение в краткие структуры данных

Структура данных является *краткой*, если она занимает пространство  $O(\text{теоретико-информационное минимальное количество битов, необходимое для различения объекта})$ . Это количество обычно равно  $\lg(|\text{множество всех возможных объектов}|)$ . Например, для представления перестановки требуется  $\lg(n!) \approx n \lg n$  битов, хотя такое представление с использованием набора битов мало что дает и замедляет работу. Внешние потери фрагментации из-за управления памятью в этой модели не учитываются, хотя и важны.

Набор битов — это естественная базовая структура данных, и чтобы она была максимально полезной, ее нужно расширить *структурой данных с выбором ранга*, которая поддерживает следующие операции:

- ◆  $\text{Rank}(i)$  — количество единиц перед позицией  $i$ ;
- ◆  $\text{Select}(i)$  — позиция первой единицы;
- ◆  $\text{Rank}_0(i)$  и  $\text{select}_0(i)$  — эти операции для нуля.

Операции ранга и выбора являются инверсиями друг друга и обладают следующими свойствами ( $\text{rank}_0$  и  $\text{select}_0$  симметричны):

- ◆  $\text{Rank}_0(i) = i - \text{rank}(i)$ ;
- ◆  $\text{Rank}(\text{select}(i)) = i$ ;
- ◆  $\text{Select}(\text{rank}(i)) = (\text{установлен ли } i\text{-й бит? } i : i - 1)$ ;
- ◆ Если  $j = \text{select}(i)$ , то  $\text{rank}(j) = i$ , а  $\text{rank}_0(j) = j - i$ .

Следует отметить, что в литературе эти операции иногда определяются для индексации массива, в которых индексация ведется с 1, при этом операция  $\text{rank}(i)$  также учитывает текущий бит. Было предложено много сложных реализаций для набора битов размером  $n$ , позволяющих получить использование пространства  $O(n)$  и время  $O(1)$ . В нашем случае предпочтительнее простейшая и достаточно эффективная реализация: в любом слове в наборе битов хранится кумулятивный счетчик битов. Для объемов более  $2^{32}$  и в наборе битов, и в счетчиках используются 64-битные слова, дающие  $O(n)$  дополнительного пространства:

- ◆  $O(1)$   $\text{rank}(i)$  — используется функция  $\text{counts}[i/64 - 1]$  и биты слова набора битов  $i/64$ ;
- ◆  $O(\lg(n))$   $\text{select}(i)$  — используется бинарный поиск по счетчикам битов и битам найденного слова битового набора

Алгоритм побайтово просматривает биты наборов битов, используя предварительно вычисленную таблицу счетчиков для любого числа байтов. Реализация  $\text{select}_0(i)$  подобна реализации  $\text{select}(i)$ , только для 0:

```
int rank64(unsigned long long x, int i)
{ // i-й бит не включен
    assert(0 <= i && i < numeric_limits<unsigned long long>::digits);
    // установка битов на позициях >= i равными 0, установка popCount
    return popCountWord(x & ((1ull << i) - 1));
    // чтобы включить i-й бит, используется индекс i+1 вместо i
    // переполнение не страшно и нормально, потому что результат будет равен -1
}

int select64(unsigned long long x, int i, bool is0 = false)
{
    assert(0 <= i && i < numeric_limits<unsigned long long>::digits);
    static PopCount8 p8;
    int result = 0, byteBits = numeric_limits<unsigned char>::digits;
    // используется переменная popCount, чтобы обратиться к байту с нужной позицией
    while(x)
    {
        int temp = p8(x & 0xff);
        if(is0) temp = byteBits - temp;
        if(i - temp < 0) break;
        result += byteBits;
        x >>= byteBits;
        i -= temp;
    } // сканирование битов в найденном байте для поиска желаемой битовой позиции
    for(int j = 0; j <= byteBits; ++j)
    {
        bool temp = x & 1 << j;
        if(is0) temp = !temp;
        if(temp && i-- == 0) return result;
        ++result;
    }
    return -1;
}

class RankSelect
{
    enum{B = numeric_limits<unsigned long long>::digits};
    Bitset<unsigned long long> bitset;
    // кумулятивный подсчет всех битов в слове
    Vector<unsigned long long> counts;
    long long counts0(long long i){return (i + 1) * B - counts[i];}
    long long getCount(long long i, bool is0)
        {return is0 ? (i + 1) * B - counts[i] : counts[i];}
public:
    RankSelect(unsigned long long initialSize = 0): bitset(initialSize){}
    Bitset<unsigned long long>& getBitset(){return bitset;}
```

```

void finalize()
{
    counts = bitset.getStorage();
    for(long long i = 0; i < bitset.wordSize(); ++i) counts[i] =
        popCountWord(bitset.getStorage()[i]) + (i == 0 ? 0 : counts[i-1]);
}
long long rank(long long i)
{
    assert(0 <= i && i < bitset.getSize());
    long long index = i / B;
    return (index > 0 ? counts[index - 1] : 0) +
        rank64(bitset.getStorage()[index], i % B);
}
long long rank0(long long i){return i - rank(i);}
long long select(long long i, bool is0 = false)
{
    assert(0 <= i && i < bitset.getSize());
    long long left = 0, right = bitset.wordSize() - 1;
    while(left < right)
    {
        long long middle = (left + right)/2;
        if(getCount(middle, is0) <= i) left = middle + 1;
        else right = middle - 1;
    }
    long long result = select64(bitset.getStorage()[left],
        i - (left == 0 ? 0 : getCount(left - 1, is0)), is0);
    if(result != -1) result += left * B;
    return result;
}
long long select0(long long i){return select(i, true);}
long long prev(long long i){return select(rank(i) - 1);}
long long prev0(long long i){return select0(rank0(i) - 1);}
long long next(long long i){return select(rank(i));}
long long next0(long long i){return select0(rank0(i));}
};

```

Поскольку функции ранга и выбора статичны, структуры, опирающиеся на них, не могут быть создаваться динамическими. Можно убрать лишнее пространство за счет двухуровневой структуры, но делать так не рекомендуется:

- ◆ уровень 1 содержит 64-битные кумулятивные суммы 256-битных блоков;
- ◆ уровень 2 содержит 8-битные подсчеты рангов для 64- или 32-битных блоков. Затраты памяти становятся равными  $0,5n$  или  $0,33n$  вместо  $n$ .

Простой вариант реализации — дерево. Избыточность пространства присутствует в левом, правом и родительском указателях. Для полного бинарного дерева они не нужны, потому что дерево может быть представлено в виде кучи, в которой избыточности вообще нет. Для общего бинарного дерева нужен массив элементов и небольшая

дополнительная структура для навигации. Существует  $\left(\frac{2n+1}{n}\right)/(2n+1)$  деревьев с  $n$  узлами, и тогда  $2n$  битов более чем достаточно для представления такой структуры вместо  $96n$  битов (32 бита на указатель). Пространство для данных не изменяется.

В работе [14.8] предложены приведенные далее простые реализации. Общее дерево называется *LOUDS*, оно просто и эффективно и должно строиться путем обхода по уровням, что неудобно, т. к. для этого нужна очередь или итеративное углубление. Это может быть полезно, например, для представления некоторых больших файлов XML:

```
class BinaryTree
{ // не распространяется на d-арные деревья, в отличие от кучи,
  // которой требуется не менее log(d)n битов
  RankSelect rs;
  int convert(int i){return rs.getBitset()[i] ? rs.rank(i) : -1;}
public:
  void addNodeInLevelOrder(bool isNotExternalDummyLeaf)
  {rs.getBitset().append(isNotExternalDummyLeaf);}
  void finalize(){rs.finalize();}
  int parent(int i){return (rs.select(i)+1)/2-1;}
  int leftChild(int i){return convert(2 * rs.rank(i) + 1);}
  int rightChild(int i){return convert(2 * rs.rank(i) + 2);}
};

class OrdinalTree
{
  RankSelect rs;
  int convert(int i){return rs.getBitset()[i] ? rs.rank(i) : -1;}
public:
  OrdinalTree(){rs.getBitset().append(1);rs.getBitset().append(0);}
  void addNodeInLevelOrder(int nChildren)
  {
    for(int i = 0; i < nChildren; ++i) rs.getBitset().append(1);
    rs.getBitset().append(0);
  }
  void finalize(){rs.finalize();}
  int parent(int i){return rs.select(i) - i - 1;}
  int firstChild(int i){return convert(rs.select0(i) + 1);}
  int nextChild(int i) {return convert(rs.select(i) + 1);}
};
```

Еще одно простое приложение — это поиск в тексте по сжатому шаблону. Предположим, что текст сжат, а шаблона нет. Вы можете вернуть текст в исходное состояние и выполнять в нем поиск и другие операции, но иногда требуется работать непосредственно со сжатым текстом, что в случае с очень большими текстами часто имеет смысл. Если текст сжат с использованием кода Хаффмана или другого метода преобразования символов в символы, вы можете так же сжать шаблон и выполнять сопоставление с помощью любого метода, который не пропускает символы (поскольку у нас нет произвольного доступа).

Довольно простая схема, обеспечивающая произвольный доступ, выглядит следующим образом (см. [14.6]):

1. Ранжировать все символы по частоте их появления, и пусть кодовые слова для любого символа будут представлены рангами:  $0 \rightarrow 0$ ,  $1 \rightarrow 1$ ,  $2 \rightarrow 00$ ,  $3 \rightarrow 01$  и т. д. — т. е. двоичными кодами определенной длины.
2. Этот код не содержит префиксов, поэтому нужно создать набор битов того же размера, что и закодированный текст, с одним набором для каждого бита начала символа.

3. Поместить структуру данных с выбором ранга в набор битов, чтобы разрешить произвольный доступ к структуре.

Получаемая структура поддерживает все операции, поддерживаемые исходной строкой, но с замедлением  $O(\lg(n))$  (из-за реализации операции выбора). Можно написать код лучше, но тогда надо предварительно обработать текст, использовать функцию перемещения на передний план и сохранить его отображение символов. Это часто используемый метод организации произвольного доступа к любому символьному коду, и он эффективен, поскольку код не обязательно должен не иметь префиксы. Специализированные алгоритмы сопоставления позволяют использовать более эффективные методы сжатия.

Основная область исследований строковых алгоритмов — это сжатые индексы, такие как *FM-индекс* (см. [14.1]). Массив суффиксов требует много места для вычисления и представления, поэтому желательно максимально сжать его, используя битовые алгоритмы, но это замедлит алгоритм. Идея состоит в том, что в задачах вроде секвенирования ДНК сокращение памяти с постоянным коэффициентом может привести к неразрешимости задачи. Но при работе с 64-битной архитектурой и растущим объемом памяти это не проблема. В работе [14.11] можно подробнее почитать об этих и других кратких структурах данных, включая более подходящий вариант дерева со сбалансированными скобками.

## 14.11. Примечания по реализации

Несмотря на доступность специализированных учебников, оказалось, что реализации потребовали основательного исследования исходной литературы. Выбирать алгоритмы тоже оказалось непросто.

- ◆ Базовый поиск — объединение алгоритмов Horspool и HashQ — это оригинальная идея, причем алгоритм Horspool мало где используется в качестве основного алгоритма поиска.
- ◆ Построение массива суффиксов — у меня были трудности даже с запуском реализаций из нескольких алгоритмов, написанных до меня, и в итоге я остановился на представленном здесь алгоритме, т. к. он прост и поддерживает работу с общими алфавитами.
- ◆ Для построения массива LCP необходимо знакомиться с исходной литературой.
- ◆ Для вычисления разницы в строках, как ни странно, пришлось читать исходную литературу, потому что представленный мною алгоритм уже давно не новый, но ни в одном из учебников, которые вышли позже, он не рассматривался.
- ◆ Я решил не реализовывать какие-либо сжатые алгоритмы сопоставления, поскольку это слишком узкоспециализированная область, которая до сих пор исследуется.

Большинство тестов появилось спустя годы после первых реализаций алгоритмов, и в результате было обнаружено несколько ошибок, часть которых до сих пор не исправлена.

## 14.12. Комментарии

Исследователями было предложено множество алгоритмов поиска в строках (см. [14.5]). Алгоритмы Horspool и HashQ относятся к числу самых простых и практически всегда конкурентоспособны.

Предложено также множество алгоритмов вычисления различий в строках, но, как правило, для этого достаточно алгоритма WMMM с небольшими правками. В утилите Unix diff по какой-то причине используется более старая, менее эффективная его версия (см. [14.7]). Существует связанная с этой задача поиска самой длинной общей подпоследовательности, алгоритмы которой рассмотрены в работе [14.3].

В задаче приблизительного сравнения с образцом, которое используется, например, в средствах проверки орфографии, один из простых подходов при наличии словаря известных слов заключается в том, чтобы сначала попытаться найти слово как есть. Если это не удастся, нужно сгенерировать все слова с расстоянием 1 и попытаться найти их. Затем повторить с расстоянием 2 и т. д., пока поиск не станет слишком дорогим. Сама генерация для слова из  $k$  символов реализуется просто:

1. Создать  $k$  слов, удалив каждый символ.
2. Создать  $(k + 1) | A |$  слов путем вставки каждого символа алфавита в каждую возможную позицию.
3. Создать  $(k - 1) (| A | - 1)$  слов, заменив каждый символ другим.
4. Создать  $(k - 1)$  слов, поменяв местами каждую соседнюю пару символов.

Для выполнения  $i$ -й итерации требуется время  $O((k|A|)i)$ , даже если удалить дубликаты, поэтому этот метод имеет смысл использовать, если расстояние не превышает 3.

Потенциально лучший метод — это размещение словаря в дереве VP (см. главу 18. *Вычислительная геометрия*), у которого расстояние редактирования является расстоянием между узлами. После этого можно использовать алгоритм поиска ближайшего соседа.

## 14.13. Советы по дополнительной подготовке

- ♦ Для алгоритма HashQ реализуйте логику, которая выбирает значение  $q$  автоматически и использует значение  $h$ , предложенное в работе [14.10]. Сравните его с некоторыми более удачными значениями  $h$ , описанными в главе 9. *Хеширование*.
- ♦ Настройте HashQ на сравнение последовательностей ДНК, сохраните вектор  $k$ -битных слов для  $k = 2$  (см. главу 12. *Разные алгоритмы и методы*).
- ♦ Реализуйте обработку символов-джокеров для расширенного алгоритма сдвиг-и. Подсказка: нужно вычислить отдельные маски для любого джокера и использовать их вместо характерных для символа масок.
- ♦ Реализуйте поиск подстроки на основе суффиксного массива текста. Сравните производительность этого алгоритма с алгоритмом Хорспула.
- ♦ Исследуйте и внедрите другие функции суффиксного указателя. Многие из них реализуются в виде более эффективного по времени, но менее эффективного по памяти *суффиксного дерева*.

## 14.14. Список рекомендуемой литературы

- 14.1. Adjero D., Bell T. C., & Mukherjee A. (2008). The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching. Springer.
- 14.2. Baeza-Yates R., & Salinger A. (2010). Fast intersection algorithms for sorted sequences. In Algorithms and Applications (pp. 45–61). Springer, Berlin, Heidelberg.
- 14.3. Bergroth L., Hakonen H., & Raita T. (2000). A survey of longest common subsequence algorithms. In Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000 (pp. 39–48). IEEE.
- 14.4. Crochemore M., Hancart C., & Lecroq T. (2007). Algorithms on Strings. Cambridge University Press.
- 14.5. Faro S., & Lecroq T. (2010). The exact string matching problem: a comprehensive experimental evaluation. arXiv preprint arXiv:1012.2547.
- 14.6. Fredriksson K., & Nikitin F. (2009). Simple random access compression. Fundamenta Informaticae, 92(1-2), 63-81.
- 14.7. Hunt J. J., Vo K. P., & Tichy W. F. (1998). Delta algorithms: an empirical analysis. ACM Transactions on Software Engineering and Methodology (TOSEM), 7(2), 192–214.
- 14.8. Jacobson G. (1989). Space-efficient static trees and graphs. In 30th Annual Symposium on Foundations of Computer Science (pp. 549–554). IEEE Computer Society.
- 14.9. Kärkkäinen J., Manzini G., & Puglisi S. J. (2009). Permuted longest-common-prefix array. In Combinatorial Pattern Matching (pp. 181–192). Springer.
- 14.10. Lecroq T. (2007). Fast exact string matching algorithms. Information Processing Letters, 102(6), 229–235.
- 14.11. Navarro G. (2016). Compact Data Structures: A Practical Approach. Cambridge University Press.
- 14.12. Navarro G., & Raffinot M. (2002). Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences. Cambridge University Press. Online extras at <https://www.dcc.uchile.cl/~gnavarro/FPMbook/extras.html>. Accessed August 19, 2018.
- 14.13. Baeza-Yates R., & Ribeiro-Neto B. (2011). Modern Information Retrieval. Addison-Wesley.
- 14.14. Sedgewick R., & Wayne K. (2011). Algorithms. Addison-Wesley.
- 14.15. Wu S., Manber U., Myers G., & Miller W. (1990). An  $O(NP)$  sequence comparison algorithm. Information Processing Letters, 35(6), 317–323.

# 15. Сжатие

## 15.1. Введение

Сжатие данных в современных вычислениях применяется повсеместно, и в этой главе мы рассмотрим все касающиеся его основные алгоритмы, кроме LZ77 (того, что используется в GZIP).

## 15.2. Основные ограничения

Сжатие позволяет сэкономить место на диске путем преобразования битовой строки, в которой закодирован, например, большой файл изображения, в более короткую битовую строку, которую можно потом преобразовать обратно в оригинал. Никакой алгоритм не способен сжать любую битовую строку, иначе его многократное применение позволило бы уменьшить ее размер до 0. Сжатие и распаковка являются функциями взаимно однозначного соответствия, и существует больше последовательностей длины  $n + 1$ , чем  $n$ , поэтому каждый алгоритм, укорачивающий одни битовые строки, удлиняет другие. Это может быть, например, всего один лишний бит. Но на практике сжатие значительно экономит место.

Некоторые битовые строки можно сжимать сильнее других. Например, строку, состоящую из миллиона нулей, можно представить в виде «миллиона нулей». Кратчайшее описание битовой строки — это ее *сложность по Колмогорову*, вычисление которой неразрешимо, потому что кажущаяся несжимаемой длинная битовая строка может быть сгенерирована умным генератором псевдослучайных чисел и представлена в виде начального значения и кода генератора. Обычно сложность по Колмогорову определяется по отношению к *универсальной машине* (например, к реальным программам на C++ без помощи библиотек), но это не имеет значения, поскольку сама ее идея имеет ценность только в теории.

## 15.3. Энтропия

Для последовательности символов из алфавита размера  $k$ , которые встречаются с вероятностью, заданной распределением  $X$ , практической мерой сжимаемости является *энтропия первого порядка*  $H(X) = -\sum \text{Pr}(\text{символ}) \lg(\text{Pr}(\text{символ}))$ . Если каждый символ является *суперсимволом* из  $k$  символов (например, формат `short` обычно состоит из двух `char`), говорят об *энтропии первого порядка* для  $k \rightarrow \infty$ . Длина оптимального кодового слова для символа  $\approx -\lg(\text{Pr}(\text{символ}))$ .

Создание супералфавита не влияет на энтропию, если символы независимы. Так, использование байтовых и битовых символов дает одинаковый результат, если биты



независимы. В противном случае создание суперсимволов снижает энтропию, т. е.  $H(X) \geq H(Xk)/k$ . Например, энтропия текста ASCII меньше, если вместо битов используются буквы, потому что большинство 8-битных значений никогда не встречаются, а буква *e* встречается чаще, чем *z*.

Энтропия — это минимальное  $E[\text{число битов на символ}]$ , необходимое для передачи последовательности суперсимволов, и это предел сжимаемости при отсутствии другой информации. Размер представления последовательности — это ее энтропия  $\geq$  максимальному коэффициенту сжатия. Например, энтропия английского языка  $>$  одного бита на букву, а некоторые алгоритмы сжимают различные тексты ASCII до размера  $< 2$  битов на букву (см. [15.4]).

## 15.4. Битовый поток

Битовый поток может читать и записывать биты и представляет собой обертку для набора битов. Его интерфейс является абстракцией и может быть реализован как внешняя память или поток байтов. Для эффективности также реализуется чтение и запись слов. При работе с файлами нужно закладывать определенную переносимость:

- ◆ скомпилированный код будет работать на любой машине;
- ◆ файл, сжатый на одной архитектуре, будет распаковываться на другой.

Эти требования находятся в противоречии, потому что не у каждой машины есть 8-битный тип слова для представления байта, хотя почти всех современных машин это не касается. Кроме того, символ, который в C++ считается байтом, не обязательно должен занимать 8 битов (по-прежнему `sizeof(char) == 1`), хотя почти на всех современных машинах это так. Наконец, потоки в C++ специализированы только для символов, а не для `uint_8`. Таким образом, практическое решение здесь состоит в том, чтобы использовать тип байта `unsigned char` и не предполагать, что он имеет размер 8 битов. Это нарушает переносимость файлов между архитектурами с разным размером бита символов, но использование `uint_8` приведет к ошибке компилятора из-за отсутствия типа.

Как правило, двоичные файлы непереносимы, переносятся только текстовые файлы. Когда требуется переносимость двоичных файлов, например для работы с сетевыми протоколами, укажите расположение логических битов, подходящее для общих архитектур. В необычных архитектурах такие файлы должны читаться по-своему, используя предоставленные специальные API. Например, можно использовать переносимый поток символов ASCII, но упаковывать только первые 8 битов в символе, независимо от того, сколько там их на самом деле. Здесь это не рассматривается, потому что предоставленный вектор символов может быть преобразован в специальное представление путем дальнейшей обработки. То же самое касается задач кодирования с исправлением ошибок в криптографии (обсуждается в последующих главах):

```
struct Stream
{
    unsigned long long position;
    Stream(): position(0) {}
};

struct BitStream : public Stream
{
    Bitset<unsigned char> bitset; // беззнаковый char для портативности
    enum{B = numeric_limits<unsigned char>::digits};
```

```

BitStream() {}
BitStream(Bitset<unsigned char> const& aBitset): bitset(aBitset) {}
BitStream(Vector<unsigned char> const& vector): bitset(vector) {}
void writeBit(bool value){bitset.append(value);}
bool readBit()
{
    assert(bitsLeft());
    return bitset[position++];
}
void writeByte(unsigned char byte){writeValue(byte, B);}
void writeBytes(Vector<unsigned char> const& bytes)
    {for(int i = 0; i < bytes.getSize(); ++i) writeByte(bytes[i]);}
unsigned char readByte(){return readValue(B);}
Vector<unsigned char> readBytes(int n)
{
    assert(n <= bytesLeft());
    Vector<unsigned char> result(n);
    for(int i = 0; i < n; ++i) result[i] = readByte();
    return result;
}
void writeValue(unsigned long long value, int bits)
    {bitset.appendValue(value, bits);}
unsigned long long readValue(int bits)
{
    assert(bits <= bitsLeft());
    position += bits;
    return bitset.getValue(position - bits, bits);
}
unsigned long long bitsLeft()const{return bitset.getSize() - position;}
unsigned long long bytesLeft()const{return bitsLeft()/B;}
};

```

Поток выполняет роль строителя набора битов. Чтобы преобразовать набор битов в вектор байтов, закодируйте количество битов в последнем байте хранилища набора битов в последний байт вектора:

```

Vector<unsigned char> ExtraBitsCompress(Bitset<unsigned char> const& bitset)
{
    assert(bitset.getSize() > 0); // в противном случае не имеет смысла
    Vector<unsigned char> result = bitset.getStorage();
    result.append(bitset.garbageBits());
    return result;
}
Bitset<unsigned char> ExtraBitsUncompress(Vector<unsigned char> byteArray)
{
    assert(byteArray.getSize() > 1 && byteArray.lastItem() < BitStream::B);
    int garbageBits = byteArray.lastItem();
    byteArray.removeLast();
    Bitset<unsigned char> result(byteArray);
    while(garbageBits--) result.removeLast();
    return result;
}

```

Оба алгоритма выполняются за время  $O(n)$ .

## 15.5. Коды

Код задает битовую последовательность для любого символа в некотором алфавите. Предположим, что алфавит индексированный. В методах сжатия выполняется *моделирование*, т. е. преобразование входной последовательности в сжатую или подходящую для сжатия. Затем выполняется *кодирование*, т. е. входные данные преобразуются в битовую строку.

*Префиксные коды*, где ни одно кодовое слово не является префиксом другого, декодируются однозначно. Они удовлетворяют *неравенству Крафта*:

$$\sum 2^{-\text{длина(кодовое слово)}} \leq 1,$$

которое строже, чем граница энтропии, потому что нельзя использовать биты. Кодовое слово состоит из значения и указания длины. В качестве указания может выступать:

- ◆ *завершающий символ* — такой как определенная последовательность битов;
- ◆ *закодированная длина*, предшествующая значению;
- ◆ *соглашение о фиксированной длине* — например, всегда использовать 32 бита для определенного значения.

Для устойчивости к битовым ошибкам соблюдается неравенство «закодированная длина < завершающий символ < соглашение о длине». Входной алфавит обычно представляет собой код фиксированной длины, такой как ASCII.

## 15.6. Статические коды

В статическом коде одна и та же битовая последовательность всегда соответствует одному и тому же символу, независимо от контекста. Например, в двоичном коде обычно используется 8 битов для формата `char` и 32 для формата `int`. Это почти идеально, когда знаешь, сколько битов использовать. Например, так можно закодировать ДНК:  $(A, C, G, T) \rightarrow (00, 01, 10, 11)$ .

*Унарный код* — это соответствующее значению количество единиц и завершающий 0, например  $5 \rightarrow 111110$ . Для  $b$ -битного числа требуется  $O(2^b)$  битов, поэтому код полезен только как строительный блок. Другие эффективные коды находятся между бинарными и унарными и требуют  $O(b)$  времени и места кодового слова:

```
void UnaryEncode(int n, BitStream& result)
{
    while(n-->0) result.writeBit(true);
    result.writeBit(false);
}

int UnaryDecode(BitStream& code)
{
    int n = 0;
    while(code.readBit()) ++n;
    return n;
}
```

*Гамма-код* выражает число в виде  $2^x + y$  для максимально возможного  $x$  и записывает  $x$  в унарном виде, а  $y$  — в двоичном, используя  $x$  битов, например,  $5 \rightarrow 11001$ , потому что

$5 = 22 + 1$ . Для представления числа  $n$  требуется приблизительно  $2\lg(n)$  битов, что делает код асимптотически оптимальным, поскольку  $\lg(n)$  — минимум. Не существует кода для нуля, но 1 кодируется как 1:

```
void GammaEncode(unsigned long long n, BitStream& result)
{
    assert(n > 0);
    int N = lgFloor(n);
    UnaryEncode(N, result);
    if(N > 0) result.writeValue(n - twoPower(N), N);
}

unsigned long long GammaDecode(BitStream& code)
{
    int N = UnaryDecode(code);
    return twoPower(N) + (N > 0 ? code.readValue(N) : 0);
}
```

Статические коды используются в стратегиях поиска неизвестного положительного числа с помощью сравнений. Например, унарный код подобен линейному поиску, а гамма — экспоненциальному.

Любое число  $n$  можно уникально представить как сумму некоторых чисел Фибоначчи  $\leq n$ . При этом в сумме не появляются два последовательных числа Фибоначчи (иначе они были бы заменены их суммой). *Код Фибоначчи*:

1. Найдите числа, образующие сумму.
2. Для любого числа, от меньшего к большему, поставьте 1, если оно включено, и 0, если нет.
3. Добавьте завершающую единицу.

Две последовательные единицы означают завершение сигнала. Например, 7 кодируется как 01011, потому что  $7 = 0 \times 1 + 1 \times 2 + 0 \times 3 + 1 \times 5$ . Поскольку  $i$ -е число Фибоначчи  $\approx G^i$ , где  $G$  — золотое сечение, а  $i + 1$  бит соответствует  $n$ ,  $G^i \leq n < G^{i+1}$  и  $\lg(n)/\lg(G) + 1 \approx 1,44\lg(n) + 1$  бит представляет  $n$ :

```
void advanceFib(unsigned long long& f1, unsigned long long& f2)
{
    unsigned long long temp = f2;
    f2 += f1;
    f1 = temp;
}

void FibonacciEncode(unsigned long long n, BitStream& result)
{
    assert(n > 0);
    // поиск наибольшего числа Фибоначчи f1 <= n
    unsigned long long f1 = 1, f2 = 2;
    while(f2 <= n) advanceFib(f1, f2);
    // пометка чисел в порядке убывания
    Bitset<unsigned char> reverse;
    while(f2 > 1)
    {
        reverse.append(n >= f1);
```

```

    if(n >= f1) n -= f1;
    unsigned long long temp = f1;
    f1 = f2 - f1;
    f2 = temp;
} // изменение порядка на возрастающий и добавление завершающего символа
reverse.reverse();
result.bitset.appendBitset(reverse);
result.writeBit(true);
}

unsigned long long FibonacciDecode(BitStream& code)
{
    unsigned long long n = 0, f1 = 1, f2 = 2;
    for(bool prevBit = false;; advanceFib(f1, f2))
    { // добавление следующего числа Фибоначчи
        bool bit = code.readBit();
        if(bit)
        {
            if(prevBit) break;
            n += f1;
        }
        prevBit = bit;
    }
    return n;
}

```

*Байтовый код* представляет числа по основанию 128 (при условии, что тип `char` занимает 8 битов) в виде последовательности байтов с прямым порядком (т. е. самый маленький байт идет первым), где старший бит сигнализирует о последнем символе, а значение 0 является завершающим. Например, байт-код числа  $1282 \rightarrow 10000000|10000000|00000001$ . Этот код работает быстро за счет того, что не требует битовых манипуляций, эффективен, задействует  $\lceil \log_{128}(n) \rceil$  байтов =  $\Theta(1,41 \lg(n))$  битов и часто применяется для сжатия структур данных. В UTF-8 байтовый код используется для символов Unicode:

```

void byteEncode(unsigned long long n, BitStream& result)
{
    enum{M05 = 1 << (numeric_limits<unsigned char>::digits - 1)};
    do
    {
        unsigned char r = n % M05;
        n /= M05;
        if(n) r += M05;
        result.writeByte(r);
    }while(n);
}

unsigned long long byteDecode(BitStream& stream)
{
    unsigned long long n = 0, base = 1;
    enum{M05 = 1 << (numeric_limits<unsigned char>::digits - 1)};
    for(;; base *= M05)
    {
        unsigned char code = stream.readByte(), value = code % M05;
        n += base * value;
    }
}

```

```
        if(value == code) break;
    }
    return n;
}
```

Вы можете преобразовать число в последовательность байтов без сжатия:

```
Vector<unsigned char> ReinterpretEncode(unsigned long long n, int size)
```

```
{
    assert(size > 0);
    enum{M = 1 << numeric_limits<unsigned char>::digits};
    Vector<unsigned char> result;
    while(size-- > 0)
    {
        result.append(n % M);
        n /= M;
    }
    return result;
}
```

```
unsigned long long ReinterpretDecode(Vector<unsigned char> const& code)
```

```
{
    assert(code.getSize() > 0);
    unsigned long long n = 0, base = 1;
    enum{M = 1 << numeric_limits<unsigned char>::digits};
    for(int i = 0; i < code.getSize(); ++i)
    {
        n += base * code[i];
        base *= M;
    }
    return n;
}
```

Лучший код для значений в диапазоне [1, 7] — гамма-код, для диапазона [4, 33] ∪ [128, 1596] — Фибоначчи, а для диапазона [21, 127] ∪ [987, ∞) — байт-код. На практике используется байт-код, если не известно заранее, что работать предстоит только с очень маленькими числами (табл. 15.1).

Таблица 15.1

Значение	Унарный	Гамма-код	Код Фибоначчи	Байт-код
0	0	Н/Д	Н/Д	'00000000
1	10	1	'11	'00000001
2	110	100	011	'00000010
3	1110	101	'0011	'00000011
4	11110	11000	'1011	'00000100
5	111110	11001	'00011	'00000101
6	1111110	11010	'10011	0'0000110
7	11111110	11011	'01011	'00000111
8	111111110	1110000	'000011	'00001000

Таблица 15.1 (окончание)

Значение	Унарный	Гамма-код	Код Фибоначчи	Байт-код
16	Слишком длинный	111100000	'0010011	'00010000
32		11111000000	'00101011	'00100000
64		1111110000000	'1000100011	'01000000
128		111111100000000	'100010001011	'1000000000000001
256		'11111111000000000	'0100001000011	'1000000000000010
512		'1111111110000000000	'10101001010011	'1000000000000100
1024		'111111111100000000000	'0010000100000011	'1000000000000100

### 15.7. Коды Хаффмана

Коды Хаффмана оптимальны для любого заданного эмпирического распределения наблюдаемых символов. Алгоритм собирает вероятности появления символов при первом проходе по данным и использует их для вычисления кодов на втором проходе. Двоичные деревья, где символы расположены в листьях, позволяют моделировать префиксные коды. Движение по дереву влево добавляет к коду 0, а вправо — 1, в результате на листьях образуются кодовые слова.

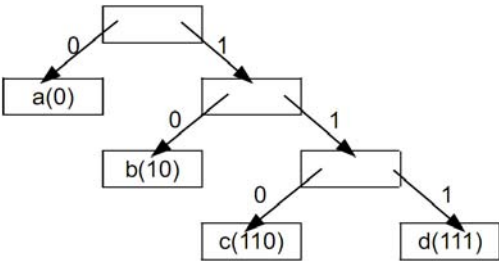


Рис. 15.1. Возможная кодировка Хаффмана для алфавита {a, b, c, d}

Предположим, что алфавит *a, b, c, d* можно закодировать кодом (00, 01, 10, 11) или (0, 10, 110, 111), как показано на рис. 15.1. Если буквы встречаются с вероятностью  $\frac{1}{4}$ , то  $E[\text{длина кодового слова}] = 2$  для первого набора и 2,25 для второго, но для вероятностей  $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}$  и  $\frac{1}{8}$  у второго набора  $E[\text{длина}] = 1,75$ . Нам нужно найти для этого распределения дерево, которое минимизирует  $E[\text{длина}]$ .

Код Хаффмана доказуемо вычисляет оптимальное дерево, которое строится снизу вверх:

- 1. Создать лес с символом в каждом листе.
- 2. Пока не останется одно дерево:
- 3. Объединить два дерева с наименьшим количеством вхождений символов.

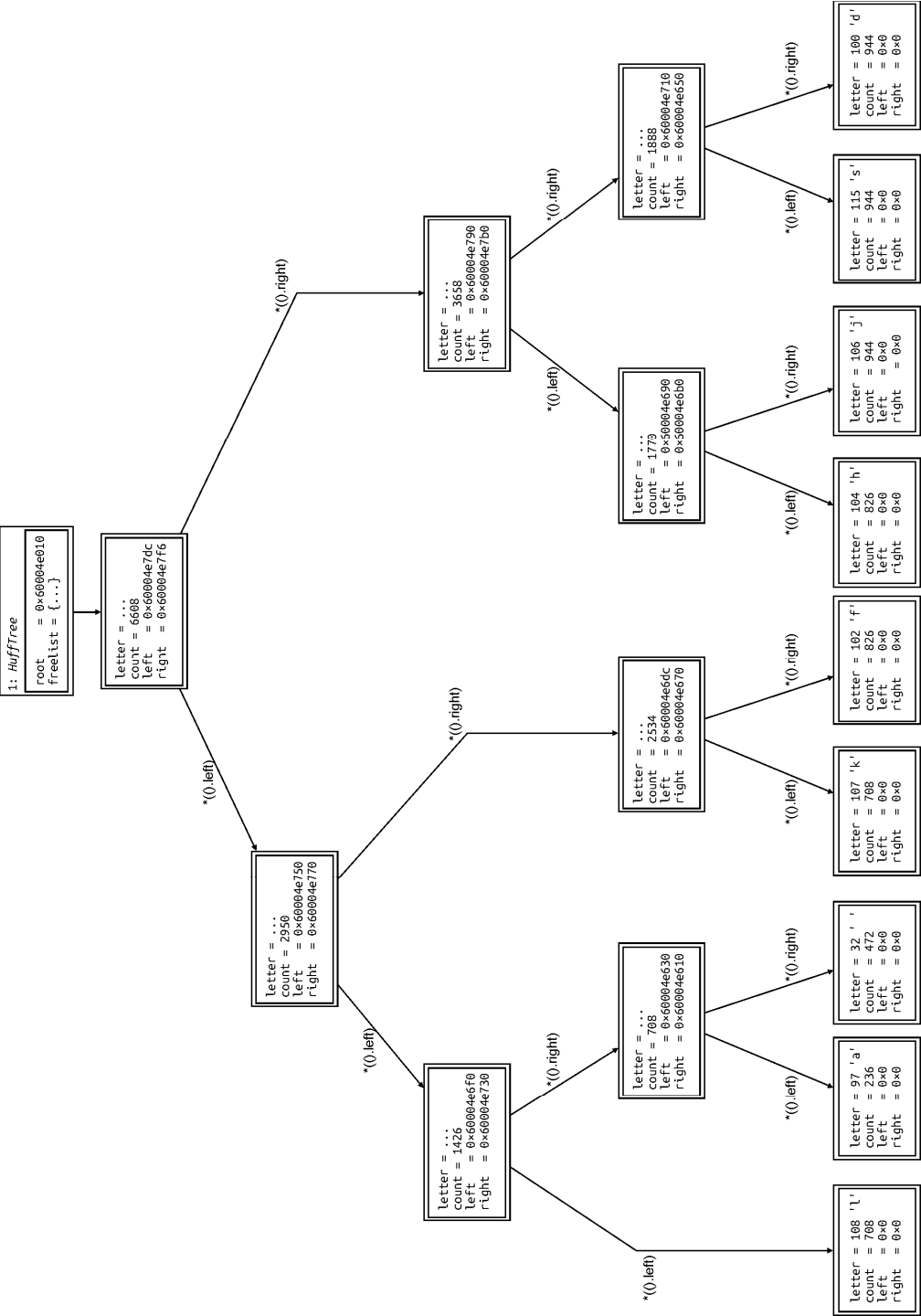


Рис. 15.2. Структура памяти дерева Хаффмана для большой строки ASCII



Таким образом, у каждого узла будет от 0 до 2 дочерних элементов, и чтобы понять, является ли он листом, нужно проверить, существует ли левый дочерний элемент. Структура памяти дерева Хаффмана для большой строки ASCII представлена на рис. 15.2.

```
struct HuffmanTree
{
    enum{W = numeric_limits<unsigned char>::digits, N = 1 << W};
    struct Node
    {
        unsigned char letter;
        int count;
        Node *left, *right;
        Node(int theCount, Node* theLeft, Node* theRight,
            unsigned char theLetter): left(theLeft), right(theRight),
            count(theCount), letter(theLetter) {}
        bool operator<(Node const& rhs) const{return count < rhs.count;}
    }* root;
    Freelist<Node> f;
    HuffmanTree(Vector<unsigned char> const& byteArray)
    { // вычисление частоты
        int counts[N];
        for(int i = 0; i < N; ++i) counts[i] = 0;
        for(int i = 0; i < byteArray.getSize(); ++i) ++counts[byteArray[i]];
        // создание листовых узлов
        Heap<Node*, PointerComparator<Node> > queue;
        for(int i = 0; i < N; ++i) if(counts[i] > 0) queue.insert(
            new(f.allocate())Node(counts[i], 0, 0, i));
        // слияние листовых узлов для создания дерева
        while(queue.getSize() > 1) // пока лес не объединится
        {
            Node *first = queue.deleteMin(), *second = queue.deleteMin();
            queue.insert(new(f.allocate())
                Node(first->count + second->count, first, second, 0));
        }
        root = queue.getMin();
    }
};
```

Обход дерева реализован как функция-член класса `Node`, и в процессе обхода создается книга кодов, которая сопоставляет каждый символ с кодом:

```
void populateCodebook(Bitset<unsigned char>* codebook)
{
    Bitset<unsigned char> temp;
    root->traverse(codebook, temp);
}

void traverse(Bitset<unsigned char>* codebook,
    Bitset<unsigned char>& currentCode)
{
    if(left) // внутренний узел
    {
        currentCode.append(false); // движение влево
```

```

        left->traverse(codebook, currentCode);
        currentCode.removeLast();
        currentCode.append(true); // движение вправо
        right->traverse(codebook, currentCode);
        currentCode.removeLast();
    }
    else codebook[letter] = currentCode; // лист
}

```

В книгу кодов записывается кодовое слово для любого символа в виде вектора байтов с дополнительными битами:

```

Vector<unsigned char> HuffmanCompress(Vector<unsigned char> const& byteArray)
{
    HuffmanTree tree(byteArray);
    Bitset<unsigned char> codebook[HuffmanTree::N], result;
    tree.populateCodebook(codebook);
    tree.writeTree(result);
    for(int i = 0; i < byteArray.getSize(); ++i)
        result.appendBitset(codebook[byteArray[i]]);
    return ExtraBitsCompress(result);
}

```

Кодовая книга записывается путем предварительного обхода дерева — для не листа пишем 0 и рекурсивно обходим потомков, а для листа пишем 1 и 8-битный символ. Например, для дерева, показанного на рис. 15.1, получим запись *01a01b01c1d*, где символы заменены соответствующими битовыми последовательностями. Функция `append` является функцией-членом `Node`:

```

void append(Bitset<unsigned char>& result)
{
    result.append(!left); // 0, если не лист,
                          // 1 в противном случае
    if(left)
    {
        left->append(result);
        right->append(result);
    }
    else result.appendValue(letter, W);
}

```

```

void writeTree(Bitset<unsigned char>& result){root->append(result);}

```

В ходе декодирования мы читаем дерево и используем его, чтобы декодировать каждый символ, применяя код для перехода к листьям:

```

Vector<unsigned char> HuffmanUncompress(Vector<unsigned char> const& byteArray)
{
    BitStream text(ExtraBitsUncompress(byteArray));
    HuffmanTree tree(text);
    return tree.decode(text);
}

Node* readHuffmanTree(BitStream& text)
{
    Node *left = 0, *right = 0;
    unsigned char letter;

```

```

    if(text.readBit()) letter = text.readValue(W); // попадание в лист
    else
    { // рекурсивная обработка внутренних узлов
        left = readHuffmanTree(text);
        right = readHuffmanTree(text);
    }
    return new(f.allocate())Node(0, left, right, letter);
}
HuffmanTree(BitStream& text){root = readHuffmanTree(text);}
Vector<unsigned char> decode(BitStream& text)
{ // неправильные биты дают неправильный результат, но код не вылетает
    Vector<unsigned char> result;
    for(Node* current = root;;
        current = text.readBit() ? current->right : current->left)
    {
        if(!current->left)
        {
            result.append(current->left->letter);
            current = root;
        }
        if(!text.bitsLeft()) break;
    }
    return result;
}

```

Для кодирования текста длины  $n$  и алфавита  $A$ , где  $n > |A|$ , требуется время  $O(n \lg(|A|))$ , а для декодирования —  $O(n)$ .  $E[\text{длина кода}] - H(X) < 1$  (см. [15.3]). Таким образом, для двоичного алфавита сжатие не используется, а для модели с  $k$  соединенными символами  $E[\text{длина кода}] - H(X) < 1/k$ . Например, для 8-битных символов избыточность битов  $\leq 12,5\%$ . В небольших алфавитах для эффективности используются суперсимволы. Очень большой файл можно для удобства разделить на блоки и сжать каждый отдельно с минимальной потерей эффективности. Коды Хаффмана работают с общими алфавитами, используя хеш-таблицу для хранения кодовой книги.

## 15.8. Сжатие словаря

Попробуем закодировать слово по его положению в списке всех найденных слов. Простейшим методом сжатия словаря является *LZW*. В начале кодирования или декодирования имеется словарь, содержащий все отдельные байты, и поддерживается один и тот же словарь размера  $n$ . Словарь кодировщика до/после записи слова  $j$  соответствует словарю декодера соответственно до/после чтения слова  $j + 1$ .

Кодировка:

1. Инициализировать словарь со всеми односимвольными словами.
2. Текущее слово  $w$  пусто.
3. Пока не закончится файл:
4.     Добавить прочитанный байт в  $w$ .

5. Если  $w \notin$  словарю:
6. Вывести индекс  $w$  без последнего байта.
7. Добавить  $w$  в словарь, если есть место.
8. Установить  $w$  в прочитанный байт.
9. Вывести индекс  $w$ .

Индексы слов кодируются в двоичном формате с использованием  $\lceil \lg(n) \rceil$ , и именно такое количество необходимо для чтения любого предыдущего слова. Например, для кодирования последовательности *abhababa* это будет выглядеть так:

Прочитанная буква	Текущее слово	Есть ли в словаре?	Следующий индекс	Вывод
<i>a</i>	<i>a</i>	да	256	none
<i>b</i>	<i>ab</i>	нет	256	97( <i>a</i> ), 8 bits
<i>h</i>	<i>bh</i>	нет	257	98( <i>b</i> ), 9 bits
<i>a</i>	<i>ha</i>	нет	258	104( <i>h</i> ), 9 bits
<i>b</i>	<i>ab</i>	да	258	none
<i>a</i>	<i>aba</i>	нет	259	256( <i>ab</i> ), 9 bits
<i>b</i>	<i>ab</i>	да	259	none
<i>a</i>	<i>aba</i>	да	259	none
				259( <i>aba</i> ), 9 bits

Словарь имеет размер  $\leq 2^{maxBits}$  и представляет собой троичное дерево пошагового поиска. Хорошим значением для *maxBits* по умолчанию является 16 — большое значение не нужно, поскольку приведет к плохому сжатию с большим количеством битов на индекс. В следующем коде реализован битовый поток, сам алгоритм LZW есть в сети, и его можно изменить для работы с дисковыми потоками:

```
void LZWCompress(BitStream& in, BitStream& out, int maxBits = 16)
{
    assert(in.bytesLeft());
    byteEncode(maxBits, out); // сохранение конфигурации
    TernaryTreapTrie<int> dictionary;
    TernaryTreapTrie<int>::Handle h;
    int n = 0;
    while(n < (1 << numeric_limits<unsigned char>::digits))
    { // инициализация со всеми байтами
        unsigned char letter = n;
        dictionary.insert(&letter, 1, n++);
    }
    Vector<unsigned char> word;
    while(in.bytesLeft())
    {
        unsigned char c = in.readByte();
        word.append(c);
        // если значение найдено, добавление продолжается
        if(!dictionary.findIncremental(word.getArray(), word.getSize(), h))
```

```

{ // слово без гарантированного вхождения
  // последнего байта в словарь
  out.writeValue(*dictionary.find(word.getArray(),
    word.getSize() - 1), lgCeiling(n));
  if (n < twoPower(maxBits)) // добавить новое слово, если есть место
    dictionary.insert(word.getArray(), word.getSize(), n++);
  word = Vector<unsigned char>(1, c); // чтение байта
}
}
out.writeValue(*dictionary.find(word.getArray(), word.getSize()),
  lgCeiling(n));
}

```

$E[\text{время выполнения}] = O(n \times \text{maxBits})$  из-за добавочного поиска, но может быть меньше, в зависимости от текста.

В ходе расшифровки создается массив отображения индексов в слова. При не первом чтении индекса вставляется новое слово, соответствующее последнему индексу + первый символ слова, соответствующего индексу. Последнее  $\in$  словарю, если только оно не равно слову, добавленному в словарь после вывода последнего слова. Это может произойти только в том случае, если оно равно последнему слову + его первый символ. Например, если *ababa* декодируется, когда *ab* уже есть в словаре, *aba* добавляется в словарь и немедленно используется для декодирования суффикса *aba*. Первое слово читается с использованием 8 битов, потому что оно не добавляется в словарь, а для любого другого слова количество битов в следующем возможном индексе равно  $\min(\text{maxBits}, \lceil \lg(n + 1) \rceil)$ . Для заданного слова количество индексных битов одинаково после/до того, как кодировщик/декодер добавляет его. Декодирование примера кодирования выглядит так:

Следующий индекс	Последнее слово	Индекс чтения	Есть ли в словаре?	Добавленное слово
256	none	97(a), 8 bits	да	none
256	97(a)	98(b), 9 bits	да	ab
257	98(b)	104(h), 9 bits	да	bh
258	104(h)	257(ab), 9 bits	да	ha
259	257(ab)	259(aba), 9 bits	нет	aba

```

void LZWUncompress(BitStream& in, BitStream& out)
{
  int maxBits = byteDecode(in), size = twoPower(maxBits), n = 0,
    lastIndex = -1;
  assert(maxBits >= numeric_limits<unsigned char>::digits);
  Vector<Vector<unsigned char> > dictionary(size);
  for (; n < (1 << numeric_limits<unsigned char>::digits); ++n)
    dictionary[n].append(n);
  while (in.bitsLeft())
  {
    int index = in.readValue(lastIndex == -1 ? 8 :
      min(maxBits, lgCeiling(n + 1)));

```

```

if(lastIndex != -1 && n < size)
{
    Vector<unsigned char> word = dictionary[lastIndex];
    word.append((index == n ? word : dictionary[index])[0]);
    dictionary[n++] = word;
}
for(int i = 0; i < dictionary[index].getSize(); ++i)
    out.writeByte(dictionary[index][i]);
lastIndex = index;
}
}

```

Время выполнения равно  $O(n + 2^{\max Bits})$ .

## 15.9. Кодирование серий байтов

Подсчет повторяющихся байтов и вывод их в формате «байт и количество его повторов» эффективно позволяет сжимать повторяющиеся последовательности байтов. Один из способов организовать такое сжатие — зарезервировать *escape-символы*  $a = 255$  и  $b = 254$ , а также вывод любого символа  $c$  в виде:

- ◆  $cak$ , если количество оставшихся одинаковых символов  $k > 1$  или  $c = a$  и  $k = 1$ ;
- ◆  $ab$ , если  $c = a$  и  $k = 0$ ;
- ◆  $c$  в противном случае.

Если подсчетов меньше 253, они помещаются в байт и не сталкиваются с escape-символами. Например, последовательность байтов 0,0,0,0,127,127,255,254,255,255 кодируется как 0,255,4,127,127,255,254,254,255,255,1:

```

enum {RLE_E1 = (1 << numeric_limits<unsigned char>::digits) - 1,
      RLE_E2 = RLE_E1 - 1};
Vector<unsigned char> RLECompress(Vector<unsigned char> const& byteArray)
{
    Vector<unsigned char> result;
    for(int i = 0; i < byteArray.getSize(); )
    {
        unsigned char byte = byteArray[i++];
        result.append(byte);
        int count = 0;
        while(count < RLE_E2 - 1 && i + count < byteArray.getSize() &&
              byteArray[i + count] == byte) ++count;
        if(count > 1 || (byte == RLE_E1 && count == 1))
        {
            result.append(RLE_E1);
            result.append(count);
            i += count;
        }
        else if(byte == RLE_E1) result.append(RLE_E2);
    }
    return result;
}

```

Если значение  $a$  не используется, возможно только сжатие. После чтения  $e$  следующий байт является числом, если только это не  $a$  и не  $b$ . Если  $a$ , то следующий байт является счетчиком, а если  $b$ , то декодируется один символ  $a$ :

```
Vector<unsigned char> RLEUncompress(Vector<unsigned char> const& byteArray)
{
    Vector<unsigned char> result;
    for(int i = 0; i < byteArray.getSize(); )
    {
        unsigned char byte = byteArray[i++];
        if(byte == RLE_E1 && byteArray[i] != RLE_E1)
        {
            unsigned char count = byteArray[i++];
            if(count == RLE_E2) count = 1;
            else byte = result.lastItem(); // временное место на случай
                                           // перераспределения вектора
            while(count-->0) result.append(byte);
        }
        else result.append(byte);
    }
    return result;
}
```

Время выполнения в обоих случаях равно  $O(n)$ .

## 15.10. Перемещение на передний план

Алгоритм *MTFT* понижает ранг часто используемых чисел, что приводит к сжатию во время кодирования. Например, повторяющиеся символы превращаются в последовательности нулей. Алгоритм кодирования:

1. Поместить все символы алфавита в список в известном порядке — например, по возрастанию численных значений.
2. Для любого входного символа:
3. Вывести его ранг.
4. Переместить в начало списка (рис. 15.3).

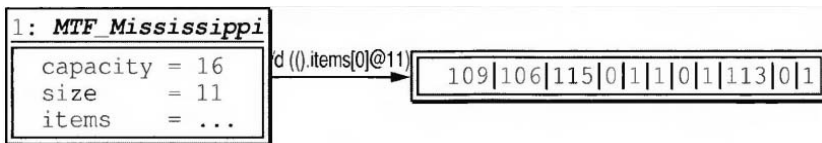


Рис. 15.3. MTFT для слова *mississippi*

Алгоритм декодирования:

1. Инициализировать список рангов одинаковыми значениями.
2. Для любого ранга:
3. Вывести символ на этой позиции.
4. Переместить ранг на передний план.

```

Vector<unsigned char> MoveToFrontTransform(bool compress,
    Vector<unsigned char> const& byteArray)
{
    unsigned char list[1 << numeric_limits<unsigned char>::digits], j, letter;
    for(int i = 0; i < sizeof(list); ++i) list[i] = i;
    Vector<unsigned char> resultArray;
    for(int i = 0; i < byteArray.getSize(); ++i)
    {
        if(compress)
        { // поиск и вывод ранга
            j = 0;
            letter = byteArray[i];
            while(list[j] != letter) ++j;
            resultArray.append(j);
        }
        else
        { // преобразование ранга в байты
            j = byteArray[i];
            letter = list[j];
            resultArray.append(letter);
        } // сдвиг списка назад, чтобы освободить место
            // для переднего элемента
        for(; j > 0; --j) list[j] = list[j - 1];
        list[0] = letter;
    }
    return resultArray;
}

```

Время выполнения кодирования и декодирования для алфавита  $A$  равно  $O(n|A|)$ , но с небольшими постоянными коэффициентами и удобством кеширования в качестве бонуса.

## 15.11. Преобразование Берроуза — Уилера

Преобразование строки по методу Берроуза — Уилера (Burrows — Wheeler Transform, BWT) состоит из последнего столбца символьной матрицы, сформированной из отсортированного списка возможных вращений строки, а также индекса исходного враще-

ния. Например, вот возможные вращения слова *click*:

<i>ckcli</i>
<i>click</i>
<i>ickcl</i>
<i>kclic</i>
<i>lickc</i>

. Исходное вращение на-

ходится в строке 1, а последний столбец равен *iklcc*. BWT упрощает сжатие вывода, потому что в каждом разделе обычно используется несколько отдельных символов, что, например, хорошо работает с алгоритмом MTFT.

Алгоритм массива суффиксов вычисляет BWT, используя функтор ранга BWT для сортировки вращений. Например, для слова *click* массив BWT равен [3, 0, 2, 4, 1]. Пусть  $t$  — это последний столбец, соответствующий преобразованной строке. Индекс исход-



ного вращения равен индексу вращения с суффиксом в индексе 0. Поскольку каждая строка является результатом вращения, ее самый правый символ предшествует самому левому, поэтому  $t[i] = \text{string}[(\text{BWT}[i] - 1) \% n]$ . Время выполнения составляет  $O(n \lg(n))$ :

```
Vector<unsigned char> BurrowsWheelerTransform(
    Vector<unsigned char> const& byteArray)
{
    int original = 0, size = byteArray.getSize();
    Vector<int> BTWArray = suffixArray<BWTRank>(byteArray.getArray(), size);
    Vector<unsigned char> result;
    for(int i = 0; i < size; ++i)
    {
        int suffixIndex = BTWArray[i];
        if(suffixIndex == 0)
        { // поиск исходной строки
            original = i;
            suffixIndex = size; // обход возможного кратного размера
                                // на следующем шаге
        }
        result.append(byteArray[suffixIndex - 1]);
    } // предполагается, что 4 байтов достаточно
    Vector<unsigned char> code = ReinterpretEncode(original, 4);
    for(int i = 0; i < code.getSize(); ++i) result.append(code[i]);
    return result;
}
```

Обратное преобразование выполняется за  $O(n)$ . Подсчеты символов используются для создания следующих массивов:

- ◆ *ранги* — сколько одинаковых символов в  $t$  предшествует символу в этой позиции. Ранги в левом столбце равны рангам в правом столбце, потому что для любых строк  $xs$  и  $ys$ , оканчивающихся одним и тем же символом  $c$ , их правое вращение имеет вид  $sx$  и  $sy$ , и если  $xs < ys$ , то  $sx < sy$ . Например, для *iklcc* с исходной позицией 1 ранги равны 00001;
- ◆ *первые позиции* — первые позиции символов в левом столбце вычислимы, поскольку они отсортированы и лежат в диапазоне  $[0, 255]$ . Первые позиции для *cikl* — 0234. Если расширить алгоритм для работы с другим диапазоном, нужно извлечь отсортированный список уникальных символов из  $t$ .

Интуитивно первые позиции определяют все вращения, начинающиеся с буквы  $c$ , а ранги говорят, какой из них соответствует символу правого столбца, потому что ранги в обоих столбцах одинаковы. Если преобразовать исходное вращение в обратном порядке, то  $t[\text{исходный индекс}]$  окажется последним символом строки. Для любого другого символа  $c = t[j]$  предыдущий текстовый символ равен  $t[\text{первые позиции}[c] + \text{ранги}[j]]$ , потому что это позиция вращения со строкой  $c$  в первом столбце и требуемым следующим символом в последнем. Для строки *iklcc* последовательность вращения:  $1 \rightarrow k$ ,  $3 + 0 = 3 \rightarrow c$ ,  $0 + 0 = 0 \rightarrow i$ ,  $2 + 0 = 2 \rightarrow l$ ,  $4 + 0 = 4 \rightarrow c$ , а обратное преобразование дает строку *click*:

```
Vector<unsigned char> BurrowsWheelerReverseTransform(
    Vector<unsigned char> const& byteArray)
```

```
{
    enum{M = 1 << numeric_limits<unsigned char>::digits};
    int counts[M], firstPositions[M], textSize = byteArray.getSize() - 4;
    for(int i = 0; i < M; ++i) counts[i] = 0;
    Vector<int> ranks(textSize); // вычисление рангов
    for(int i = 0; i < textSize; ++i) ranks[i] = counts[byteArray[i]]++;
    firstPositions[0] = 0; // вычисление первых позиций
    for(int i = 0; i < M - 1; ++i)
        firstPositions[i + 1] = firstPositions[i] + counts[i];
    Vector<unsigned char> index, result(textSize); // извлечение
                                                // исходного вращения
    for(int i = 0; i < 4; ++i) index.append(byteArray[i + textSize]);
    // сборка в обратном порядке
    for(int i = textSize - 1, ix = ReinterpretDecode(index); i >= 0; --i)
        ix = ranks[ix] + firstPositions[result[i] = byteArray[ix]];
    return result;
}
```

Выход системы сжатия равняется Huffman(RLE(MTFT(BWT(вход)))), реализованному в BZIP2, который работает быстро и эффективно. В MTFT используется группировка символов BWT, которая формирует небольшие числа. RLE сжимает серии 0, 1 и 2, и его значение  $a = 255$  вряд ли появится в выводе MTFT:

```
Vector<unsigned char> BWTCompress(Vector<unsigned char> const& byteArray)
{
    return HuffmanCompress(RLECompress(MoveToFrontTransform(true,
        BurrowsWheelerTransform(byteArray))));
}
Vector<unsigned char> BWTUncompress(Vector<unsigned char> const& byteArray)
{
    return BurrowsWheelerReverseTransform(MoveToFrontTransform(false,
        RLEUncompress(HuffmanUncompress(byteArray))));
}
```

Тестовые метрики описанных здесь алгоритмов сжатия представлены на рис. 15.4.

Файл	Размер	Хаффман	BWT	LZW	7-Zip BZIP2
a.txt	1	3	9	2	37
bible.txt	4047392	2218529	934693	1417732	846185
dickens.txt	31457485	17786401	9046529	13252448	8899726
ecoli.txt	4638690	1159679	1319213	1213574	1250991
mobydick.txt	1191463	667648	410470	495146	371466
pi10mm.txt	10000000	4249754	4381817	4524962	4308898
world192.txt	2473400	1558714	489749	925798	489543

Рис. 15.4. Тестовые метрики, описанные в главе алгоритмов сжатия

В большинстве случаев, а особенно для работы с текстом, лучше всего работает алгоритм BWT, но мы это знали и так. BZIP2 работает лучше, чем моя реализация, но он сильно оптимизирован.

## 15.12. Примечания по реализации

Только алгоритм RLE позволяет хоть как-то поиграться с настройками параметров. Остальные реализации сделаны, что называется, «по учебнику». У набора битов подразумевается поддержка функций потока, но может быть полезно добавить в API парочку других функций — таких как прибавление целого набора битов за раз.

## 15.13. Комментарии

Теория информации полна любопытных определений, но даже ее изобретатель признал, что их использование не упрощает решение задач.

Существует множество других статических кодов, например:

- ♦ в коде *Tunstall* используется двоичный код, и когда размер алфавита не равен степени двойки, остальные значения кодируются парами символов. Например, если есть алфавит  $a, b, c, ab$  кодируется как 3. Но в этом коде трудно определить, какие пары наиболее вероятны;
- ♦ код *Фибоначчи* (см. [15.4]) немного эффективнее гамма-кода для значений от 8 и выше, но всего на 1 бит лучше, чем байт-код для значений больше 16, а в какой-то момент становится даже хуже. Он более сложен и, по всей видимости, не имеет своей ниши для использования;
- ♦ некоторые коды оптимальны для конкретных распределений данных: например, код *Golomb* — для геометрического и *Bigna* — для степенного закона. Но такие алгоритмы сложнее и не универсальны.

*Арифметические коды* являют собой альтернативу кодам Хаффмана. Они сложны в реализации и намного медленнее, но немного лучше выполняют сжатие, особенно при работе с небольшими алфавитами. Раньше их патентовали, но сейчас срок действия патентов истек. Коды Хаффмана являются более практичным вариантом. Если это вам интересно, обратитесь за подробностями к работе [15.2].

В алгоритме кодирования длин серий можно использовать различное количество escape-символов и разные для них значения. Мои эксперименты с BTW показывают, что эффективнее всего использовать два escape-символа с максимально возможными значениями.

Некоторые структуры данных, такие как *расширяющееся дерево*, могут сделать алгоритм MTFT асимптотически оптимальным, но они сложны, и у них высокие постоянные коэффициенты. Но MTFT обычно не является узким местом производительности конвейера сжатия, поэтому большого смысла от использования деревьев нет.

Для реализации BWT существуют сложные алгоритмы, обеспечивающие время  $O(n)$ , и к тому же появляются новые. BZIP2 — лучший универсальный компрессор. Если важна эффективность, стоит отдать предпочтение более быстрым словарным методам — таким как *GZIP*. Есть также варианты *PPM*, которые чуть эффективнее, но гораздо медленнее (см. [15.3]).

Скорость распаковки обычно более важна, чем скорость сжатия, потому что обычно файл сжимается один раз, а потом многократно распаковывается, — например, если его скачивают с сервера разные пользователи. Однако в задачах вроде резервного копиро-

вания файлов скорость сжатия важнее, поскольку большинство файлов, будучи запакованными, больше никогда не используются.

Когда распакованные данные отличаются от оригинала, говорят о сжатии с *потерями*. Такие алгоритмы полезны для работы с мультимедийными данными и реализованы такими стандартами, как MP3 и MPEG. Идея состоит в том, чтобы снизить качество, но так, чтобы люди не заметили особой разницы, и применить методы без потерь. Снижение качества зависит от среды, но часто используют преобразование Фурье и отбрасывают более высокие частоты. Подробности приведены в работе [15.3].

## 15.14. Советы по дополнительной подготовке

- ◆ Рассмотрите реализацию простого кодирования RLE, в котором использовалось бы только число  $a$ . В этом случае мы выводим *сак*, если  $k > 1$  или  $c = a$ , иначе выводим  $c$ . Изучите влияние сжатия BWT на производительность.
- ◆ Реализуйте сжатие LZ77, используемое в GZIP. Сравните его производительность и эффективность с LZW. Учитывая, что большинство современных программ сжатия использует варианты LZ77, подумайте, есть ли вообще смысл от алгоритма LZW, несмотря на простоту его реализации?
- ◆ Исследуйте и реализуйте компрессор PPM и сравните его с BZIP2.

## 15.15. Список рекомендуемой литературы

- 15.1. Adjero D., Bell T. C., & Mukherjee A. (2008). The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching. Springer.
- 15.2. Moffat A., & Turpin A. (2002). Compression and Coding Algorithms. Springer.
- 15.3. Salomon D. & Motta G. (2010). Handbook of Data Compression. Springer.
- 15.4. Sayood K. (2002). Lossless Compression Handbook. Academic Press.
- 15.5. Sedgewick R., & Wayne K. (2011). Algorithms. Addison-Wesley.

## 16. Комбинаторная оптимизация

Осел, не способный выбрать один из двух стогов, остается голодным.  
*Русская народная поговорка*

Нужно лишь единожды просчитать все возможности.  
*Александр Котов о шахматных ходах*

### 16.1. Введение

Эта глава в основном посвящена общим методам комбинаторной оптимизации. Вопросы численной оптимизации будут рассмотрены позже в отдельной главе. Здесь же мы сделаем акцент на основных метаэвристических методах, а также кратко опишем процесс введения ограничений.

Допустим, требуется найти наилучший выбор среди, возможно, бесчисленного множества, используя некоторый показатель качества. Различные алгоритмы ищут решение через поиск минимума этого показателя, потому что максимум — это тот же минимум, но с минусом. Искомое значение для этого выбора называют *экономической полезностью*, т. е. степенью счастья, которое дает этот выбор. Например, эмпирическая полезность для человеческого счастья как функция богатства считается равной  $O(\lg(\text{богатство}))$ . Но обычно параметр полезности математически неудобен или неизвестен, поэтому вместо него используют функции тождества.

Проверка каждого возможного выбора требует слишком много ресурсов, но в некоторых задачах прямой перебор допустим. Здесь пригодятся комбинаторные алгоритмы генерации (см. главу 12. *Разные алгоритмы и методы*).

### 16.2. Теория сложности

В этом разделе приведен краткий обзор наиболее важных идей. В работе [16.14] можно найти более подробную информацию.

У всех задач есть определенная сложность. Наиболее важные классы сложности:

- ◆ *легкая/разрешимая* — задачу можно решить за полиномиальное время  $O(n^k)$  для некоторой константы  $k$ ;
- ◆ *сложная/трудноразрешимая* — время на решение превышает полиномиальное;
- ◆ *неразрешимая* — ни один алгоритм не позволяет решить задачу.

Невозможно решить задачу, если для нее недостаточно информации. Например, нельзя предсказать будущее. Но существуют и полностью определенные неразрешимые задачи. Например, *задача остановки*: пусть есть программа  $X$ , тогда входит ли  $X$  в бесконечный цикл при некоторых входных данных? Если существует программа  $A$  (источник

$X$ , входные данные), которая вычисляет ответ *завершение/цикл*, создадим программу  $B$ (источник  $X$ ), которая вызывает  $A$ (источник  $X$ , источник  $X$ ), и, если ответ *завершение*, входит в бесконечный цикл. Вызываем  $B$ (источник  $B$ ). Если  $A$ (источник  $B$ , источник  $B$ ), вызванная  $B$ , *завершается*,  $B$ (источник  $B$ ) переходит в цикл, и, если это так,  $B$  (источник  $B$ ) завершается, что противоречит ответу  $A$  в обоих случаях. Основная характеристика неразрешимости заключается в попадании в цикл без информации, когда произойдет выход. Технически на компьютерах с конечной памятью неразрешимости не существует, потому что любая программа, состояние памяти которой повторяется, находится в бесконечном цикле, но на практике от этого свойства пользы нет.

Задача принадлежит  $NP$ , если она выдает ответ в формате *да/нет*, и существует алгоритм с полиномиальным временем, который может проверить ее правильность. Вычисление является *простым* относительно класса  $C$ , если на его выполнение требуется меньше ресурсов, чем на решение самой сложной задачи в классе  $C$ , — например, алгоритм с полиномиальным временем прост относительно  $NP$ . Задача  $X$  является  $C$ -*сложной*, если можно использовать ее в качестве решателя «черного ящика» для любой задачи  $A \in C$  после приведения экземпляра  $A$  к экземпляру  $X$  с помощью алгоритма, простого относительно  $C$ . Задача называется  $C$ -*полной*, если она  $C$ -сложна и принадлежит  $C$ .

Многие  $NP$ -полные задачи должны решаться с минимальными затратами. Они эквивалентны *задачам принятия решений*, в которых ставится вопрос, существует ли решение с конкретной стоимостью, и ответ на него получается с помощью экспоненциального поиска (см. главу 23. *Численные алгоритмы: работа с функциями*) по стоимости решателем «черного ящика». Самые полезные задачи оптимизации являются  $NP$ -сложными.

Задачи, принадлежащие  $PSPACE$ , требуют полиномиального пространства и сложнее, чем  $NP$ -полные. Они часто возникают в многопользовательских играх. Например: как поставить мат за  $n$  ходов в шахматах, независимо от того, что делает противник.

Отношения между различными классами сложности задач показаны на рис. 16.1.

В алгоритме, который решает сложную задачу, придется пожертвовать одним из следующих свойств:

- ♦ полиномиальное время в худшем случае — постарайтесь достичь хорошей скорости в среднем;

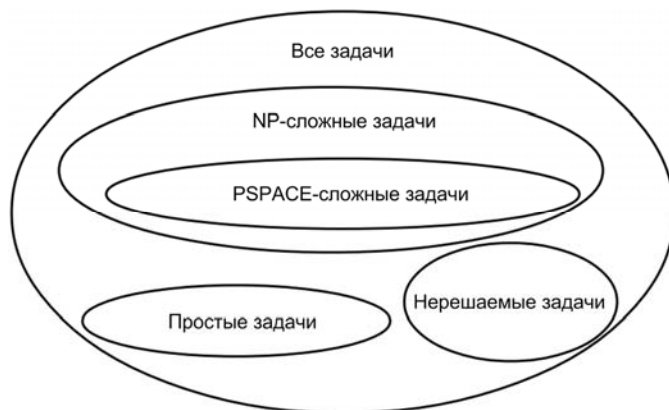


Рис. 16.1. Отношения между различными классами сложности задач

- ♦ точность ответа — можно получить достаточно близкое приближение;
- ♦ работа при любых условиях — задача будет решаться только при соблюдении определенной структуры.

### 16.3. Типичные сложные задачи

Рассмотренные далее задачи просты в описании и часто встречаются в литературе по оптимизации. К ним применяются наиболее естественные типы представления решения, включая перестановку, подмножество и разбиение. Задачи с комбинациями не рассматриваются. Во всех случаях для создания заданных наборов определенного размера используется простой генератор случайных чисел. Основное внимание уделяется тестированию и сравнению алгоритмов решения, поэтому не обращайте внимания на сложность получающихся задач или выбор генератора. Для каждого из представленных классов задач опубликованы большие их коллекции, часто сопровождаемые оптимальными решениями. У каждой задачи есть один или несколько известных алгоритмов решения. Они приведены в литературе — здесь же только показано, как общие алгоритмы могут быть применены к общим задачам с минимальными усилиями.

В задаче коммивояжера (Traveling Salesman Problem, TSP) дан набор точек и расстояний между каждой их парой. Нужна такая перестановка, при которой посещение точек в указанном порядке минимизирует суммарное пройденное расстояние. В условие может включаться или не включаться возврат из последней точки в первую. Расстояния обычно евклидовы (рис. 16.2).

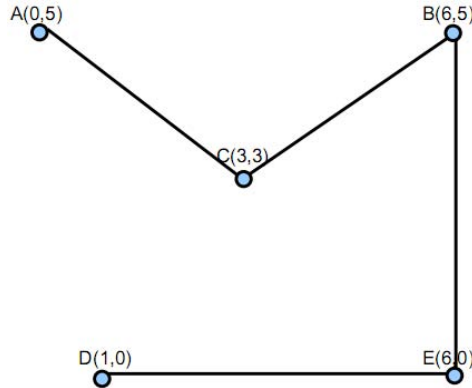


Рис. 16.2. Решение 5-точечной евклидовой задачи коммивояжера без возврата

Задача коммивояжера (ЗК) является эталоном для тестирования новых алгоритмов оптимизации, и о ней известно очень много (см. [16.1]). Например:

- ♦ для случайных точек  $\in [0, 1]^2$   $E[\text{лучшая стоимость обхода}]$  равна  $\sqrt{\frac{n}{2}}$ ;
- ♦ для евклидова расстояния лучшее решение состоит в обходе вершин выпуклой оболочки в порядке выпуклой оболочки (см. главу 18. *Вычислительная геометрия*).

Чтобы сгенерировать случайный экземпляр задачи для тестирования, можно использовать приведенный далее код. Решатель ожидает, что любая структура данных задачи

способна оценить представление своего решения. В задачах на перестановку ожидается возможность оценить один шаг с указанием, является ли он возвратом в исходное положение. Понятие `EuclideanDistance` обсуждается в главе 18. *Вычислительная геометрия*:

```
struct TSPRandomInstance
{
    Vector<Vector<double> > points;
    TSPRandomInstance(int n)
    {
        for(int i = 0; i < n; ++i)
        {
            Vector<double> point;
            for(int j = 0; j < 2; ++j) point.append(GlobalRNG().uniform01());
            points.append(point);
        }
    }
    double evalStep(int from, int to, bool isReturn = false) const
    { // оценка одного шага
        if(isReturn) return 0; // бесплатный возврат
        assert(from >= 0 && from < points.getSize() && to >= 0 &&
            to < points.getSize());
        EuclideanDistance<Vector<double> >::Distance d;
        return d(points[from], points[to]);
    }
    double operator()(Vector<int> const& permutation) const
    { // оценка полного решения
        assert(permutation.getSize() == points.getSize());
        double result = 0;
        for(int i = 1; i < points.getSize(); ++i)
            result += evalStep(permutation[i - 1], permutation[i]);
        return result;
    }
};
```

*Задача об укладке рюкзака (knapsack problem)* — это максимизация веса и количества укладываемых предметов с заданными значениями пользы и веса. Требуется выбрать подмножество товаров, которое максимизирует суммарную пользу так, что их суммарный вес не превышает вместимость.

Простой генератор принимает равномерное распределение пользы и весов, а вместимость делает равной  $\sum \text{весов}/2$ :

```
class KnapsackRandomInstance
{
    Vector<double> profits, weights;
    double capacity;
public:
    double getProfit(int i) const
    {
        assert(i >= 0 && i < profits.getSize());
        return profits[i];
    }
};
```



```

double getWeight(int i) const
{
    assert(i >= 0 && i < weights.getSize());
    return weights[i];
}
int getSize() const { return profits.getSize(); }
double getCapacity() const { return capacity; }
KnapsackRandomInstance(int n): capacity(0)
{ // равномерное случайное распределение пользы и весов,
  // емкость равна половине суммы весов
    for(int i = 0; i < n; ++i)
    {
        profits.append(GlobalRNG().uniform01());
        weights.append(GlobalRNG().uniform01());
        capacity += weights[i];
    }
    capacity /= 2;
}
double operator()(Vector<bool> const& subset) const
{
    assert(subset.getSize() == getSize());
    double totalProfit = 0, totalWeight = 0;
    for(int i = 0; i < subset.getSize(); ++i) if(subset[i])
    {
        totalProfit += getProfit(i);
        totalWeight += getWeight(i);
    } // не подходящим предметам назначается бесконечный балл
    return totalWeight <= capacity ? -totalProfit :
        numeric_limits<double>::infinity();
}
};

```

В задаче выполнимости (Satisfiability Problem, SAT) задается  $n$  логических переменных и формула, представляющая собой объединение через логическое И многих предложений, каждое из которых является логическим ИЛИ нескольких переменных. Требуется найти подмножество истинных переменных такое, чтобы, когда остальные ложны, формула была верна, или объявить проблему неразрешимой. Задача оптимизации называется *задачей максимальной выполнимости* и требует удовлетворения как можно большего количества условий.

Простой генератор для варианта задачи с тремя переменными на предложение берет  $n$  переменных и  $m$  предложений, при этом нормальной сложностью считается  $m = 10n$ . Каждое предложение состоит из случайно выбранных и перевернутых переменных:

```

class Satisfiability3RandomInstance
{
    Vector<Vector<int> > clauses;
    int n;
public:
    Satisfiability3RandomInstance(int theN, int nc): n(theN),
        clauses(nc, Vector<int>(3))

```

```

{ // n предложений для n переменных
  for(int i = 0; i < nc; ++i)
  {
    Vector<int>& clause = clauses[i];
    for(int j = 0; j < clause.getSize(); ++j)
    { // случайные значения и знаки
      int variable = GlobalRNG().mod(n);
      clause[j] = variable * GlobalRNG().sign();
    }
  }
}

int getSize()const{return n;}
int getClauseSize()const{return clauses.getSize();}
Vector<int> const& getClause(int i)const
{
  assert(i >= 0 && i < getClauseSize());
  return clauses[i];
}
};

```

Задача *упаковки контейнера* (bin packing problem): есть набор предметов с известными весами и бесконечное число ящиков с вместимостью, превышающий максимальный вес предметов. Нужно разделить предметы так, чтобы для любой части сумма весов контейнера и количество используемых контейнеров было минимальным.

В простом варианте генератора используются случайные веса и вместимость, равная 5:

```

class BinPackingRandomInstance
{
  Vector<double> weights;
  double binSize;
public:
  double getWeight(int i)const
  {
    assert(i >= 0 && i < weights.getSize());
    return weights[i];
  }
  double getBinSize()const{return binSize;}
  BinPackingRandomInstance(int n): binSize(5) // равномерно
    // распределенные случайные веса
  {for(int i = 0; i < n; ++i) weights.append(GlobalRNG().uniform01());}
};

```

Задача *целочисленного программирования* (integer programming problem) — это задача из области линейного программирования (см. главу 24. Численная оптимизация), где каждая переменная решения является дискретной. Округление соответствующих переменных решения линейной программы в меньшую сторону дает субоптимальное решение, поскольку оптимальное округляет в меньшую сторону только некоторые из них, но в остальном — это хорошая отправная точка для дальнейшего улучшения.

Многие задачи можно свести к целочисленному программированию, и для этого нужно выразить задачу в терминах целочисленного программирования, а там уже хорошо спроектированные решатели умеют решать задачи малого и среднего размера. Сегодня

существует много хороших коммерческих и бесплатных решателей, но эту тему мы обсуждать не будем.

Указанные задачи являются NP-полными и предполагают нахождение некоторого комбинаторного объекта, в том числе перестановки, подмножества или разбиения, дающего наилучший результат.

## 16.4. Алгоритмы приближения

Иногда можно получить решение за полиномиальное время, субоптимальное не более чем в  $C$  раз. К таким алгоритмам аппроксимации часто относятся жадные эвристики. Например, в задаче упаковки контейнеров *описанная далее* стратегия подбора является 2-аппроксимированной (см. [16.8]). Алгоритм работает в реальном времени:

1. Если следующий элемент не влезает в текущий контейнер, закрываем его и открываем другой.
2. Кладем элемент в текущую корзину.

Алгоритмы аппроксимации полезны только в случае, когда они быстрые и простые. К сожалению:

- ◆ их нужно разработать и доказать, что они дают  $C$ -приближение для любой задачи;
- ◆ у многих задач можно доказать, что не существует приближенного алгоритма с достаточно хорошим  $C$ ;
- ◆ многие алгоритмы аппроксимации имеют полиномиальное время выполнения высокого порядка;
- ◆ решения, найденные эвристическими методами, обычно лучше.

Для задачи упаковки контейнеров лучший приближенный алгоритм — *уменьшение по первому приближению*. Элементы сортируются по весу и сначала укладываются большие объемы. Этот алгоритм тоже жадный и довольно хорошо выполняет упаковку:

```
template<typename PROBLEM>
Vector<Vector<int>> > firstFitDecreasing(PROBLEM const& p, int n)
{
    assert(n > 0);
    Vector<pair<double, int>> > items(n);
    double binSize = p.getBinSize();
    for(int i = 0; i < n; ++i)
    {
        items[i].first = p.getWeight(i);
        assert(items[i].first <= binSize); // в противном случае
                                           // задача плохая
        items[i].second = i;
    }
    quickSort(items.getArray(), 0, n - 1,
        PairFirstComparator<double, int, ReverseComparator<double>> >());
    Vector<Vector<int>> > result(1); // в начале контейнер пуст
    Vector<double> sums(1, 0);
    for(int i = 0; i < n; ++i)
    {
        bool fit = false;
```

```

for(int bin = 0; bin < result.getSize(); ++bin)
    if(sums[bin] + items[i].first <= binSize)
    {
        fit = true;
        result[bin].append(items[i].second);
        sums[bin] += items[i].first;
        break;
    }
    else if(bin == result.getSize() - 1)
    { // не помещается в последний контейнер, добавление нового
        result.append(Vector<int>());
        sums.append(0);
    }
}
return result;
}

```

Соответствующая теория и связанные исследования приведены в работе [16.6].

## 16.5. Эвристика жадного построения

Во многих сложных задачах можно получить хорошее решение, построив его по жадным критериям. Некоторые из них являются приближенными алгоритмами. Например, в задаче о рюкзаке можно выбирать предметы путем уменьшения соотношения прибыли/затрат до тех пор, пока следующий такой предмет не превысит вместимость. Этот алгоритм обычно дает хорошие результаты, но не является приближенным:

```

template<typename PROBLEM>
Vector<pair<double, int> > getKnapsackRatios(PROBLEM const& p)
{
    int n = p.getSize();
    double capacity = p.getCapacity();
    Vector<pair<double, int> > ratios(n);
    for(int i = 0; i < n; ++i)
        ratios[i] = make_pair(p.getProfit(i)/p.getWeight(i), i);
    quickSort(ratios.getArray(), 0, n - 1,
        PairFirstComparator<double, int, ReverseComparator<double> >());
    return ratios;
}

template<typename PROBLEM>
Vector<bool> approximateKnapsackGreedy(PROBLEM const& p)
{
    double remainingCapacity = p.getCapacity();
    Vector<pair<double, int> > ratios = getKnapsackRatios(p);
    Vector<bool> solution(ratios.getSize(), false);
    for(int i = 0; i < ratios.getSize(); ++i)
    {
        int j = ratios[i].second;
        if(remainingCapacity > p.getWeight(j))
        {
            remainingCapacity -= p.getWeight(j);

```

```
        solution[j] = true;
    }
}
return solution;
}
```

Жадные алгоритмы обычно масштабируются до задач, по крайней мере, среднего размера, поэтому их используют довольно часто.

## 16.6. Ветвление и границы

Для некоторых задач с инкрементально конструируемыми решениями можно вычислить *нижнюю границу* остальной части решения. Например, в задаче коммивояжера стоимость посещения оставшихся городов  $\geq$  стоимости их MST + стоимость кратчайшего пути от последнего посещенного города до любого непосещенного. *Алгоритм ветвления и границ* (Branch and Bound, B&B) начинает с бесконечно большой глобальной нижней границы и рекурсивно перечисляет все решения, указывая по одному компоненту за раз (рис. 16.3). Нижние границы позволяют:

- ♦ попробовать наиболее перспективный компонент следующим;
- ♦ не рассматривать компонент, когда имеющаяся стоимость + нижняя граница  $\geq$  глобальной нижней границе.

Следующий Совокупная стоимость MST Стоимость MST Ребро соединения Стоимость ребра Нижняя граница	Выбор A				
	В	С	Д	Е	
		6	3.61	5.1	7.81
	DC + ED	ED + EB	CB + EC	DC + CB	
		8.61	10	7.85	7.21
	CB	DC	DC	ED	
		3.61	3.61	3.61	5
	18.21		17.21		16.55
	Отклонено		Отклонено		Отклонено
Следующий Совокупная стоимость MST Стоимость MST Ребро соединения Стоимость ребра Нижняя граница	Выбор D			Выбор C	
	В	С	Е	В	Д
		12.17	8.7	10.1	7.21
	EC	EB	CB	ED	EB
		4.24	5	3.61	5
	DC	CB	EC	DC	CB
		3.61	3.61	4.24	3.61
	20.02		17.31		17.95
	Отклонено		Отклонено		Отклонено
Следующий Совокупная стоимость Ребро соединения Стоимость ребра Нижняя граница	Выбор C		Выбор B		Выбор D
	В	Е	Д	Е	В
		12.31	12.95	14.28	12.21
	EB	EB		ED	EB
		5	5	5	5
	17.31		17.95		19.28
	Отклонено		Отклонено		Отклонено
	17.31		17.21		17.21
	Отклонено		Отклонено		Отклонено
Следующий Совокупная стоимость	Выбор B		Выбор E		
	Е		Д		
	17.31		17.21		

Рис. 16.3. Расчеты алгоритма B&B для 3К, показанной на рис. 16.1. Сначала выбирается маршрут ADCB, затем, в конце концов, находится оптимальный маршрут ACBE. Столь же оптимальный путь ACDE отклонен

Если задать достаточно жесткие нижние границы, алгоритм рано отбросит многие субоптимальные решения и относительно быстро найдет оптимальное. В&В — это алгоритм с произвольной остановкой, т. е. его преждевременная остановка дает наилучший ответ, найденный до сих пор, при условии, что хотя бы одно полное решение было получено. Чтобы сделать реализацию универсальной, структура данных задачи вычисляет нижние границы. Общее правило, применимое для этой и других подобных реализаций, состоит в том, что проблемный объект не должен иметь какого-либо модифицируемого состояния, и должна быть возможность передать его алгоритму. В приведенной далее реализации также убрана рекурсия. Базовый принцип состоит в том, чтобы отменить предыдущий шаг, выполнить следующий шаг и создать новое состояние и новые следующие шаги:

```
template<typename PROBLEM> pair<typename PROBLEM::X, bool> branchAndBound(
    PROBLEM const& p, int maxLowerBounds = 1000000)
{ // до остановки нужно получить хотя бы одно полное решение
    typename PROBLEM::X best; // предполагается наличие конструктора по умолчанию
    double bestScore = numeric_limits<double>::infinity();
    typename PROBLEM::INCREMENTAL_STATE is = p.getInitialState();
    bool foundCompleteSolution = false; // гарантируется полнота решения
    struct BBState
    {
        Vector<pair<double, typename PROBLEM::Move> > moves;
        int prevMove;
    };
    Stack<BBState> states;
    bool start = true;
    while(!states.isEmpty() || start)
    {
        bool goNextLevel = start, isSolutionComplete = false;
        if(start) start = false;
        else // работа на текущем уровне
        {
            BBState& level = states.getTop();
            int& i = level.prevMove;
            // отмена предыдущего шага, если он был
            if(i != -1) p.undoMove(is, level.moves[i].second);
            ++i; // проверка следующего шага, если он есть
            if(i < level.moves.getSize() && level.moves[i].first < bestScore)
            { // если шаг не отброшен и есть куда идти, делается следующий шаг
                p.move(is, level.moves[i].second);
                if(p.isSolutionComplete(is))
                { // обновление лучшего решения
                    double score = p.evaluate(is);
                    if(score < bestScore)
                    {
                        best = p.extractSolution(is);
                        bestScore = score;
                    } // обновление флагов
                    isSolutionComplete = true;
                    foundCompleteSolution = true;
                }
            }
        }
    }
}
```

```

        else goNextLevel = true;
    }
}
if(goNextLevel && (!foundCompleteSolution || maxLowerBounds > 0))
{ // настройка следующего уровня
    BBState levelNext = {p.generateMoves(is), -1};
    int m = levelNext.moves.getSize();
    assert(m > 0); // в противном случае задача нерешаема
    maxLowerBounds -= m;
    quickSort(levelNext.moves.getArray(), 0, m - 1,
        PairFirstComparator<double, typename PROBLEM::Move>());
    states.push(levelNext);
} // если решение полное, отменяем текущий шаг
// и больше не генерируем
else if(!isSolutionComplete) states.pop(); // возврат к предыдущему шагу
}
return make_pair(best, maxLowerBounds > 0);
}

```

Время выполнения зависит от качества нижних границ и может быть в худшем случае экспоненциальным.

Приведенный далее код позволяет использовать В&В для задачи коммивояжера. Но он не масштабируется за пределы  $n = 100$ , потому что вычисление списка перемещений занимает время  $O(n^3)$  из-за дорогостоящего вычисления нижней границы MST:

```

template<typename PROBLEM> double findMSTCost(PROBLEM const& instance,
    Vector<int> const& remPoints)
{
    int n = remPoints.getSize();
    if(n <= 1) return 0;
    GraphAA<double> g(n);
    for(int i = 0; i < n; ++i)
        for(int j = i + 1; j < n; ++j) g.addUndirectedEdge(i, j,
            instance.evalStep(remPoints[i], remPoints[j]));
    assert(validateGraph(g));
    Vector<int> parents = MST(g);
    double sum = 0;
    for(int i = 0; i < parents.getSize(); ++i)
    {
        int parent = parents[i];
        if(parent != -1)
            sum += instance.evalStep(remPoints[i], remPoints[parent]);
    }
    return sum;
}

template<typename PROBLEM> class BranchAndBoundPermutation
{
    PROBLEM const& p;
    int const n;
public:
    typedef Vector<int> INCREMENTAL_STATE;
    INCREMENTAL_STATE getInitialState() const{return INCREMENTAL_STATE();}
}

```

```

typedef Vector<int> X;
BranchAndBoundPermutation(int theN, PROBLEM const& theProblem): n(theN),
    p(theProblem){}
typedef int Move;
bool isSolutionComplete(INCREMENTAL_STATE const& permutation) const
{
    return permutation.getSize() == n;
}
double evaluate(INCREMENTAL_STATE const& permutation) const
{
    assert(isSolutionComplete(permutation));
    return p(permutation);
}
X extractSolution(INCREMENTAL_STATE const& permutation) const
{
    assert(isSolutionComplete(permutation));
    return permutation;
}
Vector<pair<double, Move> > generateMoves(
    INCREMENTAL_STATE const& permutation) const
{
    double sumNow = 0;
    Vector<bool> isIncluded(n, false);
    for(int i = 0; i < permutation.getSize(); ++i)
    {
        isIncluded[permutation[i]] = true;
        if(i > 0) sumNow += p.evalStep(permutation[i - 1], permutation[i]);
    }
    Vector<pair<double, Move> > result;
    for(int i = 0; i < n; ++i)
        if(!isIncluded[i])
        {
            Vector<int> remainder;
            for(int j = 0; j < n; ++j) if(j != i && !isIncluded[j])
                remainder.append(j);
            double lb = sumNow + (permutation.getSize() > 0 ?
                p.evalStep(permutation.lastItem(), i) : 0) +
                findMSTCost(p, remainder);
            result.append(pair<double, Move>(lb, i));
        }
    return result;
}
void move(INCREMENTAL_STATE& permutation, Move m) const
{
    permutation.append(m);
}
void undoMove(INCREMENTAL_STATE& permutation, Move m) const
{
    permutation.removeLast();
}
};

template<typename INSTANCE> Vector<int> solveTSPBranchAndBound(
    INSTANCE const& instance, int maxLowerBounds)
{
    return branchAndBound(BranchAndBoundPermutation<INSTANCE>(
        instance.points.getSize(), instance), maxLowerBounds).first;
}

```



В задаче об упаковке рюкзака в качестве простого механизма определения нижней границы можно применить расширение жадного алгоритма с использованием дробных элементов, чтобы заполнить оставшуюся емкость лучшими элементами с точки зрения прибыли/затрат. Когда жадный алгоритм отбрасывает предмет из-за чрезмерного веса, он перебирает остальные предметы, стараясь упаковать как можно больше, чтобы получить строгое улучшение. Но для нижней границы предположим, что можно взять долю пропущенного элемента в размере оставшейся вместимости. Это наилучший сценарий, а значит, и правильная нижняя граница:

```
template<typename PROBLEM> class BranchAndBoundKnapsack
{
    PROBLEM const& p; // порядок 1
    int const n; // порядок 2
    Vector<pair<double, int> > const ratios;
public:
    struct INCREMENTAL_STATE
    {
        Vector<bool> subset;
        int current;
        double currentTotalProfit, currentTotalWeight;
    };
    INCREMENTAL_STATE getInitialState() const
    {
        INCREMENTAL_STATE is = {Vector<bool>(n, false), 0, 0, 0};
        return is;
    }
    typedef Vector<bool> X;
    BranchAndBoundKnapsack(PROBLEM const& theProblem): p(theProblem),
        n(p.getSize()), ratios(getKnapsackRatios(p)) {}
    typedef pair<int, bool> Move;
    bool isSolutionComplete(INCREMENTAL_STATE const& is) const
    {return is.current == n;}
    double evaluate(INCREMENTAL_STATE const& is) const
    {
        assert(isSolutionComplete(is));
        return p(is.subset);
    }
    X extractSolution(INCREMENTAL_STATE const& is) const
    {
        assert(isSolutionComplete(is));
        return is.subset;
    }
    Vector<pair<double, Move> > generateMoves(INCREMENTAL_STATE const& is) const
    {
        Vector<pair<double, Move> > result;
        for(int i = 0; i < 2; ++i)
        {
            double lb = is.currentTotalProfit,
                remainingCapacity = p.getCapacity() - is.currentTotalWeight;
            int j = ratios[is.current].second;
```

```

        if(i) // текущий шаг
        {
            remainingCapacity -= p.getWeight(j);
            lb += p.getProfit(j);
        }
        if(remainingCapacity >= 0) // оставшиеся шаги, если таковые есть
        {
            for(int k = is.current + 1; k < n; ++k)
            {
                int j = ratios[k].second;
                if(remainingCapacity >= p.getWeight(j))
                {
                    remainingCapacity -= p.getWeight(j);
                    lb += p.getProfit(j);
                }
                else
                { // достигнут неподходящий элемент, используются дроби
                    lb += remainingCapacity * ratios[k].first;
                    break;
                }
            }
            result.append(pair<double, Move>(-lb, make_pair(j, i)));
        }
    }
    return result;
}

void move(INCREMENTAL_STATE& is, Move const& m) const
{
    if(m.second)
    {
        is.subset[m.first] = m.second;
        is.currentTotalProfit += p.getProfit(m.first);
        is.currentTotalWeight += p.getWeight(m.first);
    }
    ++is.current;
}

void undoMove(INCREMENTAL_STATE& is, Move const& m) const
{
    if(m.second)
    {
        is.subset[m.first] = false;
        is.currentTotalProfit -= p.getProfit(m.first);
        is.currentTotalWeight -= p.getWeight(m.first);
    }
    --is.current;
}

};

template<typename INSTANCE> Vector<bool> solveKnapsackBranchAndBound(
    INSTANCE const& instance, int maxLowerBounds)
{
    return branchAndBound(BranchAndBoundKnapsack<INSTANCE>(instance),
        maxLowerBounds).first;
}

```

В задаче коммивояжера алгоритм B&B не масштабируется из-за затрат памяти  $O(n^2)$  на хранение всех ходов. Можно использовать алгоритм *realtime A\**, который выполняет шаг только с наилучшей нижней границей. Он полезен, как и жадная эвристика построения:

```
template<typename PROBLEM> typename PROBLEM::X realtimeAStar(PROBLEM const& p)
{ // до остановки нужно получить хотя бы одно полное решение
  typename PROBLEM::INCREMENTAL_STATE is = p.getInitialState();
  do
  {
    Vector<pair<double, typename PROBLEM::Move> > moves =
      p.generateMoves(is);
    assert(moves.getSize() > 0); // иначе задача не решается
    int best = argMin(moves.getArray(), moves.getSize(),
      PairFirstComparator<double, typename PROBLEM::Move>());
    p.move(is, moves[best].second);
  }while(!p.isSolutionComplete(is));
  return p.extractSolution(is);
}
```

В задаче коммивояжера его можно использовать следующим образом:

```
template<typename INSTANCE> Vector<int> solveTSPTAS(INSTANCE const&
  instance)
{
  return realtimeAStar(BranchAndBoundPermutation<INSTANCE>(
    instance.points.getSize(), instance));
}
```

В задаче об упаковке нужно предположить, что выбор хода и результат точно такие же, как и у жадного алгоритма, но накладных расходов получается больше.

## 16.7. Поиск крачайшего пути в пространстве с нижними границами

*Пространство состояний* — это огромный неявный граф, в котором вершины — возможные состояния задачи, а ребра — действия по переходу из одного состояния в другое. Стоимость каждого ребра  $\geq 0$ . Кратчайший путь из начального состояния в целевое состояние образует решение. *A\** — это обобщение алгоритма Дейкстры, который отдает приоритет вершинам по известной стоимости + нижняя граница остатка пути к цели. Таким образом, он с самого начала не учитывает все вершины графа. Нижние границы и проверки цели зависят только от текущего состояния и пути к нему. *Открытое множество* содержит рассматриваемые узлы с наилучшими известными нижними границами, а *закрытое множество* содержит посещенные узлы с известными расстояниями пути:

1. Поместить начальный узел в открытое множество с приоритетом нижней границы.
2. Пока открытое множество не опустеет:
3. Взять узел с наилучшей нижней границей.

- 4. Обновить узел данными о расстояниях пути и поместить его в закрытое множество.
- 5. Если цель достигнута, вернуть путь решения.
- 6. Поместить его потомков в открытое множество. Если они там уже есть, обновить нижнюю границу, если она меньше.
- 7. Сообщить об отсутствии решения.

В открытом множестве для эффективного обновления дочерних нижних границ можно задействовать индексированную кучу. Для закрытого множества можно использовать хеш-таблицу от состояния до предыдущего состояния, чтобы можно было восстановить путь решения. Реализация предполагает, что у состояний есть уникальные идентификаторы, а вызывающая программа сопоставляет их со своим собственным представлением состояния. Передайте алгоритму закрытое множество, чтобы он мог вычислить нижние границы. Например, для решения ЗК граф представляет собой дерево, в котором любой узел имеет ребро, соединяющее его с каждым непроверенным городом (рис. 16.4).

Следующий	Выбор A				
Совокупная стоимость	B	C	D	E	
MST		6	3.61	5.1	7.81
Стоимость MST	DC + ED	ED + EB	CB + EC	DC + CB	
Ребро соединения	CB	DC	DC	ED	
Стоимость ребра	3.61	3.61	3.61	5	
Нижняя граница	18.21	17.21	16.55	20.02	

Следующий	Выбор D				Выбор C			
Совокупная стоимость	B	C	E		B	D	E	
MST	EC	EB	CB		ED	EB	DB	
Стоимость MST	4.24	5	3.61		5	5	7.07	
Ребро соединения	DC	CB	EC		DC	CB	DC	
Стоимость ребра	3.61	3.61	4.24		3.61	3.61	3.61	
Нижняя граница	20.02	17.31	17.95		15.82	15.82	18.52	

Следующий	Выбор B			Выбор D		
Совокупная стоимость	D	E		B	E	
Ребро соединения	ED	ED	EB	EB	EB	
Стоимость ребра	5	5	5	5	5	
Нижняя граница	19.28	17.21		19.28	17.21	

Следующий	Выбор E	
Совокупная стоимость	D	
	17.21	

Рис. 16.4. Расчеты A\* для ЗК с рис. 16.1. Сначала выбирается AD, затем пробуем ACD и, наконец, находим оптимальную траекторию ACBE

Для реализации нужно дать алгоритму получить доступ к закрытому множеству, потому что некоторым вычислениям оно требуется. Основным необходимым ресурсом является память, нужная для хранения закрытых и открытых состояний, поэтому следует ограничить общее возможное количество состояний. Но поскольку мы запоминаем каждое сгенерированное состояние, это также является ограничением на количество вычисляемых нижних границ, за исключением незначительной доли повторных вычислений:

```

template<typename PROBLEM> struct AStar
{ // закрытое множество хранится как дерево с родительскими указателями
    typedef typename PROBLEM::STATE_ID STATE_ID;
    typedef typename PROBLEM::HASHER HASHER;
    typedef ChainingHashTable<STATE_ID, STATE_ID, HASHER> L;
    class PredVisitor
    {
        L& pred;
    public:
        PredVisitor(L& thePred): pred(thePred){}
        STATE_ID const* getPred(STATE_ID x) const {return pred.find(x);}
        Vector<STATE_ID> getPath(STATE_ID x) const
        {
            Vector<STATE_ID> path;
            for(;;)
            {
                path.append(x);
                STATE_ID* px = pred.find(x);
                if(px) x = *px; // формируемый путь
                else break; // предыдущего шага нет
            }
            path.reverse(); // нужен путь в пути, а не порядок
                           // родительского указателя
            return path;
        }
    };
    static pair<Vector<STATE_ID>, bool> solve(PROBLEM const& p,
        int maxSetSize = 1000000)
    { // родитель текущего состояния
        typedef pair<double, STATE_ID> QNode;
        IndexedHeap<QNode,
            PairFirstComparator<double, STATE_ID>, STATE_ID, HASHER> pQ;
        L pred;
        PredVisitor v(pred);
        bool foundGoal = false;
        STATE_ID j = p.start(); // у начального состояния нет предыдущего
        pQ.insert(QNode(p.remainderLowerBound(p.nullState(), j, v),
            p.nullState()), j);
        while(!pQ.isEmpty() && pred.getSize() + pQ.getSize() < maxSetSize)
        {
            pair<QNode, STATE_ID> step = pQ.deleteMin();
            j = step.second;
            if(j != p.start())
                pred.insert(j, step.first.second); // теперь есть лучший предшественник
            if(p.isGoal(j, v))
            {
                foundGoal = true;
                break;
            }
            // вычитание нижней границы последнего хода, чтобы получить
            // точное расстояние
            double dj = step.first.first -
                p.remainderLowerBound(step.first.second, j, v);

```

```

Vector<STATE_ID> next = p.nextStates(j, v);
for(int i = 0; i < next.getSize(); ++i)
{
    STATE_ID to = next[i];
    double newChildLowerBound = dj + p.distance(j, to, v) +
        p.remainderLowerBound(j, to, v);
    QNode const* current = pQ.find(to);
    if((current && newChildLowerBound < current->first) ||
        (!current && !pred.find(to))) // обновление, если получилось лучше
        pQ.changeKey(QNode(newChildLowerBound, j), to);
}
} // формирование пути или лучшего текущего решения
return make_pair(v.getPath(j), foundGoal);
}
};

```

Алгоритм  $A^*$  оптимален для заданной функции нижней границы по количеству рассматриваемых состояний, от которых зависит время выполнения (см. [16.12]).  $A^*$  рассматривает меньше состояний, чем  $B\&B$ , потому что:

- ◆ дополняет узел лучшей глобальной нижней границей, а не лучшим дочерним элементом текущего узла;
- ◆ операция изменения ключа заставляет рассматривать только наилучший путь к текущему состоянию, а решения, основанные на неоптимальных путях, в дальнейшем вообще не рассматриваются.

Но  $B\&B$  выигрывает в других аспектах:

- ◆ структуры данных открытого/закрытого состояния могут стать слишком большим, и, если задача не будет решена с заданным ограничением памяти, будет возвращено только частичное решение. Алгоритму  $B\&B$  требуется память  $O(\text{длина пути})$ , поэтому он может позволить себе больше расчетов. Найденное решение может быть оптимальным, но недоказуемым;
- ◆ шаг по алгоритму  $A^*$  неэффективен по сравнению с  $B\&B$ , что важно для дешевых нижних границ;
- ◆ составить структуру данных задачи для алгоритма  $A^*$  сложнее, чем для  $B\&B$ .

Таким образом, для комбинаторных задач  $B\&B$  лучше подходит как решатель, дающий хорошее приближенное решение, а  $A^*$  хорош для доказательства оптимальности решения при малых  $n$ . Но для задач вроде решения кубика Рубика  $A^*$  работает лучше, потому что поиск  $B\&B$  в глубину в этом случае работает хуже, чем поиск  $A^*$  в ширину. Тем не менее  $B\&B$  работает с некоторой *итеративной стратегией* углубления, где максимальная глубина ограничивается поэтапно, используя удвоение пределов.

В комбинаторных задачах определение идентификаторов состояний в алгоритме  $A^*$  сложно, потому что нельзя определить идентификатор по положению. Алгоритм  $A^*$  предполагает уникальные состояния, но это зависит от конкретной задачи и не всегда очевидно, что должно быть уникальным. Состояние обычно зависит от:

1. Того, какие компоненты частичного решения присутствуют.
2. Последовательности, в которой эти компоненты добавляются.

$A^*$  хорошо справляется с пунктом 2. Таким образом, состояние должно основываться только на пункте 1. Для экономии памяти можно использовать структуру набора битов (хотя бы для части состояния). Эта структура используется для хранения приведенных далее перестановок ЗК, а последний элемент позволяет различать вращения.

Запоминание полного частичного решения в качестве состояния, как в алгоритме В&В, — плохая идея, потому что в обоих случаях используется больше памяти, а операция смены ключа не приносит выгоды. Более простое представление основано на индексировании каждого вычисления нижней границы с нуля. Полные решения создаются из последнего хода и его предшественников в закрытом множестве  $A^*$ . В результате получаем дешевое состояние из одного слова, но не имеем преимуществ от операции смены ключа. Таким образом, для задач, где пункт 2 не имеет значения, следует рассматривать только алгоритм В&В.

Алгоритм  $A^*$  не кажется полезным для решения ЗК. Даже если у нас выделено памяти аж на множество размера  $10^7$ , алгоритм может решать случайные экземпляры размера 35, но для 40 (в экспериментах используется приращение 5) уже не хватает памяти. В&В может доказать оптимальность для размера 30, но не 35 (хотя решение получается оптимально или близко). Поэтому приведенная далее реализация выполнена лишь ради доказательства концепции. Ей не нужны функции посетителя, в отличие от реализации, где применено простое представление состояния. Для сравнения: в последней используется примерно на порядок больше состояний, и она может решать экземпляры размером 30, но не 35:

```
template<typename PROBLEM> struct AStartTSPProblem
{
    PROBLEM const& problem;
    typedef pair<Bitset<>, int> STATE_ID;
    STATE_ID nullState() const
    { return make_pair(Bitset<>(problem.points.getSize()), -1); }
    struct HASHER
    {
        DataHash<> h;
        HASHER(unsigned long long m) : h(m) {}
        unsigned long long operator() (STATE_ID const& s) const
        {
            Vector<unsigned long long> storage = s.first.getStorage();
            storage.append(s.second);
            return h(storage);
        }
    }; // неизвестно, какой узел лучше выбрать в качестве первого
    STATE_ID start() const { return nullState(); }
    Vector<int> convertStatePath(Vector<STATE_ID> const& path) const
    { // начало первого состояния
        Vector<int> result;
        for(int i = 1; i < path.getSize(); ++i)
            result.append(path[i].second);
        return result;
    }
    Vector<int> findRemainder(STATE_ID const& id) const
    {
        Vector<int> result;
```

```

    for(int i = 0; i < id.first.getSize(); ++i)
        if(!id.first[i]) result.append(i);
    return result;
}
template<typename VISITOR>
bool isGoal(STATE_ID id, VISITOR const& dummy) const
{
    id.first.flip();
    return id.first.isZero();
}
template<typename VISITOR>
Vector<STATE_ID> nextStates(STATE_ID const& id, VISITOR const& dummy) const
{
    Vector<STATE_ID> result;
    Vector<int> remainder = findRemainder(id);
    for(int i = 0; i < remainder.getSize(); ++i)
    {
        STATE_ID to = id;
        to.first.set(remainder[i], true);
        to.second = remainder[i];
        result.append(to);
    }
    return result;
}
template<typename VISITOR> double remainderLowerBound(STATE_ID const& dummy,
    STATE_ID const& to, VISITOR const& dummy2) const
{
    return findMSTCost(problem, findRemainder(to));
}
template<typename VISITOR> double distance(STATE_ID const& j,
    STATE_ID const& to, VISITOR const& dummy) const
{
    return j == start() ? 0 : problem.evalStep(j.second, to.second);
}
};

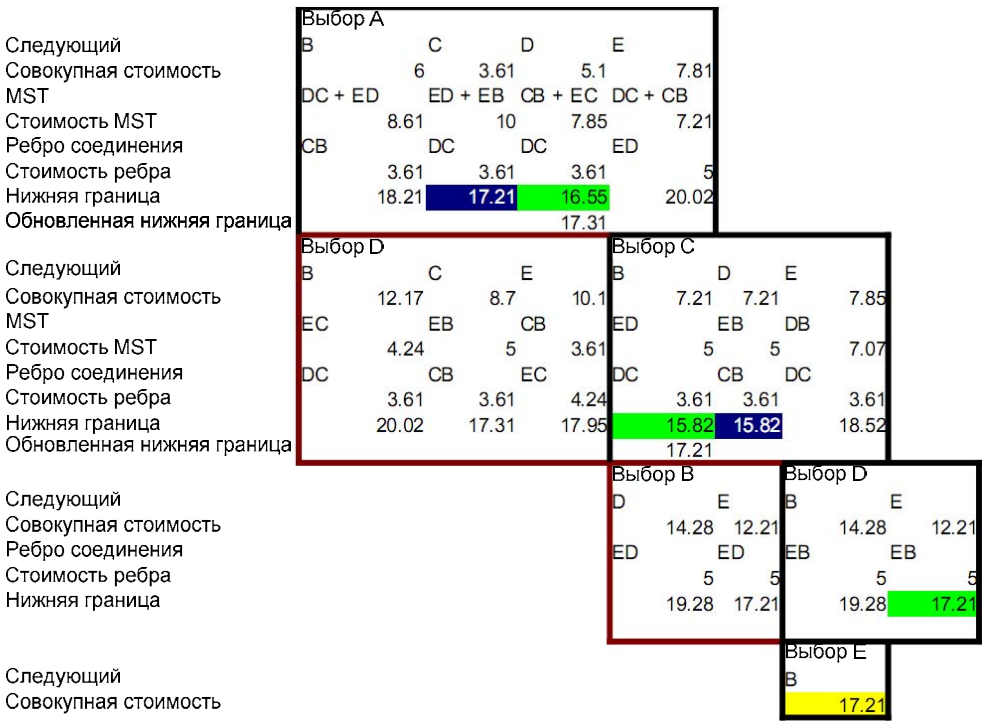
```

Алгоритм *рекурсивного поиска по первому наилучшему* (Recursive best-first search, RBFS) использует пространство  $O(\text{длина пути})$  (при условии ограниченности числа дочерних узлов), но просматривает некоторые состояния больше одного раза (рис. 16.5). Он сохраняет в каждом узле наиболее известное значение наилучшего альтернативного поддерева, доступного из любого узла-предка, и забывает некорневые узлы:

1. Текущий узел = корень, нижняя граница которого равна нулю либо  $\infty$ .
2. Поместить всех потомков в приоритетную очередь, где приоритет задается их нижней границей.
3. Пока цель не найдена, значение корня =  $\infty$ , если решения не существует или превышены доступные вычислительные ресурсы:
4. Взять лучшего потомка из очереди.
5. Если таковых нет, вернуть поддерево с нижней границей, равной  $\infty$ .
6. Если нижняя граница потомка больше, чем у альтернативного варианта, вернуть нижнюю границу.



- 7. В противном случае вернуться к потомку, альтернатива которого равна  $\min$ (второй по качеству приоритет дочернего элемента, альтернатива текущего узла).
- 8. Вернуть потомка обратно в очередь, обновив его нижнюю границу.



**Рис. 16.5.** Расчеты алгоритма RBFS для задачи коммивояжера с рис. 16.1. Сначала выбирается путь AD, затем ACB, потом ACD, а затем оптимальный ACDE

Алгоритм продвигается за счет того, что известные альтернативные нижние границы только увеличиваются. Таким образом, он не попадает в бесконечный цикл, даже если некоторые нижние границы равны нулю:

```
template<typename PROBLEM> struct RecursiveBestFirstSearch
{
    typedef typename PROBLEM::STATE_ID STATE_ID;
    typedef Stack<STATE_ID> L;
    L pred; // путь к цели
    PROBLEM const& p;
    enum{SUCCESS = -1, FAILURE = -2};
    typedef pair<double, STATE_ID> INFO; // нижняя граница и состояние
    bool foundGoal;
    Vector<STATE_ID> bestPath;
    class PredVisitor
    { // предполагается, что текущий узел всегда находится на вершине стека,
      // поэтому предыдущий узел - следующий элемент в стеке
      Stack<STATE_ID>& pred;
```

```

public:
    PredVisitor(L& thePred): pred(thePred){}
    STATE_ID const* getPred(STATE_ID dummy)const
    {
        Vector<STATE_ID>& storage = pred.storage;
        return storage.getSize() > 1 ? &storage[storage.getSize() - 2] : 0;
    }
    Vector<STATE_ID> getPath(STATE_ID dummy)const
    {
        Vector<STATE_ID> path;
        Stack<STATE_ID> s = pred;
        while(!s.isEmpty()) path.append(s.pop());
        path.reverse(); // путь не должен следовать родительскому указателю
        return path;
    }
};

double work(INFO state, double alternative, double pathCost,
    int& maxLowerBounds)
{ // остановка, если цель найдена, или возможные шаги кончились,
  // или исчерпаны вычислительные ресурсы
    PredVisitor v(pred);
    if(p.isGoal(state.second, v)) return SUCCESS;
    Vector<STATE_ID> next = p.nextStates(state.second, v);
    if(next.getSize() == 0) return numeric_limits<double>::infinity();
    if(maxLowerBounds < next.getSize()) return FAILURE;
    maxLowerBounds -= next.getSize();
    // сортировка потомков по нижней границе
    Heap<INFO, PairFirstComparator<double, STATE_ID> > children;
    for(int i = 0; i < next.getSize(); ++i)
        children.insert(INFO(max(state.first, pathCost +
            p.distance(state.second, next[i], v) +
            p.remainderLowerBound(state.second, next[i], v)), next[i]));
    for(;;)
    {
        INFO best = children.deleteMin();
        // не обрабатываем оставшихся потомков, если альтернатива
        // лучше, и возвращаем значение текущего лучшего потомка
        if(best.first > alternative) return best.first;
        // вычисляем d перед сохранением лучшего,
        // иначе разрываем инвариант
        double d = p.distance(state.second, best.second, v);
        pred.push(best.second);
        // используйте лучшую альтернативу и следующий лучший дочерний элемент
        best.first = work(best, children.isEmpty() ?
            alternative : min(children.getMin().first, alternative),
            pathCost + d, maxLowerBounds);
        if(best.first == SUCCESS) return SUCCESS;
        else if(best.first == FAILURE) return FAILURE;
        children.insert(best); // помещаем потомка в очередь с новой оценкой
        pred.pop(); // отменяем шаг
    }
}

```

```

RecursiveBestFirstSearch(PROBLEM const& theProblem,
    int maxLowerBounds = 10000000): p(theProblem), foundGoal(false)
{
    pred.push(p.start());
    foundGoal = (work(INFO(0.0, pred.getTop()),
        numeric_limits<double>::infinity(), 0, maxLowerBounds) == SUCCESS);
    PredVisitor v(pred);
    bestPath = v.getPath(pred.getTop());
}
};

```

Алгоритм RBFS использует слишком мало памяти. Время выполнения должно быть близко к  $A^*$ , но его трудно проанализировать, потому что многое зависит от успешности жадного выбора. Вы можете хранить значения забытых поддеревьев в кеше LRU, но это сложно реализовать.

Алгоритм RBFS более полезен в задаче коммивояжера, чем  $A^*$ , поскольку решает задачи с 20, а не с 25 экземплярами. Алгоритм имеет смысл для задач с дешевыми нижними границами — такими как кубик Рубика, решение которых алгоритмом  $A^*$  нарушает разумные ограничения памяти. Подробнее о решении таких головоломок можно почитать в работе [16.7].

## 16.8. Локальный поиск

В большинстве задач оптимизации аналогичные решения дают схожее качество. Таким образом, локальный поиск (также называемый *восхождением на холм*) начинается с начального решения, построенного случайным образом с помощью эвристики, алгоритма аппроксимации или других средств, а затем вносит изменения, пока не будет достигнут *локальный оптимум*. Возможные шаги:

- ◆ *первое обнаруженное улучшение* — полезно, когда нужно проверить все потенциальные ходы. Не масштабируется на больших задачах, т. к. очень расточительно генерировать большой список ходов, если первый ход находится быстро;
- ◆ *лучший* — наиболее полезный вариант, если лучший шаг эффективно вычисляется специальным алгоритмом, а не проверкой всех шагов;
- ◆ *случайное улучшение* — простой вариант, который хорошо масштабируется для больших задач, если большинство ходов ведет к улучшению решения. Но для обнаружения локального оптимума будет нужно проверить, что по крайней мере  $k$  последних ходов не дают улучшения. Можно задать значение  $k = 1000$  или выбрать другое в зависимости от задачи и выбранной стратегии.

При выполнении шагов можно делать *пошаговую оценку* — т. е. вычислять качество решения-кандидата по сравнению с качеством текущего решения и параметров шага за время  $O(n)$ , даже если пространство представления решения и время выполнения хода равны  $O(n)$ . В результате случайные, не приводящие к улучшению, ходы получаются дешевыми, и отклонить их — не проблема.

Реализация локального поиска допускает все типы перемещения и инкрементную оценку. Максимальный останов при оптимальном использовании равен 1. Для случайного улучшения хорошей стратегией является подсчет количества возможных ходов  $N$

для примененного оператора и использование, возможно,  $O(Mg(N))$  для каждой задачи. Также может иметь смысл выполнять оценку хвоста (см. [16.20]). Разрешено выполнять ход с нулевой разницей, чтобы избежать застревания на плато:

```
template<typename PROBLEM> typename PROBLEM::X localSearch(PROBLEM const& p,
    typename PROBLEM::X x, int maxMoves, int maxStall = -1)
{ // второе приведенное ниже значение - это балл
    typedef pair<typename PROBLEM::MOVE, double> MOVE;
    for(int i = 0; maxMoves-- && (maxStall == -1 || i < maxStall); ++i)
    {
        MOVE m = p.proposeMove(x);
        if(m.second <= 0)
        {
            i = -1; // в случае принятия счетчик сбрасывается
            p.applyMove(x, m.first);
        }
    }
    return x;
}
```

Существуют стандартные ходы, используемые для поиска решения (сложности приведены для случайного улучшения и задачи размера  $n$ ):

◆ перестановка:

- удалить элемент и вставить его между двумя другими —  $O(n)$ . Польза выглядит невеликой;
- поменять местами два соседних элемента —  $O(1)$ ;
- обратить порядок в некоторой части массива перестановок —  $O(n)$ . Эквивалентно обмену двумя ссылками при условии симметричной стоимости ребер. Для симметричных задач сложность равна  $O(1)$ ;
- поменять местами два соседних элемента —  $O(1)$ . Непопулярный метод в ЗК, является частным случаем обращения:

```
template<typename EVALUATOR> class SymmetricPermutationProblemReverseMove
{
    EVALUATOR const& e;
public:
    typedef pair<Vector<int>, typename EVALUATOR::INCREMENTAL_STATE> X;
    typedef pair<int, int> MOVE;
    SymmetricPermutationProblemReverseMove(EVALUATOR const& theE): e(theE){}
    pair<MOVE, double> proposeMove(X const& x) const
    {
        Vector<int> m = GlobalRNG().randomCombination(2, x.first.getSize());
        if(m[0] > m[1]) swap(m[0], m[1]);
        return make_pair(make_pair(m[0], m[1]),
            e.evalReverse(m[0], m[1], x.first, x.second));
    }
    void applyMove(X& x, MOVE const& m) const
    {
        assert(m.first < m.second && m.first >= 0 &&
            m.second < x.first.getSize());
```

```

        x.second = e.updateIncrementalStateReverse(m.first, m.second, x.first,
            x.second);
        x.first.reverse(m.first, m.second);
    }
    double getScore(X const& x) const { return e(x.first); }
};

template<typename EVALUATOR> class PermutationProblemSwapMove
{
    EVALUATOR const& e;
public:
    typedef pair<Vector<int>, typename EVALUATOR::INCREMENTAL_STATE> X;
    typedef pair<int, int> MOVE;
    PermutationProblemSwapMove(EVALUATOR const& theE): e(theE){}
    pair<MOVE, double> proposeMove(X const& x) const
    {
        Vector<int> m = GlobalRNG().randomCombination(2, x.first.getSize());
        if(m[0] > m[1]) swap(m[0], m[1]); // необязательно, но так чище
        return make_pair(make_pair(m[0], m[1]),
            e.evalSwap(m[0], m[1], x.first, x.second));
    }
    void applyMove(X& x, MOVE const& m) const
    {
        assert(m.first < m.second && m.first >= 0 &&
            m.second < x.first.getSize());
        x.second = e.updateIncrementalStateSwap(m.first, m.second, x.first,
            x.second);
        swap(x.first[m.first], x.first[m.second]);
    }
    double getScore(X const& x) const { return e(x.first); }
};

```

◆ подмножество:

- выбрать  $i$  и отменить выбор  $j$  элементов —  $O(i + j)$ . Простой частный случай этой операции — инвертирование выбора элемента:

```

template<typename EVALUATOR> class SubsetProblemFlipMove
{
    EVALUATOR const& e;
public:
    typedef pair<Vector<bool>, typename EVALUATOR::INCREMENTAL_STATE> X;
    typedef int MOVE;
    SubsetProblemFlipMove(EVALUATOR const& theE): e(theE){}
    pair<MOVE, double> proposeMove(X const& x) const
    {
        int i = GlobalRNG().mod(x.first.getSize());
        return make_pair(i, e.evalStep(i, x.first, x.second));
    }
    void applyMove(X& x, MOVE const& i) const
    { // обновление состояния
        assert(i >= 0 && i < x.first.getSize());
        x.second = e.updateIncrementalState(i, x.first, x.second);
    }
};

```

```

        // обмен
        x.first[i] = !x.first[i];
    }
    double getScore(X const& x) const { return e(x.first); }
};

```

◆ разделение:

- перемещение  $i$  элементов из раздела  $A$  в раздел  $B$  и  $j$  элементов из  $B$  в  $A$  выполняется за  $O(i + j)$ . Простой случай — переместить один элемент из одного раздела в другой. Удобным представлением разделов является список списков элементов, которые принадлежат друг другу:

```

template<typename EVALUATOR> class PartitionProblemSwapMove
{
    EVALUATOR const& e;
public:
    typedef pair<Vector<Vector<int>>,
    typename EVALUATOR::INCREMENTAL_STATE> X;
    struct MOVE{int index, from, to;};
    PartitionProblemSwapMove(EVALUATOR const& theE): e(theE){}
    pair<MOVE, double> proposeMove(X const& x) const
    {
        assert(x.first.getSize() >= 2); // в противном случае выполнение прерывается
        Vector<int> fromTo =
            GlobalRNG().randomCombination(2, x.first.getSize());
        Vector<int> const& binl = x.first[fromTo[0]];
        assert(binl.getSize() > 0); // в противном случае задача
                                   // не решается
        int index = GlobalRNG().mod(binl.getSize());
        MOVE m = {index, fromTo[0], fromTo[1]};
        return make_pair(m, e.evalStep(index, m.from, m.to, x.first,
            x.second));
    }
    void applyMove(X& x, MOVE const& m) const
    { // обновление состояния
        assert(m.from >= 0 && m.from < x.first.getSize() &&
            m.to >= 0 && m.to < x.first.getSize() && m.index >= 0 &&
            m.index < x.first[m.from].getSize());
        x.second = e.updateIncrementalState(m.index, m.from, m.to, x.first,
            x.second);
        // обмен
        Vector<int>& binl = x.first[m.from];
        x.first[m.to].append(binl[m.index]);
        binl[m.index] = binl.lastItem();
        binl.removeLast();
        // удаление binl, если он опустел
        if(binl.getSize() == 0)
        {
            binl = x.first.lastItem();
            x.first.removeLast();
        }
    }
};

```

```
double getScore(X const& x) const {return e(x.first);}
};
```

В задачах перестановки (вроде 3К) локальный поиск можно использовать следующим образом:

```
template<typename INSTANCE> Vector<int>
solveSymmetricPermutationLocalSearchReverse(INSTANCE const& instance,
Vector<int> const& initial, int maxMoves)
{ // ограничение локального поиска задается как 10nlg(n)
int nPossibleMoves = initial.getSize() * initial.getSize()/2;
return localSearch(SymmetricPermutationProblemReverseMove<INSTANCE>(
instance), make_pair(initial, instance.getIncrementalState(initial)),
maxMoves, int(10 * nPossibleMoves * log(nPossibleMoves))).first;
}

template<typename INSTANCE> Vector<int> solvePermutationLocalSearchSwap(
INSTANCE const& instance, Vector<int> const& initial, int maxMoves)
{ // ограничение локального поиска задается как 10nlg(n)
int nPossibleMoves = initial.getSize() * initial.getSize()/2;
return localSearch(PermutationProblemSwapMove<INSTANCE>(
instance), make_pair(initial, instance.getIncrementalState(initial)),
maxMoves, int(10 * nPossibleMoves * log(nPossibleMoves))).first;
}
```

Для задач подмножества — таких как задача об упаковке рюкзака и задача о максимальной выполнимости — локальный поиск реализуется следующим образом:

```
template<typename INSTANCE> Vector<bool> solveSubsetLocalSearchFlip(
INSTANCE const& instance, Vector<bool> const& initial, int maxMoves)
{ // ограничение локального поиска задается как 10nlg(n)
return localSearch(SubsetProblemFlipMove<INSTANCE>(instance),
make_pair(initial, instance.getIncrementalState(initial)), maxMoves,
int(10 * initial.getSize() * log(initial.getSize()))).first;
}
```

Для задачи разделения — таких как укладка контейнера — локальный поиск используется следующим образом:

```
template<typename INSTANCE> Vector<Vector<int> >
solvePartitionLocalSearchSwap(INSTANCE const& instance,
Vector<Vector<int> > const& initial, int maxMoves)
{ // ограничение локального поиска задается как 10nlg(n)
int nPossibleMoves = initial.getSize() * initial.getSize()/2;
return localSearch(PartitionProblemSwapMove<INSTANCE>(instance),
make_pair(initial, instance.getIncrementalState(initial)), maxMoves,
int(10 * nPossibleMoves * log(nPossibleMoves))).first;
}
```

Приведенные здесь методы стандартны для объектно-ориентированного проектирования и не ориентированы на конкретные задачи, что позволяет использовать их многократно, но это иногда негативно влияет на эффективность. А вот алгоритмы В&В и А\* ориентированы на задачу. Методы, в ограничениях у которых нельзя задать надлежащую штрафную функцию, также должны быть связаны с задачей, по крайней мере, в рамках конкретного способа обеспечения выполнимости.

Тип выполняемых шагов определяет граф с вершинами, соответствующими решениям, а ребра соединяют эти вершины с другими вершинами, до которых можно добраться за один шаг. Количество итераций локального поиска  $\leq$  размера самого длинного пути улучшения и равно  $O(\text{экспонента от } n)$ , поэтому количество итераций всегда следует ограничивать. Тип шага считается *полным*, если граф сильно связан. Полнота графа позволяет посетить каждое решение. Визуально граф похож на гористый ландшафт, где высота вершины пропорциональна качеству решения.

Лучшее из возможных решений называют *глобальным оптимумом*. Локальный поиск может застрять в локальном оптимуме более низкого качества. К сложным ландшафтным особенностям, затрудняющим локальный поиск, относятся:

- ◆ *плато* — плоская область, в которой все решения имеют примерно одинаковое качество;
- ◆ *лунка для гольфа* — локальный оптимум, который значительно лучше, чем все близлежащие решения.

*Окрестность решения* — это множество всех решений, в которые можно перейти из него за один шаг. Размером окрестности называют количество допустимых ходов. Окрестность *содержит* другую, если в нее входит каждое ее решение. Локальный оптимум в одной окрестности может не быть таковым в другой, не содержащейся в ней окрестности. Вы можете делать ходы из нескольких окрестностей, которые не содержат друг друга, и менять окрестность в случае застревания. Также вы можете делать более «длинный» шаг в большой окрестности, если малый шаг не дает покинуть малую окрестность.

*Метаэвристикой* принято называть стратегии, позволяющие избежать застревания в локальном оптимуме. Многие из них были предложены и популяризированы, но в этой главе мы обсудим лишь некоторые из них:

- ◆ практическую ценность имеют только те эвристики, которые направляют локальный поиск. Во всей литературе этот вывод делается единодушно. Во многих наиболее эффективных вариантах так или иначе используется локальный поиск, поскольку он слишком хорош, чтобы его игнорировать;
- ◆ следует рассматривать только методы, как можно меньше требующие от пользователя. Минимальным требованием является случайный шаг, зависящий от предметной области, и возможность его постепенной оценки. Возможно также выполнить смешивание двух решений, создавать искусственное возмущение, которое трудно отменить локальным перемещением, отменить тот или иной шаг и т. д.;
- ◆ отдавайте предпочтение более простым и устоявшимся алгоритмам, которые используются часто и успешно. Сообщество исследователей само занимается отбором лучших кандидатов и повышением их производительности. Как правило, любые улучшения оказываются незначительными, поэтому переходить на новые многообещающие методы пользователи стараются как можно позже;
- ◆ рассматривайте только простые и наиболее логичные настройки параметров, если в литературе не указано иное. Экспериментировать с десятками стратегий должно исследовательское сообщество, а не пользователь. Любые отклонения от «стандартной» конфигурации должны быть логически обоснованы, а повышение производительности должно подтверждаться экспериментами. Например, если значение пара-



метра равно 1, пытаться использовать значение 1.1 глупо, т. к. на любом разумном наборе тестов это даст лучшую производительность только при многократном тестировании (см. главу 21. *Вычислительная статистика*);

- ◆ не все алгоритмы и их варианты позволяют легко настраивать параметры с помощью механизма «самонастройки», который сам вычисляет значения по умолчанию. Мы рассматриваем только те алгоритмы, в которых такая возможность есть, поскольку любая настройка, которую пользователь может выполнить вручную, должна быть автоматизирована;
- ◆ не все интересные идеи превращаются в алгоритмы. Существует много стратегий, которые могут показаться разумными, но их реализация потребует подгонки под конкретные задачи, а от этого страдает универсальность.

## 16.9. Применение локального поиска к некоторым задачам

Самая сложная задача — реализовать эффективную инкрементную оценку для работы с общими шагами. В задачах с ограничениями — таких как упаковка рюкзака или контейнера — для простоты можно ввести небольшой штраф. Работа с возможными решениями не позволит использовать общие шаги многократно, но с практической точки зрения может оказаться лучше. Рассматривать этот вариант мы не будем.

В задаче коммивояжера оценку обратных и обменных шагов нужно выполнять осторожно, чтобы не допустить глупых ошибок. Но в остальном реализация простая. Инкрементное состояние — это текущая оценка, которая обычно содержит минимальную необходимую информацию. В приведенной далее и в других реализациях этого раздела создан интерфейс, который будет ожидать шаги. Он состоит из методов `evaluate*`, `updateIncrementalState*`, `getIncrementalState*` и `operator()`:

```
template<typename PROBLEM> class TSPPProxy
{
    PROBLEM const& p;
    double evalStep(Vector<int> const& permutation, int from, int to) const
    {return p.evalStep(permutation[from], permutation[to], to == 0);}

public:
    typedef double INCREMENTAL_STATE; // текущий балл
    TSPPProxy(PROBLEM const& theP): p(theP) {}
    INCREMENTAL_STATE updateIncrementalStateReverse(int i, int j,
        Vector<int> const& permutation, INCREMENTAL_STATE const& is) const
    { // берутся ребра с i-1 по i и с j по j + 1.
        // Прибавляем i-1 к j и i к j + 1
        int n = permutation.getSize();
        double imlFactor = i > 0 ? evalStep(permutation, i - 1, j) -
            evalStep(permutation, i - 1, i) : 0,
            jplFactor = j + 1 < n ? evalStep(permutation, i, j + 1) -
            evalStep(permutation, j, j + 1) : 0;
        return is + imlFactor + jplFactor;
    }
}
```

```

double evalReverse(int i, int j, Vector<int> const& permutation,
    INCREMENTAL_STATE const& is) const
{
    return updateIncrementalStateReverse(i, j, permutation, is) - is;
}
INCREMENTAL_STATE updateIncrementalStateSwap(int i, int j,
    Vector<int> const& permutation, INCREMENTAL_STATE const& is) const
{
    // берутся ребра c i-1 по i и c j по j + 1.
    // Прибавляем j-1 к j и j к j + 1
    // прибавление ребер i-1 к j и j к i + 1, j-1 к i и i к j + 1
    int n = permutation.getSize();
    double imlFactor = i > 0 ? evalStep(permutation, i - 1, j) -
        evalStep(permutation, i - 1, i) : 0,
        iplFactor = i + 1 < n ? evalStep(permutation, j, i + 1) -
        evalStep(permutation, i, i + 1) : 0,
        jmlFactor = j > 0 ? evalStep(permutation, j - 1, i) -
        evalStep(permutation, j - 1, j) : 0,
        jplFactor = j + 1 < n ? evalStep(permutation, i, j + 1) -
        evalStep(permutation, j, j + 1) : 0;
    return is + imlFactor + iplFactor + jmlFactor + jplFactor;
}
double evalSwap(int i, int j, Vector<int> const& permutation,
    INCREMENTAL_STATE const& is) const
{
    return updateIncrementalStateSwap(i, j, permutation, is) - is;
}
INCREMENTAL_STATE getIncrementalState(Vector<int> const& permutation) const
{
    return p(permutation);
}
double operator () (Vector<int> const& permutation) const
{
    return getIncrementalState(permutation);
}
};

```

В задаче о рюкзаке — в отличие от ЗК — для оценки переворота элемента из-за ограничения веса требуется дополнительное инкрементное состояние. Чтобы создать нужный градиент, надо добавить к лишнему весу такой штраф, чтобы свести на нет всю его прибыль:

```

template<typename PROBLEM> class KnapsackPenaltyProxy
{
    PROBLEM const& p;
    double profitLimit;
    double getScore(pair<double, double> const& is) const
    {
        // оценка полного решения в рамках задачи минимизации
        double score = -is.first, capacity = p.getCapacity();
        if(is.second > capacity) // простой штраф на случай
            // превышения вместимости
            score += profitLimit * exp((is.second - capacity)/capacity);
        return score;
    }
public:
    typedef pair<double, double> INCREMENTAL_STATE; // сперва идет прибыль
    KnapsackPenaltyProxy(PROBLEM const& theP): p(theP), profitLimit(0)
    {
        for(int i = 0; i < p.getSize(); ++i) profitLimit += p.getProfit(i);
    }
    INCREMENTAL_STATE updateIncrementalState(int iToFlip,
        Vector<bool> const& subset, INCREMENTAL_STATE const& is) const
    {
        bool selection = !subset[iToFlip];
    }
};

```

```

    return INCREMENTAL_STATE(is.first + p.getProfit(iToFlip) *
        (selection ? 1 : -1), is.second + p.getWeight(iToFlip) *
        (selection ? 1 : -1));
}
double evalStep(int iToFlip, Vector<bool> const& subset,
    INCREMENTAL_STATE const& is) const
{ // оценка разницы после одного шага
    return getScore(updateIncrementalState(iToFlip, subset, is)) -
        getScore(is);
}
INCREMENTAL_STATE getIncrementalState(Vector<bool> const& subset) const
{
    assert(subset.getSize() == p.getSize());
    double totalProfit = 0, totalWeight = 0;
    for(int i = 0; i < subset.getSize(); ++i) if(subset[i])
    {
        totalProfit += p.getProfit(i);
        totalWeight += p.getWeight(i);
    }
    return INCREMENTAL_STATE(totalProfit, totalWeight);
}
double operator() (Vector<bool> const& subset) const
{ return getScore(getIncrementalState(subset)); }
};

```

В задаче минимальной выполнимости произвести инкрементную оценку сложно, т. к. инкрементное состояние — это число истинных к текущему моменту частей выражения. Можно сопоставить переменной со списком предложений, содержащих ее:

```

template<typename PROBLEM> class SatisfiabilityProxy
{
    PROBLEM const& p;
    Vector<Vector<int> > varToClauseMap;
    bool evaluateClause(Vector<bool> const& subset, Vector<int> const& clause,
        int overrideI = -1) const
    {
        bool value = false;
        for(int j = 0; j < clause.getSize(); ++j)
        {
            int i = abs(clause[j]);
            bool valVar = subset[i];
            if(i == overrideI) valVar = !valVar; // переопределение означает переворот
            if(clause[j] < 0) valVar = !valVar; // обратное значение переменной
            value |= valVar; // удовлетворение есть объединение
                                // предложений "или" через "и"
        }
        return value;
    }
}
public:
    typedef int INCREMENTAL_STATE; // nSatisfied
    SatisfiabilityProxy(PROBLEM const& theP): p(theP),
        varToClauseMap(p.getSize())

```

```

{ // используется n предложений для n переменных
  for(int i = 0; i < p.getClauseSize(); ++i)
  {
    Vector<int> const& clause = p.getClause(i);
    for(int j = 0; j < clause.getSize(); ++j)
    { // у переменных есть знаки
      int variable = abs(clause[j]);
      varToClauseMap[variable].append(i);
    }
  }
}

INCREMENTAL_STATE updateIncrementalState(int iToFlip,
  Vector<bool> const& subset, INCREMENTAL_STATE const& is) const
{
  Vector<int> const& affectedClauses = varToClauseMap[iToFlip];
  int nSatisfiedOld = 0, nSatisfied = 0;
  for(int i = 0; i < affectedClauses.getSize(); ++i)
  {
    Vector<int> const& clause = p.getClause(affectedClauses[i]);
    nSatisfiedOld += evaluateClause(subset, clause);
    nSatisfied += evaluateClause(subset, clause, iToFlip);
  }
  return is + (nSatisfied - nSatisfiedOld);
}

double evalStep(int iToFlip, // оценка разницы после одного шага
  Vector<bool> const& subset, INCREMENTAL_STATE const& is) const
{ return -(updateIncrementalState(iToFlip, subset, is) - is); }

INCREMENTAL_STATE getIncrementalState(Vector<bool> const& subset) const
{
  assert(subset.getSize() == varToClauseMap.getSize());
  int nSatisfied = 0;
  for(int i = 0; i < p.getClauseSize(); ++i)
    nSatisfied += evaluateClause(subset, p.getClause(i));
  return nSatisfied;
}

double operator()(Vector<bool> const& subset) const
{ return -getIncrementalState(subset); }
};

```

В задаче об упаковке контейнера инкрементное состояние является текущей оценкой, как и в ЗК. Полученное разделение не обязательно должно соответствовать допустимой упаковке, поэтому можно ввести штраф за переполнение объема:

```

template<typename PROBLEM> class BinPackingProxy
{
  PROBLEM const& p;
  double getBinScore(Vector<int> const& bin) const
  { // оценка полного решения в рамках задачи минимизации
    if(bin.getSize() == 0) return 0;
    double sum = 0;
    for(int i = 0; i < bin.getSize(); ++i)
      sum += p.getWeight(bin[i]);
  }
};

```

```

    double score = 1, binSize = p.getBinSize();
    if(sum > binSize) // простой штраф на случай превышения вместимости
        score += exp((sum - binSize)/binSize);
    return score;
}

public:
    typedef double INCREMENTAL_STATE; // текущий балл
    BinPackingProxy(PROBLEM const& theP): p(theP) {}
    INCREMENTAL_STATE updateIncrementalState(int index, int from, int to,
        Vector<Vector<int> >const& bins, INCREMENTAL_STATE const& is)const
    {
        assert(index < bins[from].getSize());
        Vector<int> &bin1 = (Vector<int>&)bins[from],
            &bin2 = (Vector<int>&)bins[to];
        double partialScore = is - getBinScore(bin1) - getBinScore(bin2);
        // шаг
        bin2.append(bin1[index]);
        bin1[index] = bin1.lastItem();
        bin1.removeLast();
        double score = partialScore+ getBinScore(bin1) + getBinScore(bin2);
        // отмена шага
        bin1.append(bin2.lastItem());
        bin2.removeLast();
        swap(bin1[index], bin1.lastItem());
        return score;
    }
    double evalStep(int index, int from, int to, Vector<Vector<int> >const&
        bins, INCREMENTAL_STATE const& is)const
    {return updateIncrementalState(index, from, to, bins, is) - is;}
    INCREMENTAL_STATE getIncrementalState(Vector<Vector<int> >const& bins)
        const
    {
        double sum = 0;
        for(int i = 0; i < bins.getSize(); ++i) sum += getBinScore(bins[i]);
        return sum;
    }
    double operator() (Vector<Vector<int> > const& bins)const
    {return getIncrementalState(bins);}
};

```

## 16.10. Алгоритм имитации отжига

*Метаэвристики* — это стратегии глобального исследования. В частности, они позволяют алгоритму локального поиска избегать локальных оптимумов. Эвристика *имитации отжига* (Simulated Annealing, SA) работает со случайными ходами и принимает с некоторой вероятностью ухудшающие качество решения ходы. Эта вероятность изначально высокая, но уменьшается, пока не становится очень малой, что делает процесс эквивалентным локальному поиску. Случайный ход приводит поиск в часть ландшафта с глобальным оптимумом, а возможный локальный поиск находит его. Название алго-

ритма связано с физическим процессом отжига при охлаждении металла, который дает лучшую структуру при медленном опускании.

Изменение оценки  $\Delta f$  = предлагаемая оценка – текущая оценка, а «температура»  $T$  определяет вероятность принятия плохого шага, если  $e^{-\Delta f / T} > \text{uniform01}$ . Для плохого шага  $\Delta f > 0$ . Существует и более простое эквивалентное условие  $\Delta f < T \times \text{exponential01}$  ( $-\log(\text{равномерное изменение})$  – экспоненциальное изменение) — см. *разд. 6.15. Метод Монте-Карло*. Каждая итерация умножает  $T$  на «коэффициент охлаждения», уменьшая значение.

Таким образом, конечная температура  $T_{\text{last}} = T_0 \times \text{коэффициент охлаждения}^{n\text{Moves}}$ . Любые плохие ходы с  $\Delta f$  ниже заданного будут иметь очень высокую вероятность, а реализация позволяет управлять ими, считая  $\Delta f$  большей величиной, заданной вызывающей стороной:

```
template<typename PROBLEM> typename PROBLEM::X simulatedAnnealing(
    PROBLEM const& p, typename PROBLEM::X x, double T, double coolingFactor,
    double TCap, int maxMoves)
{
    assert(maxMoves > 0 && isfinite(T) && T > TCap && TCap >= 0 &&
        coolingFactor < 1 && coolingFactor > 0 && isfinite(p.getScore(x)));
    typedef pair<typename PROBLEM::MOVE, double> MOVE;
    for(; maxMoves--> 0; T *= coolingFactor)
    { // сравнение считается ложным, если m.second = NaN
        MOVE m = p.proposeMove(x);
        if(m.second <= 0 || max(m.second, TCap) < T *
            GlobalRNG().exponential(1)) p.applyMove(x, m.first);
    }
    return x;
}
```

Выбрать хорошие значения для параметров непросто. Обычно пользователю приходится пробовать несколько вариантов и выбирать тот, который лучше всего подходит для предметной области. Здесь можно использовать какой-либо вариант поиска по сетке или другой метод оптимизации (см. *главу 24. Численная оптимизация*). Можно также ввести процедуру самонастройки:

- ◆ предположим, что имеется начальное решение случайного качества, поэтому после случайного шага качество все еще будет случайным;
- ◆ изначально будем принимать  $\approx 90\%$  шагов. Грамотная стратегия состоит в том, чтобы потратить небольшую часть всегда принимаемых начальных ходов — например  $\sqrt{(n\text{Moves})}$ , чтобы оценить значение изменения 95-го процента. Если у вас есть информация о предметной области, можно вместо этого использовать максимально возможное изменение. Затем  $T_0$  вычисляется как:

$$\frac{\text{типичное значение}}{-\log(\text{желаемая вероятность})};$$

- ◆ единственным ограничением времени выполнения является число итераций. Обычно для поиска глобального оптимума требуется много итераций, поэтому задайте в качестве максимального количества столько, сколько позволяет терпение (неплохой вариант:  $10^6$ );

- ♦ установите коэффициент охлаждения таким, чтобы большинство последующих итераций выполняли не только локальный поиск. Нам не нужно, чтобы значение  $T$  слишком быстро приближалось к нулю, как бывает при выборе стандартного коэффициента охлаждения  $0,9999*$  (хотя для некоторых простеньких задач из  $1 - \frac{10}{nMoves}$  шагов такое значение подойдет). Выберите  $T_{cap}$  как изменение 5-го перцентиля от начальных случайных ходов. Если вам известно о минимально возможном изменении, превышающем ноль, используйте его. Система считается «замороженной», если вероятность принятия равна  $\frac{1}{nMoves}$ , а  $T_{last}$  задается в зависимости от нее. Тогда

$$\text{коэффициент охлаждения} = \left( \frac{T_0}{T_{last}} \right)^{\frac{1}{nMoves}}.$$

Такой подход является численно безопасным.

```
template<typename PROBLEM> typename PROBLEM::X
selfTunedSimulatedAnnealing(PROBLEM const& p, typename PROBLEM::X x,
int maxMoves = 1000000)
{
    assert(maxMoves >= 9); // минимальное значение, необходимое для оценки
    int nEstimate = int(sqrt(maxMoves)); // разумное значение
    typedef pair<typename PROBLEM::MOVE, double> MOVE;
    // оценка T
    Vector<double> values;
    for(int i = 0; i < nEstimate; ++i)
    { // выполнение случайных шагов для определения амплитуды изменений
        MOVE m = p.proposeMove(x);
        double change = abs(m.second);
        if(isfinite(change) && change > 0)
        {
            values.append(change);
            p.applyMove(x, m.first);
        }
    }
    maxMoves -= nEstimate;
    if(values.getSize() < 3) // задача не решается, нужен локальный поиск
        return localSearch(p, x, maxMoves, -1);
    double prFirst = 0.9, prLast = 1.0/maxMoves,
    deltaMax = quantile(values, 0.95), deltaMin = quantile(values, 0.05),
    T0 = -deltaMax/log(prFirst), TLast = -deltaMin/log(prLast),
    inverseRange = max(TLast/T0, numeric_limits<double>::min()),
    coolingFactor = pow(inverseRange, 1.0/maxMoves);
    return simulatedAnnealing(p, x, T0, coolingFactor, deltaMin, maxMoves);
}
```

Для задач перестановок, как 3К, имитацию отжига используют следующим образом:

```
template<typename INSTANCE> Vector<int>
solveSymmetricPermutationSimulatedAnnealingReverse(INSTANCE const& instance,
Vector<int> const& initial, int maxMoves)
```

```

{
    return selfTunedSimulatedAnnealing(SymmetricPermutationProblemReverseMove<
        INSTANCE>(instance), make_pair(initial, instance.getIncrementalState(
            initial)), maxMoves).first;
}
template<typename INSTANCE> Vector<int>
    solvePermutationSimulatedAnnealingSwap(INSTANCE const& instance,
        Vector<int> const& initial, int maxMoves)
{
    return selfTunedSimulatedAnnealing(PermutationProblemSwapMove<
        INSTANCE>(instance), make_pair(initial, instance.getIncrementalState(
            initial)), maxMoves).first;
}

```

Для задач подмножества — таких как задача об упаковке рюкзака и задача максимальной выполнимости — имитацию отжига реализуют следующим образом:

```

template<typename INSTANCE> Vector<bool> solveSubsetSimulatedAnnealing(
    INSTANCE const& instance, Vector<bool> const& initial, int maxMoves)
{
    return selfTunedSimulatedAnnealing(SubsetProblemFlipMove<INSTANCE>(
        instance), make_pair(initial, instance.getIncrementalState(initial)),
        maxMoves).first;
}

```

Для задач разделения — таких как упаковка контейнера — имитацию отжига используют следующим образом:

```

template<typename INSTANCE> Vector<Vector<int> >
    solvePartitionSimulatedAnnealing(INSTANCE const& instance,
        Vector<Vector<int> > const& initial, int maxMoves)
{
    return selfTunedSimulatedAnnealing(PartitionProblemSwapMove<INSTANCE>(
        instance), make_pair(initial, instance.getIncrementalState(initial)),
        maxMoves).first;
}

```

Некоторые эксперименты с самонастраивающей версией этого алгоритма показывают, что вероятность принятия и оценка решения уменьшаются с довольно разумной скоростью. В задачах, в которых возможное значение изменения всегда равно 1, возникает интересное явление — «непараметрический» отжиг, при котором плохие ходы делаются с некоторой вероятностью, независимо от  $\Delta f$ , а в таких задачах это и есть наилучшая стратегия. Для задач с различными значениями  $\Delta f$  это уже не так эффективно из-за возможных случайных слишком плохих ходов, но для ходов с качеством ниже  $T_{\text{ср}}$  такой подход неплох, если более полных знаний о задаче у пользователя нет. Таким образом, на первом этапе алгоритма выполняется хорошо настроенный симулированный отжиг, а на втором этапе, где настройка параметров затруднена, работает непараметрический отжиг.

Вопрос заключается в том, можно ли улучшить жестко закодированные числа и некоторые другие параметры. Использование 90%-ной вероятности первоначального принятия «типичного значения изменения» согласуется с выводами, приведенными в [16.16], и многими другими источниками. Значение  $-1/\log(0,9) \approx 9,5$  довольно велико, но более удачных стратегий в литературе, похоже, нет.



Часто советуют использовать максимальное и минимальное видимое значение. Когда диапазон значений в задаче очень велик, и большинство значений сконцентрировано вокруг более узкого поддиапазона, слишком много вычислений уйдет на слишком высокие или слишком низкие значения редких изменений. Но в случае самонастройки это разумно, особенно учитывая, что алгоритм имитации отжига не очень чувствителен к параметрам охлаждения.

Подстановка максимума и минимума в 95-й и 5-й квантили оправдана тем, что это крайние квантили, которые поддаются оценке с достаточно низкой дисперсией при выбранном числе начальных шагов. Заодно вы получаете небольшую защиту от слишком широких диапазонов значений. Но оценка в этом случае слишком зависит от начальной конфигурации, поэтому начальная температура может соответствовать только штрафным и недопустимым решениям. Однако есть ограничение температуры, которое спасает от этой проблемы. Существует незначительное допущение, которое состоит в том, что окрестность симметрична. В целом процедура самонастройки не может конкурировать с ручной настройкой, основанной на знании предметной области.

Мультипликативный коэффициент охлаждения используется чаще всего, но это не единственный вариант. Технически смоделированный отжиг допускает выполнение нескольких шагов для заданного значения  $T$ , что эквивалентно одному шагу с дискретным уменьшением  $T$ . При выборе правильной скорости работать будет любая функция, уменьшающая  $T$  с 1 до 0 или до некоторого небольшого значения. Можно ввести, например, линейное или полиномиальное уменьшение. Но только при экспоненциальном уменьшении посредством умножения вероятность принятия  $T_{\text{ср}}$  ведет себя двояко-экспоненциально — т. е. сначала выглядит линейной, а затем медленно уменьшается до целевого значения. Преимущество графиков полиномиального убывания, по-видимому, состоит в том, что они всегда доходят ровно до нуля, что избавляет от необходимости оценивать окончательное значение  $T_{\text{last}}$ . Но нам бы хотелось контролировать изменения  $T$  в разумном диапазоне, чтобы избежать слишком быстрого охлаждения и чрезмерных затрат времени.

Экспоненциальное распределение в этом случае сомнительно. Возможно, лучше взять распределение с толстым хвостом — например, распределение Леви. Алгоритму полезно небольшое количество шумов. Использование распределения с толстым хвостом сводит поведение алгоритма к непараметрическому отжигу, который редко допускает изменения на порядки больше, чем текущее  $T$ .

Хранение лучшего найденного решения дает мало выгоды, но занимает много времени выполнения при постепенном вычислении шагов.

Пожалуй, лучшее обоснование эффективности имитации отжига заключается в том, что алгоритм попадает в самую глубокую впадину, из которой не может выбраться (см. [16.13]). По сути, это похоже на табу-поиск (см. *разд. 16.20. Комментарии*). Исходя из приведенных ранее рассуждений о его различных вариантах и из того, что с момента его появления в 1983 году алгоритм был стабилен, приведенную реализацию можно считать окончательной. Во многих случаях, особенно, когда доступное пользователю время ограничено, разумно сделать так, чтобы имитация отжига была и первой, и последней процедурой, испробованной на этой задаче.

Теория имитации отжига несколько своеобразна. Подробнее о ней можно почитать в работе [16.17], а доказательства содержатся в [16.8]. Там приведен график, иллюстри-

рующий, что алгоритм гарантированно сходится, но количество рассмотренных решений превышает количество существующих. Для типичных реализаций с мультипликативным снижением температуры теоретической базы нет, зато есть практический опыт. Как и любой другой метаэвристический алгоритм, это, скорее, алгоритм аппроксимации без гарантий производительности.

Оригинальная версия алгоритма имитации отжига основывалась на сходимости простого математического процесса: при каждой температуре есть случайное блуждание, которое бесконечно перемещается по пространству состояний, что дает некоторую вероятность посещения того или иного решения. По мере уменьшения  $T$  вероятности концентрируются вокруг набора глобальных минимумов. При ограниченном количестве ходов для каждого значения  $T$  и медленном уменьшении вычислительный процесс достаточно похож на математический. Но выполнять многократные проверки одного и того же решения очень расточительно, поэтому поведение вычислительного процесса отличается от модельного.

Для математического процесса любое значение  $T > 0$  работает одинаково, но для вычислительного процесса запуск с высоким  $T$  попросту необходим. Современная версия алгоритма с одним шагом при каждом значении температуры, но с более медленным снижением более устойчива. Если предположить, что смоделированный отжиг не чувствителен к какой-либо полосе  $\varepsilon$  вокруг текущей  $T$  и что уменьшение происходит достаточно медленно, то обе версии будут работать одинаково. Математику для этого алгоритма следует считать, скорее, рекомендацией, а не набором четких указаний.

## 16.11. Повторный локальный поиск

Имитация отжига больше всего подходит для случайных шагов, а алгоритм *повторного локального поиска* (Iterated Local Search, ILS) — для лучшего или первого улучшения. Алгоритм итеративно находит локальный минимум и *прыгает* далеко в ландшафт, сохраняя лучшее найденное решение. Этот прыжок должен быть таким, чтобы локальный поиск не мог слишком легко его отменить. Можно выполнить *частичный перезапуск*, не теряя всех текущих компонентов решения, и это решение обычно лучше, чем *решение со случайным перезапуском*, которое, как правило, дает среднее качество — это называют *катастрофой центрального предела*. Все метаэвристики можно рассматривать как попытки направить локальный поиск в нужную сторону при случайном перезапуске. Таким образом, случайный перезапуск тоже можно использовать.

По умолчанию следует установить максимальное количество прыжков равным 100, но можно задать и больше, если терпение позволяет:

```
template<typename PROBLEM> typename PROBLEM::X iteratedLocalSearch(
    PROBLEM& p, typename PROBLEM::X x, int maxBigMoves)
{
    typename PROBLEM::X bestX = x;
    double bestScore = p.getScore(bestX);
    while(maxBigMoves-->0)
    {
        x = p.localSearchBest(x);
        // обновление лучшего значения
        double xScore = p.getScore(x);
```

```

        if(xScore < bestScore)
        {
            bestScore = xScore;
            bestX = x;
        }
        p.bigMove(x);
    }
    return bestX;
}

```

ILS лучше всего подходит в случаях, когда локальный поиск выполняется специализированным алгоритмом, далеким от обычного локального поиска. Пользователю может потребоваться провести немало исследований, чтобы правильно выбрать направление прыжка. Выполнение в качестве перезапуска пары случайных шагов приводит только к тому, что общий алгоритм оказывается хуже, чем имитация отжига. Таким образом, для простоты и наглядности в приведенных далее реализациях для конкретных представлений используется случайный перезапуск.

Для симметричных задач перестановки:

```

template<typename EVALUATOR> struct SymmetricPermutationILSFromRandReverseMove
{
    EVALUATOR const& e;
    typedef Vector<int> X;
    int lsMoves;
    SymmetricPermutationILSFromRandReverseMove(EVALUATOR const& theE,
        int lsMaxMoves): lsMoves(lsMaxMoves), e(theE) {}
    X localSearchBest(X const& x)
    {return solveSymmetricPermutationLocalSearchReverse(e, x, lsMoves);}
    double getScore(X const& x){return e(x);}
    void bigMove(X& x)
    {GlobalRNG().randomPermutation(x.getArray(), x.getSize());}
};

template<typename INSTANCE>
Vector<int> solveSymmetricPermutationIteratedLocalSearch(INSTANCE const&
    instance, Vector<int> const& initial, int lsMaxMoves, int bigMoves)
{
    SymmetricPermutationILSFromRandReverseMove<INSTANCE> move(instance,
        lsMaxMoves/bigMoves);
    return iteratedLocalSearch(move, initial, bigMoves);
}

```

Для задач подмножества:

```

template<typename EVALUATOR> struct SubsetILSFromRandFlipMove
{
    EVALUATOR const& e;
    typedef Vector<bool> X;
    int lsMoves;
    SubsetILSFromRandFlipMove(EVALUATOR const& theE, int lsMaxMoves):
        lsMoves(lsMaxMoves), e(theE) {}
    X localSearchBest(X const& x)
    {return solveSubsetLocalSearchFlip(e, x, lsMoves);}
}

```

```

double getScore(X const& x){return e(x);}
void bigMove(X& x) // простой перезапуск
    (for(int i = 0; i < x.getSize(); ++i) x[i] = GlobalRNG().mod(2));
};
template<typename INSTANCE> Vector<bool> solveSubsetIteratedLocalSearch(
    INSTANCE const& instance, Vector<bool> const& initial, int lsMaxMoves,
    int bigMoves)
{
    SubsetILSFromRandFlipMove<INSTANCE> move(instance, lsMaxMoves/bigMoves);
    return iteratedLocalSearch(move, initial, bigMoves);
}

```

Вы также можете ввести критерии приемлемости для отклонения слишком уж некачественных прыжков, но в этом мало смысла, потому что даже некачественные результаты являются частью их назначения.

## 16.12. Генетические алгоритмы

Генетические алгоритмы просты и популярны, т. к. уже давно описаны в доступной литературе. Однако многие исследования, включая мои собственные, показывают, что методы локального поиска с одним решением более масштабируемы, меньше требуют от пользователя и для типичных комбинаторных задач дают лучшие решения при том же вычислительном бюджете. Вам следует знать об этом, прежде чем тратить время на этот подход (хотя для вашей задачи он может оказаться идеален). В задачах непрерывной оптимизации (см. главу 24. *Численная оптимизация*) инкрементальная эффективность случайных локальных скачков исчезает, а генетические алгоритмы оказываются полезны. Кроме того, они хороши в задачах многокритериальной оптимизации (см. далее в этой главе).

Алгоритм хранит набор решений и итеративно обновляет его путем:

- ◆ *выбора* лучшего решения;
- ◆ *скрещивания* пар решений;
- ◆ *мутации* отдельных решений.

Алгоритм вдохновлен процессом биологической эволюции. В задачах вроде упаковки рюкзака, где для представления решения используются подмножества, шаги оказываются простыми, а в ЗК все сложнее, т. к. решение представляется в виде перестановки (обратите внимание на работу [16.5]). Общий рецепт таков:

- ◆ отбор — сначала нужно выбрать лучшее решение. Затем выполняется *турнирный отбор* — для каждого оставшегося места выбирается лучшее из двух случайно выбранных решений;
- ◆ скрещивание — реализация зависит от представления, но для подмножеств и многомерных векторов чего-либо можно выполнить *равномерное скрещивание*. Выберите какой-нибудь индекс  $i$ . Тогда для первого «потомка» берем первую часть до  $i$  от «родителя» 1, а вторую часть — от «родителя» 2. Для второго «потомка» — наоборот. В результате «потомки» должны быть похожи на «родителей» — т. е. хорошие свойства последних в основном сохраняются, как и при прыжках в алгоритме ILS;

- ♦ мутация — в современных генетических алгоритмах вместо мутации используют локальный поиск с некоторой небольшой вероятностью. Получившийся таким образом алгоритм называется *генетическим локальным поиском*. Иногда также используются названия *генетический гибрид* и *меметический алгоритм*.

Представленная реализация исходит из того, что вызывающая сторона передает операторы для скрещивания и локального поиска:

```
template<typename PROBLEM> pair<typename PROBLEM::X, double>
geneticLocalSearch(PROBLEM const& p, int populationSize, int nLocalMoves,
int maxEvals)
{
    assert(maxEvals > populationSize && populationSize > 1);
    int n = 2 * (populationSize/2) + 1; // должно быть нечетным,
        // в идеале первое, остальные эволюционируют попарно
    Vector<pair<double, typename PROBLEM::X> > population(n),
        populationNew(n);
    for(int i = 0; i < n; ++i) population[i].first =
        p.evaluate(population[i].second = p.generate());
    PairFirstComparator<double, typename PROBLEM::X> c;
    while((maxEvals -= (n - 1) * nLocalMoves) >= 0)
    { // сохранение лучшего решения
        populationNew[0] = population[argMin(population.getArray(), n, c)];
        // турнирный отбор
        for(int i = 1; i < n; ++i)
        {
            int j = GlobalRNG().mod(n), k = GlobalRNG().mod(n), winner =
                c(population[j], population[k]) ? j : k;
            populationNew[i] = population[winner];
        }
        for(int i = 1; i + 1 < n; i += 2)
        { // скрещивание родителей, получение потомков
            p.crossover(populationNew[i].second, populationNew[i + 1].second);
            // локальный поиск и оценка
            for(int j = 0; j < 2; ++j)
            {
                populationNew[i + j].second =
                    p.localSearch(populationNew[i + j].second, nLocalMoves);
                populationNew[i + j].first =
                    p.evaluate(populationNew[i + j].second);
            }
        }
        population = populationNew;
    }
    int best = argMin(population.getArray(), n, c);
    return make_pair(population[best].second, population[best].first);
}
```

Чтобы использовать этот алгоритм в задачах подмножества, например в задаче об упаковке рюкзака, нужно указать операторы и решить, как распределить оценки. Разумная стратегия такова:

- ◆ если размер экземпляра  $n$  велик, а инкрементные ходы выполняются за  $O(1)$ , используйте  $O(n)$  перемещений локального поиска — например,  $n/5$ , чтобы затраты времени были примерно равны затратам на генерирование и оценку выборки;
- ◆ если  $n$  мало, или локальные ходы не дают эффективную инкрементную оценку, используйте  $nMoves^{1/3}$  локальных ходов;
- ◆ размер популяции равен  $\sqrt{\frac{nMoves}{nLocalMoves}}$ . То же самое справедливо для числа итераций:

```
template<typename FUNCTION> class GLSSubset
{
    FUNCTION const& f;
    int n;
public:
    typedef Vector<bool> X;
    GLSSubset(FUNCTION const& theF, int theN): f(theF), n(theN) {}
    X generate() const
    {
        Vector<bool> subset(n);
        for(int i = 0; i < n; ++i) subset[i] = GlobalRNG().mod(2);
        return subset;
    }
    void crossover(X& x1, X& x2) const
    { // равномерное скрещивание
        assert(x1.getSize() == x2.getSize());
        for(int k = 0; k < x1.getSize(); ++k) if(GlobalRNG().mod(2))
            swap(x1[k], x2[k]);
    }
    X localSearch(X const& x, int nLocalMoves) const
    {return solveSubsetLocalSearchFlip(f, x, nLocalMoves);}
    double evaluate(X const& x) const{return f(x);}
};

template<typename FUNCTION> Vector<bool>
geneticLocalSearchSubset(FUNCTION const& f, int n,
int maxEvals = 10000000)
{
    assert(maxEvals >= n); // простая проверка разумности,
                          // требуется 4 * nLocalMoves
    // в зависимости от стоимости инкрементного шага
    // берется то же поколение
    int nLocalMoves = max(n/5, int(pow(maxEvals, 1.0/3))),
        populationSize = int(sqrt(maxEvals * 1.0/nLocalMoves));
    // одно поколение + эквивалент оценки
    return geneticLocalSearch(GLSSubset<FUNCTION>(f, n), populationSize,
        nLocalMoves, maxEvals).first;
}
```

В задачах перестановки скрещивание выполнять сложнее, потому что в конечном итоге нужно получить корректные дочерние перестановки. Один из простых вариантов — *упорядоченное скрещивание*: при заданных  $x_1$  и  $x_2$  вторую половину элементов  $x_1$  рас-

положить в порядке  $x_2$ . Аналогичным образом расположить первую половину элементов  $x_2$  в порядке  $x_1$ . Это базовая версия с начальной позицией 0 и длиной  $n/2$ . Более общий вариант — сделать эти параметры случайными, как для равномерного скрещивания. Перестановки не зависят от того, с какого элемента они отсчитываются, поэтому начальный индекс можно использовать любой:

```
template<typename FUNCTION> class GLSPermutation
{
    FUNCTION const& f;
    int n;
    bool isSymmetric;
public:
    typedef Vector<int> X;
    GLSPermutation(FUNCTION const& theF, int theN, bool theIsSymmetric):
        f(theF), n(theN), isSymmetric(theIsSymmetric)
        {assert(theN >= 4);} // чтобы скрещивание работало, значение должно
                            // быть не менее 4
    X generate() const
    {
        X p(n);
        for(int i = 0; i < n; ++i) p[i] = i;
        GlobalRNG().randomPermutation(p.getArray(), n);
        return p;
    }
    void crossover(X& x1, X& x2) const
    { // упорядоченное скрещивание по длине [2, n-2] из
      // случайной начальной точки
      assert(x1.getSize() == n && x2.getSize() == n);
      int start = GlobalRNG().mod(n), length = 1 + GlobalRNG().mod(n - 3);
      Vector<bool> x1FromX2(n, false), x2FromX1(n, false);
      for(int i = 0; i < n; ++i)
      { // в x1 кладется вторая часть из x2, а в x2 - начало из x1
        int j = (start + i) % n; // неявное изменение стартовой точки
        if(i < length) x2FromX1[x2[j]] = true;
        else x1FromX2[x1[j]] = true;
      }
      X x1Copy = x1; // временное копирование для защиты от перезаписи
      for(int i = 0, i1 = (start + length) % n; i < n; ++i)
      { // заполняем вторую часть x1 из x2 в том же порядке
        int element = x2[(start + i) % n]; // перебор x1
                                          // от стартовой точки
        if(x1FromX2[element])
        {
            x1[i1] = element;
            i1 = (i1 + 1) % n; // продвижение
        }
      }
      for(int i = 0, i2 = start; i < n; ++i) // перебор x2
                                          // от стартовой точки
      { // заполнение второй части x2 из x1 в том же порядке
        int element = x1Copy[(start + i) % n];
```

```

        if(x2FromX1[element])
        {
            x2[i2] = element;
            i2 = (i2 + 1) % n; // продвижение
        }
    }
}

X localSearch(X const& x, int nLocalMoves) const
{ // для симметричных задач используется обратный ход
    return isSymmetric ? solveSymmetricPermutationLocalSearchReverse(f, x,
        nLocalMoves) : solvePermutationLocalSearchSwap(f, x, nLocalMoves);
}

double evaluate(X const& x) const {return f(x);}
};

template<typename FUNCTION> Vector<int>
geneticLocalSearchPermutation(FUNCTION const& f, int n, bool isSymmetric,
    int maxEvals = 10000000)
{
    assert(maxEvals >= n); // простая проверка разумности,
                          // требуется 4 * nLocalMoves
    // в зависимости от стоимости инкрементного шага
    // берется то же поколение
    int nLocalMoves = max(n/5, int(pow(maxEvals, 1.0/3))),
        populationSize = int(sqrt(maxEvals * 1.0/nLocalMoves));
    // одно поколение + эквивалент оценки
    return geneticLocalSearch(GLSPermutation<FUNCTION>(f, n, isSymmetric),
        populationSize, nLocalMoves, maxEvals).first;
}

```

Для задач разделения фокус заключается в преобразовании списка контейнеров в форму, допускающую скрещивание:

```

template<typename FUNCTION> class GLSPartition
{
    FUNCTION const& f;
    int n;
    void removeEmptyBins(Vector<Vector<int> >& partition) const
    {
        for(int i = n - 1; i >= 0; --i) if(partition[i].getSize() == 0)
        {
            partition[i] = partition.lastItem();
            partition.removeLast();
        }
    }
}

public:
    typedef Vector<Vector<int> > X;
    GLSPartition(FUNCTION const& theF, int theN): f(theF), n(theN) {}
    X generate() const
    { // случайное присвоение
        Vector<Vector<int> > partition(n);
        for(int i = 0; i < n; ++i) partition[GlobalRNG().mod(n)].append(i);
        removeEmptyBins(partition);
    }
}

```



```

    return partition;
}
void crossover(X& x1, X& x2) const
{ // преобразование вида
    X* xs[2] = {&x1, &x2}; // с++ не поддерживает ссылочные массивы
    Vector<int> assignments[2] = {Vector<int>(n, -1), Vector<int>(n, -1)};
    for(int k = 0; k < 2; ++k)
    {
        X& x = *xs[k];
        for(int bin = 0; bin < x.getSize(); ++bin)
            for(int j = 0; j < x[bin].getSize(); ++j)
            {
                int item = x[bin][j];
                assert(item >= 0 && item < n);
                assignments[k][item] = bin;
            }
    } // равномерное скрещивание
    for(int item = 0; item < n; ++item)
    { // проверка некорректных входных данных - каждый элемент должен
        // быть присвоен
        assert(assignments[0][item] != -1 && assignments[1][item] != -1);
        if(GlobalRNG().mod(2))
            swap(assignments[0][item], assignments[1][item]);
    } // преобразование в вектор контейнеров
    for(int k = 0; k < 2; ++k)
    {
        X& x = *xs[k];
        x = Vector<Vector<int>> >(n);
        for(int item = 0; item < n; ++item)
            x[assignments[k][item]].append(item);
        removeEmptyBins(x);
    }
}
X localSearch(X const& x, int nLocalMoves) const
{ return solvePartitionLocalSearchSwap(f, x, nLocalMoves); }
double evaluate(X const& x) const { return f(x); }
};

template<typename FUNCTION> Vector<Vector<int>> >
geneticLocalSearchPartition(FUNCTION const& f, int n,
int maxEvals = 10000000)
{
    assert(maxEvals >= n); // простая проверка разумности,
        // требуется 4 * nLocalMoves
    // в зависимости от стоимости инкрементного шага берется то же поколение
    int nLocalMoves = max(n/5, int(pow(maxEvals, 1.0/3))),
        populationSize = int(sqrt(maxEvals * 1.0/nLocalMoves));
    // одно поколение + эквивалент оценки
    return geneticLocalSearch(GLSPartition<FUNCTION>(f, n), populationSize,
        nLocalMoves, maxEvals).first;
}

```

В отличие от алгоритмов с одиночным решением, успех генетических алгоритмов малопонятен. Видимо, дело в том, что используемый в них процесс аналогичен парал-

лельному ILS. Например, в задачах подмножества порядок индексов не имеет значения, поэтому равномерное скрещивание — это перестановка индексов в некотором порядке и выполнение *одноточечного скрещивания* (перемена местами первой и второй частей родителей). В задачах перестановки только начальная точка и длина являются случайными, а в остальном используется *одноточечное скрещивание*. В результате получается разнообразная популяция и достигается хорошее качество алгоритма. В целом скрещивание аналогично скачкам ILS, но оно чуть более контролируемо и оптимизирует эффективность.

В рамках алгоритма нужно поддерживать разнообразие популяции, иначе он *сойдется* к единому решению. Турнирный отбор для этой задачи работает лучше всего (см. [16.5]). При реализации мутации локальным поиском схождение происходит с вероятностью 1, при этом на одну мутацию приходится много локальных перемещений, а обработка большой популяции будет невозможна из-за ограничений вычислительных ресурсов.

## 16.13. Анализ производительности

Цель приведенного здесь анализа состоит в том, чтобы проиллюстрировать применение к некоторой задаче некоторых общих алгоритмов, а не специализированных современных алгоритмов. Для каждой задачи используются только некоторые из решателей. Одни задачи очень просты по сравнению с другими, и для них все алгоритмы работают примерно одинаково. В сравнении используются представленные генераторы случайных экземпляров и выполняется только один запуск, поэтому статистические выводы сделать невозможно. Все алгоритмы запускаются на одном и том же экземпляре. Решения ILS, если таковые есть, служат лишь в качестве основы для случайного перезапуска локального поиска.

Задача коммивояжера не слишком сложна по сравнению с комбинаторным пространством, и имитация отжига с обратным шагом (STSA Reversals) здесь явный фаворит (рис. 16.6).

Укладка рюкзака — простая задача, — жадный алгоритм проигрывает только B&B, и то очень незначительно (рис. 16.7).

Problem Size	100
Expected	7.07
Initial	50.01
LS Reversals	8.23
STSA Reversals	7.67
GE Reversals	9.79
ILS Reversals	7.86
LS Swaps	17.31
STSA Swaps	15.79
B&B	8.50
RTAS	8.57

Рис. 16.6. Результаты для ЗК

Problem Size	1000
Initial	-235.06
Greedy	-405.45
LS Flips	-245.42
STSA Flips	-399.12
Genetic	-397.47
ILS	-271.29
B&B	-405.49

Рис. 16.7. Результаты для задачи об укладке рюкзака

Для задачи выполнимости с тремя переменными (3-SAT) сгенерированные экземпляры кажутся простыми. Все метаэвристики дают примерно одинаковую оценку (рис. 16.8).

Упаковка контейнеров — это простая задача, и жадный алгоритм аппроксимации справляется с ней лучше всего (рис. 16.9).

Problem Size	100 1000
Initial	-861
LS Flips	-967
STSA Flips	-970
Genetic	-971
ILS	-970

Рис. 16.8. Результаты для задачи 3-SAT

Problem Size	1000
Initial	1000.00
Greedy	104.00
LS Swaps	109.00
STSA Swaps	111.00
Genetic	124.08

Рис. 16.9. Результаты для контейнерной упаковки. Генетическое решение сработало плохо, но это поправимо

Генетический локальный поиск показал себя в задаче с рюкзаком и в 3К хуже, чем имитация отжига, и примерно столь же эффективен в задаче 3-SAT и упаковке контейнера при одинаковом количестве локальных перемещений. Но даже в этом случае и в целом генетические алгоритмы потенциально намного медленнее из-за невозможности выполнять пошаговую оценку. Бюджет алгоритмов локального поиска учитывает добавочные оценки, а бюджет других алгоритмов учитывает стандартные оценки, что не совсем корректно.

## 16.14. Подготовка данных для конкретной задачи

При решении некоторых конкретных задач есть возможность выполнить анализ и уменьшить ее размер, используя присущие ей свойства — например, симметричность. Но для этого требуется понимать задачу, и автоматически такую подготовку выполнить нельзя. Например, при наличии конкретного *расписания поездов* можно объединить общие маршруты или станции. Иногда использование специфичной для задачи информации позволяет уменьшить объем задачи до уровня, достаточного для использования алгоритма грубой силы (см. [16.19]).

## 16.15. Многоцелевая оптимизация

Пусть требуется найти решение, которое оптимизирует сразу несколько целей. Например, вы хотите свести к минимуму время путешествия к месту назначения, расход топлива и вероятность получения штрафа за рулем. Тот, кто принимает решение, может захотеть рассмотреть несколько вариантов и выбрать один из них.

Любое решение, которое по всем целям не хуже другого, является оптимальным, если только оно не строго хуже по одним целям и не одинаково по другим. Оптимальные решения образуют потенциально бесконечный *Парето-оптимальный фронт*. Управлять им сложнее, чем одним решением. Лучше всего оптимизировать какую-либо функцию полезности — например, линейную комбинацию желаемых целей с заданными весами, получая в результате одну цель. В работе [16.16] описаны многие другие

методы и концепции — в частности, алгоритмы для работы со всеми задачами одновременно.

## 16.16. Обработка ограничений

В некоторых задачах оптимизации решения состоят из переменных, значения которых принадлежат некоторой области целостности. *Ограничением* называют набор запрещенных кортежей значений. Например, решение sudoku состоит из 81 переменной в диапазоне от 1 до 9, и каждое поле, строка и столбец должны содержать неповторяющиеся цифры. Решение состоит в присваивании значений, не нарушающих заданные ограничения. Если такого решения не существует, то задача считается *неразрешимой*. Ограничения из двух переменных являются *бинарными*, и они наиболее эффективны с точки зрения представления и работы.

Самый простой способ представить переменные и допустимые значения — это вектор наборов битов. Каждая переменная связана с набором битов, размер которого равен ее диапазону, и бит  $i$  устанавливается, если разрешено  $i$ -е значение в домене. Самый простой способ представить бинарные ограничения — создать неориентированный граф, где вершины соответствуют переменным, а ребра — ограничениям между задействованными вершинами. Данные ребер содержат ограничивающий функтор, который проверяет, разрешено ли определенное значение:

```
template<typename CONSTRAINT> struct ConstraintGraph
{
    typedef GraphAA<CONSTRAINT> GRAPH;
    GRAPH g;
    Vector<Bitset<>> variables;
    void addVariable(int domain)
    {
        g.addVertex();
        variables.append(Bitset<>(domain));
    }
    void addConstraint(int v1, int v2, CONSTRAINT const& constraint)
    {
        assert(v1 != v2);
        g.addUndirectedEdge(v1, v2, constraint);
    }
    void disallow(int variable, int value)
    {variables[variable].set(value, false);}
    bool hasSolution(int variable){return !variables[variable].isZero();}
};
```

Значение разрешено, если любая другая переменная может принимать  $\geq 1$  значений таких, что пара значений не нарушает ограничения между ними:

```
bool isAllowed(int variable, int value, int otherVariable,
    CONSTRAINT const& constraint)
{
    for(int i = 0; i < variables[otherVariable].getSize(); ++i)
        if(variables[otherVariable][i] && constraint.isAllowed(variable,
            value, otherVariable, i)) return true;
```

```

    return false;
}

```

Чтобы проверить переменную по сравнению с другой и *пересмотреть* ее возможный диапазон, удалив запрещенные значения, нужно проверить каждое установленное значение. Пусть  $d$  = максимальный размер домена любой переменной. Тогда время выполнения проверки составляет  $O(d^2)$ :

```

bool revise(int variable, int otherVariable, CONSTRAINT const& constraint)
{
    bool changed = false;
    for(int i = 0; i < variables[variable].getSize(); ++i)
        if(variables[variable][i] &&
            !isAllowed(variable, i, otherVariable, constraint))
        {
            disallow(variable, i);
            changed = true;
        }
    return changed;
}

```

Проверка каждой переменной по сравнению с любой другой переменной, с которой у нее есть ограничение, выполняется до тех пор, пока прекращение внесения изменений не даст правильное окончательное решение. Упрощенный вариант AC3 (SAC3) делает это более эффективно, используя тот факт, что после пересмотра переменная не может быть изменена до тех пор, пока не будут изменены ее соседи:

1. Пересмотреть каждую переменную по отношению к каждому ее соседу и поставить ее в очередь.
2. Пока очередь не опустеет:
3. Удалить переменную из очереди.
4. Для любой соседней переменной:
5.       Пересмотреть ее по отношению к рассматриваемой переменной.
6.       Поставить в очередь, если диапазон пересмотрен.

Вместо очереди можно использовать любую списочную структуру данных. Пусть  $n$  — количество переменных, а  $c$  — количество ограничений. На каждом проходе делаются ожидаемые  $c/n$  и в худшем случае  $n$  пересмотров. Поскольку каждую переменную можно пересматривать  $\leq d$  раз, время выполнения алгоритма в наихудшем случае равно  $O(n^2 d^3)$ . В лучшем случае время равно  $O(cd^2)$ , когда шаг (2) не требуется, а среднее значение находится между ними, или если число проходов очереди равно  $O(1)$ :

```

bool SAC3Helper(int v, Queue<int>& q, Vector<bool>& onQ, bool isFirstPass)
{
    onQ[v] = false;
    for(typename GRAPH::AdjacencyIterator i = g.begin(v);
        i != g.end(v); ++i)
    { // на первом проходе пересматриваем переменные,
      // а в последующих проходах — соседей
        int revisee = i.to(), against = v;
        if(isFirstPass) swap(revisee, against);
    }
}

```

```

        if(revise(revisee, against, i.data()))
        {
            if(!hasSolution(revisee)) return false; // задача не решается
            if(!onQ[revisee]) q.push(revisee);
            onQ[revisee] = true;
        }
    }
    return true;
}

bool SAC3()
{
    Queue<int> q;
    Vector<bool> onQ(g.nVertices(), true);
    for(int j = 0; j < g.nVertices(); ++j)
        if(!SAC3Helper(j, q, onQ, true)) return false;
    while(!q.isEmpty())
        if(!SAC3Helper(q.pop(), q, onQ, false)) return false;
    return true;
}

```

Вы можете создать двоичный граф ограничений более высокого порядка, если будете проверять только одну переменную по отношению к остальным. Например, *ограничение* AllDifferent требует, чтобы в некотором подмножестве переменных у всех были разные значения. Это позволяет, например, моделировать sudoku как задачу с бинарными ограничениями:

```

struct AllDifferent
{
    LinearProbingHashTable<int, bool> variables;
    void addVariable(int variable){variables.insert(variable, true);}
    struct Handle
    {
        LinearProbingHashTable<int, bool>& variables;
        bool isAllowed(int variable, int value, int variable2, int value2)
        const
        {
            if(variables.find(variable) && variables.find(variable2))
                return value != value2;
            return true;
        }
        Handle(LinearProbingHashTable<int, bool>& theVariables):
            variables(theVariables) {}
    } handle;
    AllDifferent(): handle(variables) {}
};

```

Тогда sudoku моделируется следующим образом:

```

struct Sudoku
{
    AllDifferent ad[3][9];
    ConstraintGraph<AllDifferent::Handle> cg;
}

```

```

Sudoku(int values[81])
{
    for(int i = 0; i < 81; ++i)
    {
        cg.addVariable(9);
        if(values[i])
        {
            cg.variables[i].setAll(false);
            cg.variables[i].set(values[i] - 1, true);
        }
        else cg.variables[i].setAll(true);
    }
    for(int i = 0; i < 9; ++i)
    {
        int rowStart = i * 9, columnStart = i,
            boxStart = i/3 * 27 + (i % 3) * 3;
        for(int j = 0; j < 9; ++j)
        {
            int rowMember = rowStart + j;
            int columnMember = columnStart + j*9;
            int boxMember = boxStart + j/3 * 9 + j % 3;
            ad[0][i].addVariable(rowMember);
            ad[1][i].addVariable(columnMember);
            ad[2][i].addVariable(boxMember);
            if(j == 8) continue;
            for(int k = j+1; k < 9; ++k)
            {
                int boxMember2 = boxStart + k/3 * 9 + k % 3;
                cg.addConstraint(rowMember, rowStart + k, ad[0][i].handle);
                cg.addConstraint(columnMember, columnStart + k * 9,
                    ad[1][i].handle);
                cg.addConstraint(boxMember, boxMember2, ad[2][i].handle);
            }
        }
    }
    cg.SAC3();
}

void printSolution()const
{
    for(int i = 0; i < 81; ++i)
    {
        if(i % 9 == 0) cout << '\n';
        int count = 0, value = -1;
        for(int j = 0; j < 9; ++j)
        {
            if(cg.variables[i][j])
            {
                ++count;
                value = j + 1;
            }
        }
    }
}

```

```

        if(count > 1) cout << "x";
        else cout << value;
    }
    cout << endl;
}
};

```

Алгоритм SAC3 правильно исключает значения, которые нарушают бинарные ограничения, но не те, которые не нарушают ограничения кортежа более высокого порядка. В этом случае можно попробовать алгоритм В&В или поиск с возвратом, в котором SAC3 используется для проверки присвоения текущего значения, уменьшения диапазонов еще не выбранных переменных и выбора следующей переменной как переменной с наименьшим диапазоном для максимального сокращения. Например, чистый SAC3 может решить простую головоломку sudoku, но застревает на средней и сложной головоломке (рис. 16.10).

```

int easy() =
{
    2,0,1,7,9,5,0,0,0,
    0,0,9,0,0,0,0,1,0,
    0,0,0,3,0,1,0,0,7,
    0,2,0,5,0,0,1,7,8,
    0,8,0,0,0,0,0,9,0,
    1,5,7,0,0,4,0,3,0,
    6,0,0,8,0,2,0,0,0,
    0,9,0,6,0,0,5,0,0,
    0,0,0,1,7,9,3,0,6
};

int medium() =
{
    0,0,5,0,0,3,2,9,0,
    9,0,0,2,0,0,0,3,4,
    0,0,0,0,1,0,5,0,0,
    0,0,0,0,9,0,0,7,1,
    0,0,0,5,0,5,0,0,0,
    7,3,0,0,2,0,0,0,0,
    0,0,7,0,6,0,0,0,0,
    6,8,0,0,0,9,0,0,2,
    0,5,2,3,0,0,6,0,0
};

int hard() =
{
    0,1,2,0,6,0,8,0,0,
    0,0,0,0,3,0,0,5,0,
    6,0,0,4,0,0,0,0,7,
    0,0,6,0,0,0,0,1,0,
    0,9,7,0,0,0,6,4,0,
    0,8,0,0,0,0,7,0,0,
    8,9,0,0,0,1,0,0,3,
    0,4,0,0,5,0,0,0,0,
    0,0,1,0,2,0,9,7,0
};

```

```

"Easy Solution" Easy Solution
231795864
579468213
468321957
924536178
386217495
157984632
613852749
792643581
845179326
"Medium Solution" Medium Solution
xx5xx329x
9xx2xxx34
xxxx1x8xx
xxxx9xx71
xxx6x5xxx
73xx2xxxx
xx7x6xxxx
68xxx3xx2
x528xx6xx
"Hard Solution" Hard Solution
x12x6x8xx
x7xx3xx5x
6xx4xxxx7
xx6xxxx1x
x97xxx64x
x8xxxx7xx
8xxxx1xx3
x4xx5xxxx
xx1x2x97x
Process returned 0 (0x0)   execut
Press any key to continue.

```

Рис. 16.10. Решения sudoku разной сложности алгоритмом SAC3



## 16.17. Стохастические задачи

В задачах, где целевая функция зашумлена и нам нужно найти решение с лучшим  $E[\text{качество}]$ , эффективно использовать методы аппроксимации среднего/пути (см. главу 24. Численная оптимизация).

Приведенные здесь алгоритмы не предназначены для работы с зашумленными данными, хотя некоторые из них хорошо решают эту проблему. Например, в алгоритме имитации отжига новое решение сравнивается с предыдущим за одно сравнение, что технически недопустимо из-за шума, но при небольших уровнях шума такое сравнение использовать можно. В более крупных задачах локальный поиск по сути представляет собой случайное блуждание.

## 16.18. Общие рекомендации

*Теорема об отсутствии бесплатных обедов* (No Free Lunch theorem, NFL), описанная в работе [16.2], утверждает, что во всех возможных случаях в рамках одной задачи ни один алгоритм не работает лучше, чем случайный поиск. Основная идея состоит в том, что наличие некоторой локальной информации может для любых задач быть как полезным, так и вредным (то же самое касается и обучения — см. главу 25. Основы машинного обучения). На практике задачи, в которых локальная информация вредна, решаются редко. Впрочем, бывает, что пользы от информации тоже мало. В любой задаче справедливо сказать, что некоторые алгоритмы используют информацию о задаче лучше, чем другие, причем для каждой задачи это свои алгоритмы.

Ни один алгоритм не эффективен, если все решения, кроме лучшего, имеют одинаковое качество. Приведем некоторые выводы, следующие из теоремы NFL и общего опыта:

- ◆ используйте информацию о конкретной проблеме. Правильно заданные нижние границы и выбранные типы перемещения делают алгоритм эффективным и не дают ему рассматривать некачественные решения. Наличие такой информации дает преимущество перед перебором методом грубой силы. Также большое значение имеют пошаговая оценка и эффективное представление задачи;
- ◆ эффективные алгоритмы тратят мало ресурсов на моделирование. Например, для 3К алгоритм V&V лучше, чем A\*;
- ◆ хорошо проработанные алгоритмы в конечном итоге находят решение с вероятностью 1, как и случайный поиск, даже при каких-то нереальных условиях. Это способствует экономии ресурсов. Доказательства эффективности обычно полезны только в качестве «моральной поддержки», а подробная теория, как правило, бесполезна, если не помогает в принятии решений о реализации;
- ◆ метаэвристики, обеспечивающие выполнимость, могут не сработать для проблем с ограничениями «черного ящика», и даже подходящее начальное решение в этом случае может быть трудно подобрать. Тогда уж лучше всего использовать штрафы, создающие градиент;
- ◆ для очень больших экземпляров обычно используются только жадные алгоритмы и локальный поиск;

- ◆ во многих задачах есть *фазовые переходы*, когда экземпляры генерируются случайным образом с определенными параметрами. Эти параметры просты, но неправильная их настройка делает задачу практически неразрешимой;
- ◆ у многих хорошо изученных задач есть сложные методы решения, которые превосходят общие. Например, для ЗК *эвристика Кернигана — Лина* хорошо решает экземпляры с миллионом точек, а экземпляры с  $< 10\,000$  точек часто могут быть доказуемо оптимальным образом решены с помощью сложных алгоритмов (см. [16.1]), хотя в обоих случаях на это уходит немало времени выполнения;
- ◆ некоторые задачи считаются слишком сложными. Например, в *квадратичной задаче о назначениях* (quadratic assignment problem), которая является задачей перестановки, только очень маленькие экземпляры были решены точно. Такие задачи являются фактически конструктивным доказательством NFL. Когда алгоритм заявляет, что в каком-то случае он справился лучше, чем лучшее известное решение, это не означает, что эта разница достаточно велика и что это улучшение достаточно хорошо для других известных проблем.

## 16.19. Примечания по реализации

Большинство реализаций во многом оригинальны, по крайней мере, в вопросах выбора структуры API, который дорабатывался в процессе применения алгоритмов к разным задачам с разными свойствами.

В точных алгоритмах единственным нововведением было использование индексированной кучи в алгоритме  $A^*$ .

В разделе метаэвристики реализация имитации отжига является оригинальной, и для ее доведения до ума потребовалось много экспериментов. В генетических алгоритмах нововведением является формула размера популяции. В остальном реализации сделаны по учебникам, но из-за разнообразия вариантов следует выбирать их тщательно.

Обработка ограничений реализована довольно примитивно, но выбор основного алгоритма оригинален.

## 16.20. Комментарии

Оценка параметра  $T_{\text{last}}$  в алгоритме имитации отжига является оригинальной, как и использование значения  $T_{\text{cap}}$ . Без них любое предположение о диапазоне  $T$  можно считать произвольным, и контроль над небольшими изменениями оказывается невозможен. Другой подход, основанный на размере окрестности, упоминается в книге [16.17], но с точки зрения пользователя передача такой информации кажется неуместной и бесполезной. В этой книге вы можете почитать о ранних попытках принятия решений о реализации, но следует помнить, что в описанной там реализации выполняется множество ходов с одним и тем же значением  $T$ .

Алгоритм имитации отжига, по всей видимости, легко улучшить. Например, можно использовать непараметрический вариант. Обычно предлагается вариант с *принятием порога* — предполагается, что экспоненциальная переменная всегда равна 1. Но этот вариант не работает в задачах, где значение  $\Delta f$  всегда одно и то же, а хорошая логиче-

ская настройка параметров невозможна, в отличие от вероятностных вариантов. Единственным, но важным достоинством этого варианта, является внедрение современной версии имитации отжига с одним движением при каждом значении  $T$ . Также возможно реализовать немонотонное снижение температуры и возможное ее увеличение, если прогресса не наблюдается. Я не буду углубляться в эту тему, потому что такие алгоритмы не пользуются популярностью, но если вам интересно, можете почитать о них в работе [16.16].

Литературы по генетическим алгоритмам очень много. Начать можно с [16.16] и [16.5]. [16.2] и [16.15] — тоже хорошие работы. В ранних версиях генетических алгоритмов вместо локального поиска использовались случайные мутации. В природе то же самое — если бы локальный поиск заменил мутацию, мы были бы сверхлюдьми.

Было предложено много других метаэвристик (в работе [16.3] приведены основы, а в [16.16] и [16.5] тема раскрывается подробнее). Те, что я не обсуждал, я исключил на основании упомянутых общих принципов. Некоторым же, напротив, требуются дополнительные комментарии, приведенные далее.

*Табу-поиском* называют такую настройку локального поиска, которая запрещает определенные решения, запоминая атрибуты посещенных решений или сделанных ходов. Используйте память, а не рандомизацию, чтобы избежать локальных оптимумов. Некоторые очевидные недостатки табу-поиска:

- ◆ сейчас неясно, как реализовать этот алгоритм. К примеру, можно для каждого представления хранить список обратных шагов. Но это кажется эффективным только с лучшими или первыми улучшающими шагами, а списка небольшого размера недостаточно, чтобы избежать локальных минимумов в больших окрестностях. Например, при большом  $n$  и окрестности квадратичного размера даже локальный поиск не будет перебирать все возможные шаги. А случайный шаг работает всегда. По тем же соображениям алгоритм не распространяется на непрерывные задачи без какой-либо явной дискретизации;
- ◆ алгоритм плохо работает со случайными шагами, поэтому не будет масштабироваться до больших  $n$ ;
- ◆ для задач, в которых этот алгоритм считается эффективным, эффективность эта достигается только благодаря тонкой настройке и участию эксперта. Поэтому он больше подходит экспертов, а самонастраиваемой версии у него нет. Например, алгоритм может заикливаться, а для исправления этой проблемы требуются дополнительные средства (см., например, в [16.3]);
- ◆ трудно вывести свойства сходимости.

Оценка алгоритмов распределения аналогична алгоритму ЕМ (см. главу 28. Машинное обучение: кластеризация):

1. Создать вероятностную модель для генерации решений.
2. Пока алгоритм не сойдется:
3.     Создать популяцию из модели.
4.     Выполнить локальный поиск по каждому члену популяции.
5.     Заново оценить параметры модели от лучших представителей популяции.

Этот метод выглядит привлекательным хотя бы потому, что коллапс популяции в нем не происходит, но и свои минусы тоже есть:

- ◆ даже в случае представления в виде подмножества, в котором модель представляет собой вектор параметров Бернулли, взаимозависимость не моделируется;
- ◆ в задачах перестановки, таких как ЗК, требуется для хранения всех пар расстояний  $O(n^2)$  памяти, а алгоритм работает по принципу *эвристики построения ближайшего соседа* (начинает с любого города и идет к ближайшему непосещенному);
- ◆ в задачах разделения эффективность неизвестна (и нигде об этом не говорится). Например, список возможных разделов также занимает память  $O(n^2)$  и не работает, потому что перенумерация разделов не меняет решение, а переназначает распределения.

Самый естественный алгоритм в этой категории — *метод перекрестной энтропии* (см. [16.11]). Этот алгоритм не использует локальный поиск и реализует большой, немасштабируемый выбор размера популяции, но его легко настроить. Еще один популярный метод — *оптимизация колонии муравьев* — имитирует выбор шага через поведение муравьев (см. [16.4]). У этого алгоритма больше параметров, но он поддерживает дополнительные обновления, т. к. генерирует по одному новому решению за раз. На его настройку требуется гораздо больше сил, чем в случае перекрестной энтропии, но зато он более портативный.

Генетический алгоритм можно попробовать в деле через алгоритм оценки распределения с непараметрическим представлением модели. При скрещивании выполняется как обновление модели, так и создание новой популяции за один шаг. Может показаться, что лучше иметь непараметрическую модель, сохраняя саму популяцию и использовать ее непосредственно вместо модели, но в простой задаче это эквивалентно наличию модели, а в сложной задаче алгоритм без дополнительных усилий работать не будет. Генетическое скрещивание кажется единственным разумным способом прямого использования популяции.

В *поиске переменной окрестности* задействуется локальный поиск с несколькими типами перемещения. Базовый алгоритм использует перемещения из большей окрестности, находясь в локальном оптимуме меньшей, обычно содержащейся в ней окрестности. С точки зрения пользователя, алгоритм неуклюж, поскольку пользователю придется определить несколько типов шагов. К тому же алгоритм плохо работает со случайными ходами, потому что не знает, когда переключаться. В версии, в которой несколько шагов движения выполняются как составное движение, нужно реализовать отмену шага, но это тоже лишнее неудобство для пользователя.

*Алгоритм GRASP* — это видоизмененный жадный алгоритм для задач, в которых используется случайный выбор, — например, путем определения следующего компонента с вероятностью, пропорциональной значению или рангу возможного следующего шага. Результат улучшается локальным поиском. В более продвинутой версии алгоритма также сохраняется набор хороших решений и используется *повторное связывание путей* — т. е. исследуется путь между построенным решением и случайным решением из набора (см. [16.10]). Но:

- ◆ базовая версия вряд ли лучше случайного перезапуска, за исключением задач, где жадные алгоритмы очень хороши;

- ◆ расширенная версия делает примерно то же, что и генетическое скрещивание, — ее легко реализовать для подмножеств, но не для перестановок.

В книге [16.9] всесторонне рассматриваются несколько реализаций метаэвристики. Но это в основном фреймворки от исследователей и для исследователей, а не готовые решения для пользователей. Например, во многих из них имеет место неказистая организация, чрезмерная объектная ориентация и отвечающие сразу за всё классы, а еще добавлены методы и настройки, которые не следует использовать на практике.

## 16.21. Советы по дополнительной подготовке

- ◆ Для задачи максимальной выполнимости реализуйте жадный алгоритм, который итеративно выбирает переменную и устанавливает для нее значение, удовлетворяющее большинству предложений. Вы можете циклически перебирать предложения в случайном порядке, пока не будут внесены изменения.
- ◆ Реализуйте алгоритм В&В для задачи упаковки контейнера как расширение жадного алгоритма с первым подходящим шагом. Например, можно в качестве функции ограничения разделить оставшийся вес на размер корзины.
- ◆ Рассмотрите различные стратегии перезапуска для алгоритма  $A^*$ . Очевидным вариантом является завершение решения с использованием  $A^*$  в реальном времени. Более сложный вариант — использование частичного решения в качестве начального для следующего раунда  $A^*$ . Алгоритм повторяется, пока не будет построено полное решение. В каждой итерации должно делаться, по крайней мере, одно перемещение, но обычно их делается несколько, в зависимости от коэффициента ветвления и ограничений памяти. Изучите качество этого приближения.
- ◆ У алгоритма  $A^*$  существует простой способ сэкономить память — использовать качество существующего приближенного решения в роли границы отсечки и не ставить в очередь ходы, нижние границы которых превышают ее. Попробуйте реализовать это и посмотреть, сколько памяти удастся сэкономить. Например, для начала можно использовать метаэвстику.
- ◆ Поэкспериментируйте с алгоритмом  $A^*$  в реальном времени, чтобы проверить его полезность. Попробуйте сначала решить ЗК, а затем применить локальный поиск. Работает ли он лучше, чем имитация отжига?
- ◆ Реализуйте алгоритм RBFS нерекурсивным способом, также сохраняя в стеке перемещения кучу, альтернативный шаг и стоимость пути.
- ◆ В ЗК попробуйте поменять соседние элементы местами. В этом случае размер окрестности вычисляется линейно, а не квадратично, в отличие от других шагов.
- ◆ Реализуйте локальный поиск шагов для задачи комбинаций. Определите моделируемый отжиг и генетические алгоритмы (в последнем случае также нужно скрещивание). Найдите задачу, на которой можно его проверить.
- ◆ Реализуйте выбор первого улучшающего шага в некоторых из описанных общих представлений. Например, можно получить список шагов и рандомизировать порядок. Это можно выполнить специализированным локальным поиском, критерием завершения которого было бы отсутствие улучшающих шагов. Шаги нулевого качества могут заиклить алгоритм. Как этот подход соотносится со случайными шага-

ми для небольших задач? А для крупных? Имеет ли смысл начинать со случайных шагов и переключаться на список шагов, когда множество случайных шагов не дает улучшения? Более эффективный подход — использовать итератор перемещения. Это позволяет избежать генерации всех ходов, но порядок получается не случайным, а алгоритм — предвзятым. В лучшем случае можно случайным образом выбрать направление итерации или использовать что-то вроде перечисления кода Грея с итераторами, чтобы получить некоторую случайность. Как первое улучшение, так и лучшие шаги дают дешевую оценку.

- ◆ Исследуйте на примере ЗК специальные приемы, например основанные на множественных обращениях. Можно ли их распространить на общие стратегии, такие как ILS?
- ◆ Попробуйте реализовать имитацию отжига, применяя распределение с толстым хвостом — например, распределение Леви. Для выбора параметров используйте ту же логику — т. е. вычисление соответствующих вероятностей. Стоит также сохранять для безопасности лучшее решение. Попробуйте сделать то же самое и для числовых задач (см. главу 24. *Численная оптимизация*).
- ◆ Исследуйте возможность реализовать имитацию отжига с относительным изменением качества. Позволяет ли это упростить настройку параметров? Улучшается ли производительность?
- ◆ Реализуйте табу-поиск для различных задач, используя лучшие или первые улучшающие шаги. Попробуйте решить с помощью этого поиска малые и большие задачи и сравните эффективность с имитацией отжига.
- ◆ Изучите вероятность приемлемости в алгоритме имитации отжига для любой задачи, используя скользящее окно, например в 1000 шагов. Попробуйте менять значение шага в зависимости от текущего  $T$ . Является ли изменение вероятности в больших и малых задачах одинаково разумным?
- ◆ Реализуйте случайный поиск и локальный поиск со случайным перезапуском. Алгоритм ILS не реализуется только для задач разделения. Проведите несколько экспериментов, чтобы показать, что представленные метаэвристики позволяют улучшить результаты.
- ◆ Используйте представленные алгоритмы для решения некоторых других задач — таких как решение кубика Рубика. Можно также взять пару задач из шахмат:
  - найти на доске  $n \times n$  такое место для двух ферзей, чтобы они не угрожали друг другу;
  - найти наименьшее число коней, способных покрыть атакой доску  $n \times n$ .

## 16.22. Список рекомендуемой литературы

- 16.1. Applegate D. L., Bixby R. E., Chvatal V., & Cook, W. J. (2007). *The Traveling Salesman Problem: a Computational Study*. Princeton University Press.
- 16.2. Burke E. K., & Kendall G. (Eds.). (2013). *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer.
- 16.3. Chopard B., & Tomassini M. (2018). *An Introduction to Metaheuristics for Optimization*. Springer.

- 16.4. Dorigo M., & Stützle T. (2004). *Ant Colony Optimization*. MIT Press.
- 16.5. Gendreau M., & Potvin J. Y. (Eds). (2018). *Handbook of Metaheuristics*. Springer.
- 16.6. Gonzalez T. F. (2018). *Handbook of Approximation Algorithms and Metaheuristics*. CRC.
- 16.7. Kopec D., Pileggi C., Ungar D., & Shetty S. (2016). *Artificial Intelligence and Problem Solving*. Stylus Publishing, LLC.
- 16.8. Michiels W., Aarts E., & Korst J. (2007). *Theoretical Aspects of Local Search*. Springer.
- 16.9. Parejo J. A., Ruiz-Cortés A., Lozano S., & Fernandez P. (2012). Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing*, 16(3), 527–561.
- 16.10. Resende M. G., & Ribeiro C. C. (2016). *Optimization by GRASP*. Springer.
- 16.11. Rubinstein R. Y., & Kroese D. P. (2004). *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning*. Springer.
- 16.12. Russell S. J., Norvig P. (2010). *Artificial Intelligence: a Modern Approach*. Prentice Hall.
- 16.13. Salamon P., Sibani P., & Frost R. (2002). *Facts, Conjectures, and Improvements for Simulated Annealing*. SIAM.
- 16.14. Sipser M. (2013). *Introduction to the Theory of Computation*. Centage Learning.
- 16.15. Schneider J., & Kirkpatrick S. (2006). *Stochastic Optimization*. Springer.
- 16.16. Talbi E. G. (2009). *Metaheuristics: from Design to Implementation*. Wiley.
- 16.17. van Laarhoven P. J., & Aarts E. H. (1987). *Simulated annealing: Theory and applications*. Springer.
- 16.18. Vazirani V. V. (2004). *Approximation Algorithms*. Springer.
- 16.19. Weihe K. (2001). On the differences between «practical» and «applied». In *Algorithm Engineering* (pp. 1–10). Springer.
- 16.20. Wikipedia (2019). Coupon collector's problem.  
**[https://en.wikipedia.org/wiki/Coupon\\_collector's\\_problem](https://en.wikipedia.org/wiki/Coupon_collector's_problem)**. Accessed November 16, 2019.

# 17. Большие числа

## 17.1. Введение

Иногда, когда встроенных типов недостаточно, требуется арифметика произвольной точности — например, в задачах криптографии и научных вычислениях. Мы рассмотрим здесь простую реализацию класса больших чисел с учетом того, что:

- ♦ большинство алгоритмов представляют собой формализацию школьной математики;
- ♦ сложность операции измеряется количеством цифр на входе.

## 17.2. Представление

Число состоит из знака и вектора «цифр», представленных в некотором основании. Вот правила формирования однозначного, простого и эффективного представления:

- ♦ наименьшая значащая цифра находится по индексу 0;
- ♦ Пусть  $w \leq$  наибольшего доступного размера слова, который предполагается равным `unsigned long long`. Основание  $B = 2^{w/2}$ , потому что произведение двух битовых чисел  $w/2$  соответствует  $w$ -битному слову. Алгоритмы предполагают, что  $B$  является степенью двойки. Число `uint_32` можно использовать как наибольшее совместимое базовое слово;
- ♦ у числа нет ведущих нулей, за исключением случаев, когда число = 0;
- ♦ допускается существование  $-0$  и  $0$ , т. к. это проще, чем исправлять  $-0$  после каждой операции.

Структура памяти большого числа, равного, например,  $2^{100}$ , представлена на рис. 17.1.

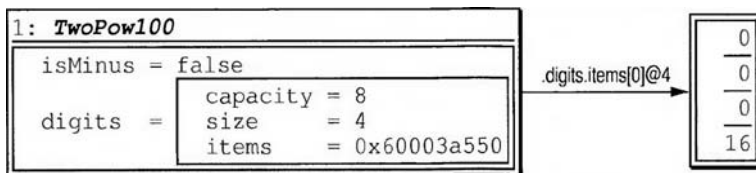


Рис. 17.1. Структура памяти большого числа, равного  $2^{100}$

```
class Number : public ArithmeticType<Number>
{
    bool isMinus;
    typedef uint32_t DIGIT;
    typedef unsigned long long LARGE_DIGIT;
```



```

enum{BASE_RADIX = numeric_limits<DIGIT>::digits};
Vector<DIGIT> digits;
DIGIT getDigit(int i) const {return i < nDigits() ? digits[i] : 0;}
Number(int size, DIGIT fill): digits(size, fill), isMinus(false) {}
void trim(){while(nDigits() > 1 && isZero()) digits.removeLast();}

public:
typedef DIGIT DIGIT_TYPE;
int nDigits() const {return digits.getSize();}
DIGIT& operator[](unsigned int i) {return digits[i];}
DIGIT const& operator[](unsigned int i) const {return digits[i];}
void appendDigit(DIGIT const& digit) {digits.append(digit);}
bool isZero() const {return digits.lastItem() == 0;}
bool isPositive() const {return !isMinus && !isZero();}
bool isNegative() const {return isMinus && !isZero();}
void negate() {isMinus = !isMinus;}
Number abs() const {return isMinus ? -*this : *this;}
bool isOdd() const {return digits[0] % 2;}
bool isEven() const {return !isOdd();}
Number(): isMinus(false), digits(1, 0) {} // по умолчанию значение 0
// удобный конструктор для одноразрядных чисел
explicit Number(long long x): isMinus(x < 0), digits(1, std::abs(x))
    {assert(std::abs(x) <= numeric_limits<DIGIT_TYPE>::max());}
};

```

Если возникает необходимость преобразования десятичных цифр, могут быть созданы и дополнительные конструкторы (мы обсудим это позже). Алгоритмы сравнения должны уметь сравнивать абсолютное значение:

```

bool absLess(Number const& rhs) const
{ // чем больше разрядов, тем число больше, иначе сравнение
  // выполняется по старшему биту
  if(nDigits() != rhs.nDigits()) return nDigits() < rhs.nDigits();
  for(int i = nDigits() - 1; i >= 0; --i)
    if(digits[i] != rhs[i]) return digits[i] < rhs[i];
  return false;
}

bool absEqual(Number const& rhs) const
{ // чем больше разрядов, тем число больше, иначе сравнение
  // выполняется по старшему биту
  if(nDigits() != rhs.nDigits()) return false;
  for(int i = 0; i < nDigits(); ++i)
    if(digits[i] != rhs[i]) return false;
  return true;
}

bool operator<(Number const& rhs) const // обработка случая -0 = 0
{return (isMinus && !rhs.isMinus && !isZero()) || absLess(rhs);}
bool operator==(Number const& rhs) const // обработка случая -0 = 0
{return (isMinus == rhs.isMinus || isZero()) && absEqual(rhs);}

```

## 17.3. Сложение и вычитание

*Полный сумматор* складывает цифры  $a$ ,  $b$  и *бит переноса* и возвращает сумму и новый перенос. Например,  $7 + 8$  с переносом 0 равно 5 с переносом 1. Зная это, мы можем складывать двухразрядные векторы по цифрам, распространяя перенос (рис. 17.2).

Результат не более чем на одну цифру длиннее любого слагаемого и обрезается, если эта цифра = 0:

```
static DIGIT fullAdder(DIGIT a, DIGIT b, bool& carry)
{
    LARGE_DIGIT sum = LARGE_DIGIT(a) + b + carry;
    carry = sum >> BASE_RADIX;
    return sum;
}

static Number add(Number const& a, Number const& b)
{ // O(|a|+|b|)
    int n = max(a.nDigits(), b.nDigits());
    Number result(n + 1, 0);
    bool carry = 0;
    for(int i = 0; i < n; ++i)
        result[i] = fullAdder(a.getDigit(i), b.getDigit(i), carry);
    result[n] = carry;
    result.trim();
    return result;
}
```

$$\begin{array}{r}
 a \quad + \quad 048 \\
 b \quad \quad 057 \\
 \hline
 \text{Сумма} \quad 105 \\
 \text{Перенос} \quad 0110
 \end{array}$$

Рис. 17.2. Пример распространения переноса в операции сложения

$$\begin{array}{r}
 a \quad - \quad 105 \\
 b \quad \quad 057 \\
 \hline
 \text{Результат} \quad 048 \\
 \text{Перенос} \quad 0110
 \end{array}$$

Рис. 17.3. Пример распространения переноса при вычитании

Вычитание двух положительных чисел предполагает, что первое больше, и помещает результат в него. Перенос распространяется, как при сложении от младшего к старшему разряду (рис. 17.3). В школе учили не так, конечно.

```
static void sub(Number& a, Number const& b)
{ // O(|a| + |b|)
    bool carry = 0;
    for(int i = 0; i < a.nDigits(); ++i)
    {
        LARGE_DIGIT digit = LARGE_DIGIT(b.getDigit(i)) + carry;
        carry = a[i] < digit;
        a[i] -= digit; // неявный модуль В
    }
    a.trim();
}
```

Операторы + и – реализованы поверх этих функций и зависят от знака:

```
Number operator-() const
{
    Number result = *this;
    result.negate();
    return result;
}

Number& operator+=(Number const& rhs)
{
    if(isMinus == rhs.isMinus)
    { // сложение с одинаковыми знаками
        digits = add(*this, rhs).digits;
        return *this;
    }
    else return *this -= -rhs; // иначе вычитание
}

Number& operator-=(Number const& rhs)
{
    if(isMinus == rhs.isMinus)
    { // вычитание с одинаковыми знаками
        if(absLess(rhs))
        {
            Number temp = rhs;
            sub(temp, *this);
            temp.negate();
            *this = temp;
        }
        else sub(*this, rhs);
        trim();
        return *this;
    }
    else return *this += -rhs; // иначе сложение
}
```

## 17.4. Операции сдвига

Сдвиг — это умножение или деление на степень двойки. Сдвиг слов и битов в целях эффективности выполняется отдельно для, например, набора битов. Время выполнения равно  $O(n)$ :

```
Number& operator>>=(unsigned int k)
{
    int skipDigits = k/BASE_RADIX, last = nDigits() - skipDigits,
        skipBits = k % BASE_RADIX;
    if(skipDigits > 0) // сперва применяются сдвиги на целое число
        for(int i = 0; i < nDigits(); ++i)
            digits[i] = i < last ? digits[i + skipDigits] : 0;
    if(skipBits > 0) // затем выполняется сдвиг
    {
        DIGIT carry = 0, tempCarry;
```

```

        for(int i = last - 1; i >= 0; --i)
        {
            tempCarry = digits[i] << (BASE_RADIX - skipBits);
            digits[i] = (digits[i] >> skipBits) | carry;
            carry = tempCarry;
        }
    }
    trim(); // в случае введения нулей
    return *this;
}

Number operator<<=(unsigned int k)
{ // сперва освобождаем место для дополнительных цифр
    int skipDigits = k/BASE_RADIX, skipBits = k % BASE_RADIX;
    for(int i = 0; i < skipDigits + 1; ++i) digits.append(0);
    if(skipDigits > 0) // применение полных сдвигов
        for(int i = nDigits() - 1; i >= 0; --i)
            digits[i] = i < skipDigits ? 0 : digits[i - skipDigits];
    if(skipBits > 0) // выполнение сдвига
    {
        DIGIT carry = 0;
        for(int i = skipDigits; i < nDigits(); ++i)
        {
            DIGIT tempCarry = digits[i] >> (BASE_RADIX - skipBits);
            digits[i] = (digits[i] << skipBits) | carry;
            carry = tempCarry;
        }
    }
    trim(); // в случае, если сдвиг бита недостаточно велик, чтобы
            // заполнить пространство
    return *this;
}

```

## 17.5. Умножение

Можно использовать школьную технику  $O(|a||b|)$ . Умножение число  $\times$  число превращается в последовательность умножений число  $\times$  цифра. Результат последнего не более чем на одну цифру больше, чем число. Например,  $98 \times 99 = 99 \times 8 + (99 \times 9) \ll_{10} 1$ . Кроме того,  $9 \times 9$  есть 1 с переносом 8, и  $99 \times 9$  вычисляется поразрядно, чтобы получить  $011 + 880 = 891$ :

```

static DIGIT digitMult(DIGIT a, DIGIT b, DIGIT& carry)
{
    LARGE_DIGIT prod = LARGE_DIGIT(a) * b;
    carry = prod >> BASE_RADIX;
    return prod;
}

static Number mult(Number const& a, DIGIT const& b)
{
    Number result(a.nDigits() + 1, 0);
    bool addCarry = 0;
    DIGIT multCarry = 0, newMultCarry;

```

```

    for(int i = 0; i < a.nDigits(); ++i)
    {
        result[i] = fullAdder(digitMult(a[i], b, newMultCarry), multCarry, addCarry);
        multCarry = newMultCarry;
    }
    result[a.nDigits()] = fullAdder(0, multCarry, addCarry);
    result.trim();
    return result;
}

Number& operator*=(Number const& rhs)
{ // O(|a|+|b|), умножение на одну цифру
    Number const& a = *this, b = rhs;
    Number product(a.nDigits() + b.nDigits(), 0);
    for(int j = 0; j < b.nDigits(); ++j)
        product += mult(a, b[j]) << BASE_RADIX * j;
    product.isMinus = a.isMinus != b.isMinus; // в случае разных знаков
                                              // число отрицательное

    product.trim();
    return *this = product;
}

```

## 17.6. Деление

Операция  $a/b$  вычисляет частное  $q$  и остаток  $r$  с использованием длинного деления. Предположим, что оба числа положительные:

1. Пусть  $r = a$ .
2. Нормализация  $r$  и  $b$ , чтобы взять самый значимый разряд от  $b \geq (B/2)$ .
3. Пока  $r < b$ :
4. Найти  $p$  = наибольшая степень основания  $B$  такая, что  $s = pb \leq r$ .
5. Найти максимальное  $k$  такое, что  $ks \leq r$ .
6.  $r -= ks$ .
7.  $q += kp$ .
8. Повторная нормализация  $r$ .

Может показаться, что в пункте (5) нужно выполнить бинарный поиск, но это можно сделать с  $O(1)$  умножениями.

Теорема (см. [17.1]): Пусть  $z$  — старшая цифра числа  $s$ , а  $x$  и  $y$  — две старшие цифры числа  $r$ . Пусть:

$$g = \begin{cases} 1, & \text{если } nDigits(r) = nDigits(s) \\ \frac{xB + y}{z} & \text{в противном случае} \end{cases}.$$

Тогда  $k \leq g \leq k + 2$ .

Доказательство: Поскольку  $b$  нормализовано,  $z \geq B/2$ . Когда количество цифр одинаково,  $k = 1$ , потому что  $z \leq x \leq B - 1$ . В противном случае  $g - k$  является наименьшим (первый случай) или наибольшим (второй случай), если остальные цифры числа  $a$

наибольшие, а числа  $b$  наименьшие в первом случае, и наоборот во втором случае, а  $z$  наименьшее. В первом случае из-за дополнительных разрядов числа  $a$  получаем  $g - k \leq \frac{1}{z} = 0$ . Во втором случае  $k = \frac{xB + y}{z + 1}$ ,  $gz + r_1 = k(z + 1) + r_2$  и  $g = k + \frac{k + r_2 - r_1}{z} \leq k + 2$ , т. к.  $\frac{k}{z} < 2$  и  $r_2 - r_1 < z$ .

Таким образом, деление сначала сдвигает  $r$  и  $b$  так, чтобы получить  $z \geq B/2$ , и сдвигает число  $r$  назад после расчета:

```
static DIGIT findK(Number const& s, Number const& b)
{ // O(|s|), поиск такого k, чтобы 0<=k<BASE и kb<=s<(k+1)b
  DIGIT guess = s.digits.lastItem()/b.digits.lastItem();
  if(s.nDigits() > b.nDigits()) guess = (s.digits.lastItem() *
    (1ull << BASE_RADIX + s[s.nDigits() - 2])/b.digits.lastItem());
  while(s < mult(b, guess)) --guess; // вычисление выполняется <= 2 раз
  return guess;
}

static Number divide(Number const& a, Number const& b1, Number& q)
{ // O(|a| * |b|)
  assert(!b1.isZero());
  q = 0;
  Number b = b1.abs(), r = a.abs(); // первая нормализация
  int norm = BASE_RADIX - lgFloor(b.digits.lastItem()) - 1;
  r <<= norm;
  b <<= norm;
  for(int i = r.nDigits() - b.nDigits(); i >= 0; --i)
  {
    int shift = i * BASE_RADIX;
    Number s = b << shift;
    DIGIT k = findK(r, s);
    q += mult(Number(1) << shift, k); // q += pk
    r -= mult(s, k);
  } // в случае разных знаков q и r отрицательные
  q.isMinus = r.isMinus = a.isMinus != b1.isMinus;
  return r >>= norm; // повторная нормализация
}

Number& operator%=(Number const& rhs)
{
  Number quotient(0);
  return *this = divide(*this, rhs, quotient);
}

Number& operator/=(Number const& rhs)
{
  Number quotient(0);
  divide(*this, rhs, quotient);
  return *this = quotient;
}
```

Умножение выполняется за время  $O(|a||b|)$ .

## 17.7. Преобразование в десятичное число

Преобразование в другое основание  $B_2$  сводится к делению и имеет такую же сложность. Несколько раз выполните деление на  $B_2$  и сохраняйте остатки, которые затем нужно записать в обратном порядке. На практике требуется только выполнить преобразование в десятичное число и использовать в качестве результата строку, поэтому при необходимости остается лишь добавить знак «минус». Об этом часто спрашивают на собеседованиях:

```
string toDecimalString() const
{
    string result;
    Number r = *this;
    while(!r.isZero())
    {
        Number q(0);
        result.push_back('0' + divide(r, Number(10), q)[0]);
        r = q;
    }
    if(isMinus) result.push_back('-');
    reverse(result.begin(), result.end());
    return result;
}

Number(string const& decimals)
{
    assert(decimals.length() > 0);
    int firstDigit = 0;
    if(decimals[0] == '-') // знак "минус"
    {
        assert(decimals.length() > 1);
        isMinus = true;
        firstDigit = 1;
    }
    Number result(0);
    for(int i = firstDigit; i < decimals.length(); ++i)
    {
        if(i == firstDigit) assert(decimals[i] != '0'); // наибольшее
                                                    // значимое число не может быть равно нулю
        else result *= Number(10);
        assert(decimals[i] >= '0' && decimals[i] <= '9');
        result += Number(decimals[i] - '0');
    }
    digits = result.digits;
}
```

## 17.8. Возведение в степень

Число вида  $x \times x \times x \times x \times x \times x \times x \times x$  содержит повторно используемые квадраты:  $x^n = x^{n/2} (x^2)^{n/2}$ . Это наблюдение позволяет получить эффективный алгоритм с  $\lg(i)$  умножениями. В начале у нас есть числа  $x$  и  $n$ , а на следующей итерации используются

$x^2$  и  $n/2$  и т. д. Вычисляемую степень нужно уменьшать после каждого умножения, чтобы избежать больших подпроизведений и не потерять в эффективности:

```
Number power(Number const& t, Number n)
{
    Number x = t, result(1);
    for(;;)
    {
        if(n.isOdd()) result *= x;
        n >>= 1; // дешевое деление на 2
        if(n.isZero()) break;
        x *= x;
    }
    return result;
}

Number modPower(Number const& t, Number n, Number const& modulus)
{
    assert(!modulus.isZero());
    Number x = t, result(1);
    for(;;)
    {
        if(n.isOdd())
        {
            result *= x;
            result %= modulus;
        }
        n >>= 1; // дешевое деление на 2
        if(n.isZero()) break;
        x *= x;
        x %= modulus;
    }
    return result;
}
```

## 17.9. Вычисление логарифма

Выполняется за время  $O(1)$ , потому что значение имеет только самый старший бит:

```
int lg()const
{
    return BASE_RADIX * (nDigits() - 1) + lgFloor(digits.lastItem());
}
```

## 17.10. Целочисленный квадратный корень

Метод Ньютона (см. главу 23. Численные алгоритмы: работа с функциями) позволяет вычислить правильный результат (см. [17.1]). Для эффективности он начинается с верхней границы, вычисленной с помощью логарифма:

```
Number sqrtInt(Number const& t)
{
    // нужно удачное начальное значение
    Number x(Number(1) << (1 + t.lg()/2));
}
```



```

for(;;)
{
    Number y = (x + t/x)/Number(2);
    if(y < x) x = y;
    else return x;
}
}

```

Обычно для сходимости требуется  $O(1)$  итераций, поэтому время выполнения равно  $O(\text{деление})$ .

## 17.11. Наибольший общий делитель

*Алгоритм Евклида* вычитает меньшее число из большего, пока меньшее не станет равно 0. Для эффективности можно вместо этого выполнить деление с остатком, и когда  $a > b$ ,  $\text{НОД}(a, b) = \text{НОД}(b, r = a \% b)$ . Расширенная версия вычисляет  $x, y$  так, что получаем  $\text{НОД}(a, b) = ax + by$  путем сохранения всех частных. Поскольку после каждой итерации НОД остается неизменным, когда  $a$  становится равно  $r$ ,  $\text{НОД}(a, b) = \text{НОД}(r, b) = rx + by = (a - bq)x + by = ax + b(y - qx)$ , и когда  $b$  становится равно 0,  $\text{НОД}(r, b) = r \times 1 + b \times 0$ . Время выполнения составляет  $O(|a||b|)$  (см. [17.2]):

```

Number extendedGcdR(Number const& a, Number const& b, Number& x, Number& y)
{
    if(!b.isPositive())
    {
        x = Number(1);
        y = Number(0);
        return a;
    }
    Number q, r = Number::divide(a, b, q), gcd = extendedGcdR(b, r, y, x);
    y -= q * x;
    return gcd;
}

Number extendedGcd(Number const& a, Number const& b, Number& x, Number& y)
{
    assert(a.isPositive() && b.isPositive());
    return a < b ? extendedGcdR(b, a, y, x) : extendedGcdR(a, b, x, y);
}

Number gcd(Number const& a, Number const& b)
{
    Number x, y;
    return extendedGcd(a, b, x, y);
}

```

## 17.12. Модульная инверсия

Инверсия  $a \% n$  существует, только если  $1 = \text{НОД}(a, n) = ax + ny$ . Она равна  $x$ , потому что  $(ax + ny) \% n = ax \% n$ :

```

Number modInverse(Number const& a, Number const& n)
{
    assert(a.isPositive() && a < n);
}

```

```

Number x, y;
extendedGcd(a, n, x, y);
if(x.isNegative()) x += n; // подстройка диапазона, если необходимо
return x;
}

```

Время выполнения равно  $O(|a||n|)$ .

## 17.13. Проверка числа на простоту

*Теорема Ферма:* Если  $n$  простое, то для любого  $a$  такого, что  $1 < a < n$  и  $\text{НОД}(a, n) = 1$ ,  $a^{n-1} \% n = 1$ . Но это верно и для *составных чисел Кармихеля*, таких как 561, поэтому использование обратного числа с различными  $a$  результата не даст. Можно использовать *алгоритм Миллера — Рабина*:

1. Пусть  $n - 1 = 2^c d$ , а  $d$  нечетно.
2.  $x = a^d \% n$ .
3.  $c$  раз:
4. Возвести в квадрат  $(x \bmod n)$ .
5. Если квадрат = 1, но не  $x = 1$  или  $x = n - 1$ ,  $n$  — составное, поскольку для простого  $p$   $x^2 = 1(\bmod p) \leftrightarrow (x + 1)(x - 1) = 0(\bmod p) \leftrightarrow x = 1(\bmod p)$  или  $x = -1(\bmod p)$ .

Доказано (см. [17.5]), что ответ:

- ♦ *составное* — верен;
- ♦ *несоставное* — ошибочен с вероятностью  $< 1/4$ , если  $1 < a < n$  выбирается случайным образом. Эта вероятность уменьшается с размером  $n$ , как и количество тестов ( $\approx 2^{-100}$ ).

```

bool provenComposite(Number const& a, Number const& n)
{
    Number ONE = Number(1), oddPart = n - ONE;
    int nSquares = 0;
    while(oddPart.isEven())
    {
        oddPart >>= 1;
        ++nSquares;
    }
    Number x = modPower(a, oddPart, n);
    for(int i = 0; i < nSquares; ++i)
    { // если x2 равно 1, x должно быть равно 1 или -1, если n — простое
        Number x2 = modPower(x, Number(2), n);
        if(x2 == ONE && x != ONE && x != n - ONE) return true;
        x = x2;
    }
    return x != ONE;
}

```

Каждому тесту нужно  $\leq \lg(n)$  квадратов числа  $a$ , что дает общую стоимость  $O(\lg(n)y^2)$  для случайных чисел  $a \in [1, n]$  и  $y$  разрядов. Алгоритм сначала выполняет пробное де-

ление на небольшие простые числа  $< 50$ . Этот путь до чисел около 2000 кажется оптимальным (см. [17.4]), но немного усложняет код (т. е. будет использоваться решето Эратосфена — см. главу 12. *Разные алгоритмы и методы*). Для эффективности в этой реализации выбрано  $a \in [2, \min(n, B - 1)]$ :

```
bool isPrime(Number const& n)
{
    n.debug();
    if(n.isEven() || n < Number(2)) return false;
    int smallPrimes[] = {3,5,7,11,13,17,19,23,29,31,37,41,43,47};
    for(int i = 0; i < sizeof(smallPrimes)/sizeof(int); ++i)
    {
        Number p = Number(smallPrimes[i]);
        if(n == p) return true;
        if((n % p).isZero()) return false;
    } // Алгоритм Миллера – Раббана, если деление не дает результата
    int nTrials = 1;
    int sizes[] = {73,105,132,198,223,242,253,265,335,480,543,627,747,927,
        1233,1854,4096}, nTests[] = {47,42,35,29,23,20,18,17,16,12,8,7,6,5,4,
        3,2};
    for(int i = 0; i < sizeof(sizes)/sizeof(*sizes); ++i)
        if(n.lg() < sizes[i])
        {
            nTrials = nTests[i];
            break;
        }
    while(nTrials--)
    { // для эффективности используются одноразрядные степени
        Number::DIGIT_TYPE max = numeric_limits<Number::DIGIT_TYPE>::max();
        if(provenComposite(Number(GlobalRNG().inRange(2, (Number(max) < n ?
            max : int(n[0])) - 1)), n)) return false;
    }
    return true;
}
```

Чтобы сгенерировать случайное  $n$ -битное простое число, нужно — пока результат не станет простым — сгенерировать случайное  $n$ -битное число и установить его старший и младший биты. По гипотезе Римана для любого  $n$   $O(1/n)$  чисел являются простыми, поэтому  $E[\text{количество тестов на простоту}] = O(n)$ . Результат не является криптографически безопасным, если генератор случайных чисел не защищен.

## 17.14. Рациональные числа

Вы можете извлечь целочисленное основание и показатель степени из числа двойной точности:

```
pair<long long, int> rationalize(double x)
{ // поддерживается только обычный формат (не decimal)
    assert(numeric_limits<double>::radix == 2 &&
        numeric_limits<double>::digits <= numeric_limits<long long>::digits);
    int w = numeric_limits<double>::digits, e;
```

```

x = frexp(x, &e); // x нормализуется к диапазону [0.5, 1)
long long mantissa = ldexp(x, w); // вычисление  $x^{53}$ 
return make_pair(mantissa, e - w);
}

```

Рациональное число состоит из большого числа в числителе и знаменателе, и после каждой операции они уменьшаются на величину *НОД*. Для простоты предположим, что в них одинаковое количество цифр:

```

struct Rational: public ArithmeticType<Rational>
{
    Number numerator, denominator;
    Rational(Number const& theNumerator = Number(0),
              Number const& theDenominator = Number(1)): numerator(theNumerator),
              denominator(theDenominator)
    {
        assert(!denominator.isZero());
        reduce();
    }
    Rational(double x): denominator(1), numerator(1)
    {
        pair<long long, int> mantissaExponent = rationalize(x);
        numerator = Number(mantissaExponent.first);
        int e = mantissaExponent.second;
        if(e < 0) denominator <<= -e;
        else if(e > 0) numerator <<= e;
    }
    void reduce()
    {
        Number g = gcd(numerator, denominator);
        numerator /= g;
        denominator /= g;
    }
    bool isZero() const {return numerator.isZero();}
    bool isMinus() const
        {return numerator.isNegative() != denominator.isNegative();}
};

```

Сложение и вычитание выполняются за время  $O(n^2)$  из-за поиска *НОД*, чтобы размеры были небольшими. Но, возможно, для дешевых операций *НОД* не нужен:

```

Rational operator-( ) const
{
    Rational result = *this;
    result.numerator.negate();
    return result;
}
Rational& operator+=(Rational const& rhs)
{
    numerator = numerator * rhs.denominator + rhs.numerator *
        denominator;
    denominator *= rhs.denominator;
    reduce();
}

```

```

    return *this;
}
Rational& operator+=(Rational const& rhs){return *this += -rhs;}

```

Умножение и деление выполняется за  $O(n^2)$ :

```

Rational& operator*=(Rational const& rhs)
{
    numerator *= rhs.numerator;
    denominator *= rhs.denominator;
    reduce();
    return *this;
}
Rational& operator/=(Rational const& rhs)
{
    assert(!rhs.isZero());
    numerator *= rhs.denominator;
    denominator * rhs.numerator;
    reduce();
    return *this;
}

```

Существует также полезная операция вычисления логарифма  $\lg$  и оценка с некоторой точностью:

```

int lg()const{return numerator.lg() - denominator.lg();}
Number evaluate(Number const& scale = Number(1))
{return numerator * scale / denominator;}

```

Во многих вычислениях вы можете имитировать рациональные числа, т. к. действительное число с фиксированной точностью представляет собой целое число, деленное на нормализующий коэффициент. Это позволяет, например, вычислять  $n$  знаков числа  $\pi$  до  $n$  цифр как  $\text{floor}(10^n \pi)$ , используя степенной ряд для  $4\text{atan}(1)$  и добавляя к целому числу дополнительные цифры, чтобы избежать ошибок округления. Но лучше не пытаться имитировать арифметику с плавающей точкой, используя рациональную арифметику, т. к., к примеру, последовательность умножений может привести к комбинаторному взрыву, и у вас быстро кончится память. Лучше пользоваться арифметикой с плавающей точкой произвольной точности (см. [17.1]).

## 17.15. Примечания по реализации

Помимо некоторых основных соглашений, реализации больших чисел взяты из учебников.

Получение рационального числа из числа с плавающей точкой — это новая идея. Возможно, здесь пригодится интервальная арифметика (см. главу 22. *Численные алгоритмы: введение и матричная алгебра*).

## 17.16. Комментарии

Самый быстрый способ умножения чисел длиной более 100 цифр — *умножение Карацубы* (но лишь до некоторого предела). Асимптотически лучший известный метод основан на быстром преобразовании Фурье (см. главу 23. *Численные алгоритмы: ра-*

*бота с функциями*) и очень сложен. Он используется для вычислений с миллионами цифр, где разнообразия алгоритмов нет.

## 17.17. Советы по дополнительной подготовке

- ◆ Реализуйте другие битовые операции — такие как "&".
- ◆ Попробуйте найти наилучшее самое большое простое число для пробного деления во время проверки простоты.
- ◆ Исследуйте и реализуйте арифметику с плавающей точкой произвольной точности.

## 17.18. Список рекомендуемой литературы

- 17.1. Brent R. P., & Zimmermann P. (2010). *Modern Computer Arithmetic*. Cambridge University Press.
- 17.2. Cormen T. H., Leiserson C. E., Rivest R. L., & Stein C. (2009). *Introduction to Algorithms*. MIT Press.
- 17.3. St Denis T. (2006). *BigNum Math: Implementing Cryptographic Multiple Precision Arithmetic*. Syngress.
- 17.4. Schneier B. (1995). *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley.
- 17.5. Welschenbach M. (2005). *Cryptography in C and C++*. Apress.

# 18. Вычислительная геометрия

## 18.1. Введение

В этой главе описываются наиболее полезные структуры данных для работы с точками в нескольких измерениях — например, для поиска ближайших соседей. Эти структуры относятся к области вычислительной геометрии, которая занимается расчетами таких объектов, как, например, выпуклые оболочки. Для VP-деревьев здесь приводятся некоторые детали оригинальной реализации.

В главе также представлены алгоритмы вычисления таких объектов и описаны присущие им проблемы надежности, связанные с реализацией на естественной арифметике с плавающей точкой. Дополнительная информация приведена в упоминаемой литературе.

## 18.2. Расстояния

Многие алгоритмы и структуры данных предполагают метрические расстояния между точками. *Евклидово расстояние* — наиболее полезная метрика, которая после возведения в квадрат равна сумме расстояний компонентов отдельных измерений. Это расстояние *инкрементно вычисляется* за время  $O(1)$  для каждого компонента. Например, для сравнения двух расстояний можно вычислить квадрат одного и прекратить вычисление квадрата другого, если инкрементный результат оказался больше:

```
template<typename VECTOR> class EuclideanDistance
{
    static double iDistanceIncremental(VECTOR const& lhs, VECTOR const& rhs,
        int i) // прибавление компонента
    {
        double x = lhs[i] - rhs[i];
        return x * x;
    }
    static double distanceIncremental(VECTOR const& lhs, VECTOR const& rhs,
        double bound = numeric_limits<double>::infinity())
    { // вычисление расстояния до достижения границы
        assert(lhs.getSize() == rhs.getSize());
        double sum = 0;
        for(int i = 0; i < lhs.getSize() && sum < bound; ++i)
            sum += iDistanceIncremental(lhs, rhs, i);
        return sum;
    }
public:
    struct Distance
```

```

{ // функтор метрики
    double operator() (VECTOR const& lhs, VECTOR const& rhs) const
    {return sqrt(distanceIncremental(lhs, rhs));}
};

struct DistanceIncremental
{ // инкрементный функтор, который возвращает квадрат расстояния
    double operator() (VECTOR const& lhs, VECTOR const& rhs) const
    {return distanceIncremental(lhs, rhs);}
    double operator() (VECTOR const& lhs, VECTOR const& rhs, int i) const
    {return iDistanceIncremental(lhs, rhs, i);}
    double operator() (double bound, VECTOR const& lhs, VECTOR const& rhs)
    const{return distanceIncremental(lhs, rhs, bound);}
};
};

```

Для хранения точек можно эффективно использовать словарь или некоторые схожие структуры данных, представляющие собой иерархии или деревья, где каждое поддереву покрывает часть объема, а корень покрывает всё. Тогда *расстоянием по иерархии* будет называться ближайшее расстояние между точкой запроса и любой точкой в поддереве.

## 18.3. VP-дерево

Если есть функция, вычисляющая метрическое расстояние между ключами, вы можете реализовать:

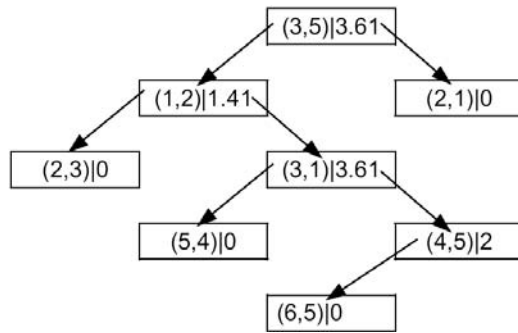
- ◆ операции отображения;
- ◆ *запрос  $k$  ближайших соседей ( $k$ -NN)*, который находит  $k$  ближайших к заданному ключу  $x$  элементов;
- ◆ *запрос расстояния*, который находит все элементы, отстоящие от  $x$  не более чем на заданное расстояние.

Подобные запросы позволяют, например, найти все строки, расстояние редактирования у которых от заданной строки  $\leq 2$ .

VP-дерево — сокращение от *vantage point* (точка обзора), также называемое деревом *почтового отделения* — выбирает объект в качестве корня и некоторый радиус  $r$  для него. Все объекты на расстоянии  $\leq r$  попадают в левое поддерево, а на расстоянии  $> r$  — в правое. Объект, у которого  $r$  = расстояние до первого объекта, вставленного в поддерево корня, становится левым дочерним. Чтобы избежать вычислительных проблем, сравнение стоит выполнять немного более гибко, отправляя чуть большие расстояния влево (см. главу 22. Численные алгоритмы: введение и матричная алгебра). Чтобы ускорить поиск, в каждом узле хранится максимальное расстояние до любого узла в поддереве (рис. 18.1).

Теорема (см. [18.5]): Пусть даны точки  $a, b, c$  в метрическом пространстве, где  $r_1 \leq d(a, b) \leq r_2$  и  $d(a, c) = k$ ,  $\max(0, k - r_2, r_1 - k) \leq d(b, c)$ . Дочерние расстояния до точки запроса ограничены снизу, когда  $a$  является ключом текущего узла,  $b$  — ключом дочернего элемента и  $c$  — ключом запроса. Поэтому следует сохранять в каждом узле радиус поддерева, т. е. расстояние до самого дальнего потомка:





**Рис. 18.1.** Структура двумерного (2D) VP-дерева с евклидовыми данными. Левая часть — это точка, а правая — расстояние до левого дочернего элемента

```

template<typename KEY, typename VALUE, typename DISTANCE> class VpTree
{
    DISTANCE lowerBound;
    static double bound(double keyDistance, double rLow, double rHigh)
    {return max(0., max(keyDistance - rHigh, rLow - keyDistance));}
    struct Node
    {
        KEY key;
        VALUE value;
        double leftChildDistance, radius;
        Node *left, *right;
        Node(KEY const& theKey, VALUE const& theValue): key(theKey), left(0),
            right(0), value(theValue), leftChildDistance(0), radius(0) {}
        double leftChildBound(double keyDistance)
        {return bound(keyDistance, 0, leftChildDistance);}
        double rightChildBound(double keyDistance)
        {return bound(keyDistance, leftChildDistance, radius);}
    } * root;
    Freelist<Node> f;
public:
    typedef DISTANCE DISTANCE_TYPE; // обновление!
    DISTANCE const& getDistance() {return lowerBound;}
    typedef Node NodeType;
    bool isEmpty() const {return !root;}
};

```

Конструкторы аналогичны бинарным деревьям поиска — за исключением того, что вместо компаратора требуется функтор расстояния:

```

Node* constructFrom(Node* n)
{
    Node* tree = 0;
    if(n)
    {
        tree = new(f.allocate())Node(n->key, n->value);
        tree->leftChildDistance = n->leftChildDistance;
        tree->radius = n->radius;
        tree->left = constructFrom(n->left);
    }
}

```

```

        tree->right = constructFrom(n->right);
    }
    return tree;
}
VpTree(DISTANCE const& theDistance = DISTANCE()): root(0),
    lowerBound(theDistance) {}
VpTree(VpTree const& rhs): lowerBound(rhs.lowerBound)
    {root = constructFrom(rhs.root);}
VpTree& operator=(VpTree const& rhs){return genericAssign(*this, rhs);}

```

Поиск идет по определению дерева, переходя к левому дочернему элементу, если расстояние узла до запроса не больше заданного, и в противном случае переходит к правому:

```

VALUE* find(KEY const& key) const
{
    Node* n = root;
    while(n && key != n->key) n = !isELess(n->leftChildDistance,
        lowerBound(key, n->key)) ? n->left : n->right;
    return n ? &n->value : 0;
}

```

Вставка на пути вниз обновляет радиус в каждом результирующем родительском узле, если новый узел становится левым дочерним:

```

void insert(KEY const& key, VALUE const& value)
{
    Node **pointer = &root, *n;
    while((n = *pointer) && key != n->key)
    {
        double d = lowerBound(key, n->key);
        n->radius = max(n->radius, d);
        if(!n->left) n->leftChildDistance = d; // образуется левый дочерний узел
        pointer = &(!isELess(n->leftChildDistance, d) ? n->left : n->right);
    }
    if(n) n->value = value; // равенство ведет к присвоению нового значения
    else *pointer = new(f.allocate())Node(key, value);
}
};

```

Высота дерева не контролируется, потому что не существует хорошего способа выполнить его балансировку (о чем подробно рассказано далее в этой главе). Таким образом, дерево может быть несбалансированным, но, как и в случае с бинарными деревьями, вставка в случайном порядке дает хорошую ожидаемую производительность. Из-за определенности иерархии для удаления узла можно использовать только слабое удаление (здесь не реализовано).

Запрос расстояния использует дочерние границы для удаления узлов дальше указанного радиуса — они не могут находиться в поддереве:

```

Vector<NodeType*> distanceQuery(KEY const& key, double radius) const
{
    Vector<NodeType*> result;
    distanceQuery(key, radius, result, root);
}

```

```

    return result;
}

void distanceQuery(KEY const& key, double radius, Vector<Node*>& result,
    Node* n) const
{
    if(!n) return;
    double d = lowerBound(n->key, key);
    if(d <= radius) result.append(n);
    if(n->leftChildBound(d) <= radius) // движение влево
        distanceQuery(key, radius, result, n->left);
    if(n->rightChildBound(d) <= radius) // затем движение вправо
        distanceQuery(key, radius, result, n->right);
}

```

Процент избыточно проверенных узлов зависит от правильности определения границ. В худшем случае такие запросы проверяют все узлы за время  $O(n)$ .

В запросе  $k$ -NN используется максимальная куча из  $k$  ближайших узлов и универсальная вспомогательная функция, которая находит расстояние до  $k$ -го ближайшего элемента, что является минимумом кучи:

```

template<typename NODE> struct QNode
{
    NODE* n;
    double d;
    bool operator<(QNode const& rhs) const { return d > rhs.d; }
    static double dHeap(Heap<QNode>& heap, int k)
    {
        return heap.getSize() < k ?
            numeric_limits<double>::max() : heap.getMin().d;
    }
};

```

Запрос  $k$ -NN выполняет разветвление и ограничение дерева, задействуя кучу для сохранения  $k$  ближайших найденных соседей. При обрезке и выборе дочерних элементов используются границы самого дальнего в текущий момент узла. Найденные соседи сортируются в куче в порядке близости:

```

Vector<NodeType*> kNN(KEY const& key, int k) const
{
    Heap<HEAP_ITEM> heap;
    kNN(root, key, heap, k);
    Vector<Node*> result; // найденные узлы, отсортированные
                        // в куче по расстоянию
    while(!heap.isEmpty()) result.append(heap.deleteMin().n);
    result.reverse();
    return result;
}

NodeType* nearestNeighbor(KEY const& key) const
{
    assert(!isEmpty());
    return kNN(key, 1)[0];
}

```

```

typedef QNode<Node> HEAP_ITEM;
void kNN(Node* n, KEY const& key, Heap<HEAP_ITEM>& heap, int k) const
{
    if(!n) return;
    // замена самого дальнего n в куче текущим n, если оно ближе
    HEAP_ITEM x = {n, lowerBound(key, n->key)};
    if(heap.getSize() < k) heap.insert(x);
    else if(x.d < HEAP_ITEM::dHeap(heap, k)) heap.changeKey(0, x);
    // сперва расширяется ближайший потомок
    double lb = n->leftChildBound(x.d), rb = n->rightChildBound(x.d);
    Node* l = n->left, *r = n->right;
    if(lb > rb) // переход к узлу с наименьшей границей выполняется раньше,
    // чтобы снизить риск перехода к другому узлу путем помещения
    // в кучу более близких узлов
        swap(lb, rb);
        swap(l, r);
    }
    if(lb <= HEAP_ITEM::dHeap(heap, k)) kNN(l, key, heap, k);
    if(rb <= HEAP_ITEM::dHeap(heap, k)) kNN(r, key, heap, k);
}

```

Чтобы расширить VP-дерево во внешнюю память, нужно в каждом узле разместить больше ключей и указателей, отсортированных по расстояниям до родительского узла.

## 18.4. k-d-дерево

Допускающие операцию сравнения точки в каждом измерении поддерживают *запрос диапазона*, который находит все элементы, чьи ключи для указанных измерений находятся в заданном диапазоне. *k-d-дерево* — это бинарное дерево, которое поочередно разветвляется в каждом измерении. Некоторые (но не все) ключи различных измерений могут быть равными (рис. 18.2).

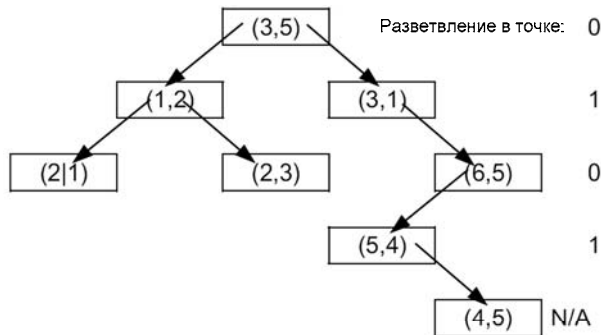


Рис. 18.2. Структура двумерного *k-d-дерева* с евклидовыми данными

```

template<typename KEY, typename VALUE,
        typename INDEXED_COMPARATOR = LexicographicComparator<KEY>> class KDTree
{
    INDEXED_COMPARATOR c;

```

```

struct Node
{
    KEY key;
    VALUE value;
    Node *left, *right;
    Node(KEY const& theKey, VALUE const& theValue): key(theKey),
        value(theValue), left(0), right(0) {}
}* root;
Freelist<Node> f;
int D;
Node* constructFrom(Node* n)
{
    Node* tree = 0;
    if(n)
    {
        tree = new(f.allocate())Node(n->key, n->value);
        tree->left = constructFrom(n->left);
        tree->right = constructFrom(n->right);
    }
    return tree;
}
public:
    typedef Node NodeType;
    bool isEmpty()const{return !root;}
    KDTree(int theD, INDEXED_COMPARATOR theC = INDEXED_COMPARATOR()): root(0),
        c(theC), D(theD) {}
    KDTree(KDTree const& rhs): c(rhs.c){root = constructFrom(rhs.root);}
    KDTree& operator=(KDTree const& rhs){return genericAssign(*this, rhs);}
};

```

Поиск и вставка выполняются аналогично несбалансированному бинарному дереву поиска, за исключением циклического прохода по измерениям. Многоразовый `findPointer` также вычисляет родителя найденного узла для запроса 1-NN:

```

Node** findPointer(KEY const& key, Node*& parent)const
{
    Node* n, **pointer = (Node**)&root; // преобразование в константу
    parent = 0;
    for(int i = 0; (n = *pointer) && !c.isEqual(key, n->key);
        i = (i + 1) % D)
    {
        parent = n;
        pointer = &(c(key, n->key, i) ?
            n->left : n->right);
    }
    return pointer;
}
VALUE* find(KEY const& key)const
{
    Node *n = *findPointer(key, n);
    return n ? &n->value : 0;
}

```

```

void insert(KEY const& key, VALUE const& value)
{
    Node *dummy, **pointer = findPointer(key, dummy);
    if(*pointer) (*pointer)->value = value;
    else *pointer = new(f.allocate())Node(key, value);
}

```

Для VP- и  $k$ -d-деревьев, элементы которых вставляются в случайном порядке,  $E[\text{высота}] = O(\lg(n))$ . Высота в худшем случае равна  $O(n)$ . Обычно это не проблема, потому что, если значения точек в одном измерении расположены в случайном порядке, а  $D$  — это константа относительно  $n$ ,  $E[\text{высота}] = O(\lg(n))$ . Лучшей метрикой производительности является эффективность запроса по каждому указанному параметру. Например, для двумерного  $k$ -d-дерева, сбалансированного по координате  $x$ , а не по координате  $y$ , запрос диапазона по  $(x, y)$  аналогичен линейному сканированию  $y$  по результату запроса диапазона по  $x$ . Вращения не поддерживаются, но операция вставки может частично перестраиваться для поддержания баланса с использованием амортизированного или ожидаемого времени  $O(\lg(n)^2)$  (см. главу 12. *Разные алгоритмы и методы*). Структура может поддерживать удаление, но оно работает неэффективно, поэтому при необходимости лучше использовать слабые удаления.

Запрос диапазона выполняет поиск в глубину, который проверяет, находятся ли рассматриваемые узлы в диапазоне  $[l, u]$ . Если узел не попадает в диапазон, то его дочерние элементы тоже не попадают, что позволяет отсечь часть дерева:

```

Vector<NodeType*> rangeQuery(KEY const& l, KEY const& u,
    Vector<bool> const& dimensions) const
{
    Vector<Node*> result;
    rangeQuery(l, u, dimensions, result, root, 0);
    return result;
}

void rangeQuery(KEY const& l, KEY const& u, Vector<bool> const& dimensions,
    Vector<Node*>& result, Node* n, int i) const
{
    if(!n) return;
    bool inRange = true; // проверка, попадает ли текущий узел в диапазон
    for(int j = 0; j < D; ++j)
        if(dimensions[j] && (c(n->key, l, j) ||
            c(u, n->key, i))) inRange = false;
    if(inRange) result.append(n);
    int j = (i + 1) % D; // проверка исключительно диапазона для желаемых измерений
    if(!(dimensions[i] && c(n->key, l, i)))
        rangeQuery(l, u, dimensions, result, n->left, j);
    if(!(dimensions[i] && c(u, n->key, i)))
        rangeQuery(l, u, dimensions, result, n->right, j);
}

```

Время выполнения у сбалансированного дерева в худшем случае составляет  $O(n^{(D-1)/D})$ , что оптимально для пространства  $O(n)$ . В среднем для малых  $D$  время выполнения составляет  $O(\lg(n))$  (см. [18.5]). Интуитивно ясно, что  $k$ -d-дерево полезно скорее для малых значений  $D$ , потому что у больших значений много переменных для разделения, каждая из которых используется лишь несколько раз.

*Запрос частичного соответствия* находит все узлы, ключ которых совпадает с заданным только в указанных измерениях. Этот запрос сводится к запросу диапазона, где  $l = u$ . *Интервальный запрос* (здесь не реализован) находит все интервалы, которые содержат заданную точку, в виде запроса 2D-диапазона  $[x, \max] \times [\min, x]$ , где значения  $\min/\max$  задаются явно или заменяются подходящими  $-\infty$  и  $\infty$ . Эта операция обобщается на прямоугольники и гиперпрямоугольники.

Запрос  $k$ -NN работает с метрическими расстояниями, но для эффективности дополнительно предполагает инкрементное вычисление расстояния, поэтому функция расстояния должна иметь вид  $\sum_{0 \leq i < d} ag(x[i], y[i])$ . Используется алгоритм В&В, аналогичный применяемому в VP-дереве, но в нем задействуются ненормализованные расстояния в форме  $d(x, y)$ , которые по мере выполнения поиска постепенно обновляются. Изначально частичный ключ =  $x$ , что соответствует нулевому расстоянию до дерева. Когда поиск идет влево, частичный ключ левого узла не меняется, а правый узел устанавливает свою текущую координату равной координате текущего узла. Симметрично при движении вправо частичный ключ правого узла не меняется, а левый узел устанавливает свою текущую координату равной координате текущего узла. Таким образом, наилучшая нижняя граница расстояния по иерархии равна  $d(x, \text{частичный ключ})$ . Это значение не уменьшается по мере выполнения поиска. Сравнение расстояний выполняется инкрементально;

```
typedef QNode<Node> HEAP_ITEM;
template<typename DISTANCE> void kNN(Node* n, KEY const& key,
    Heap<HEAP_ITEM>& heap, int k, int i, KEY& partial,
    double partialDistance, DISTANCE const& distance) const
{
    double best = HEAP_ITEM::dHeap(heap, k);
    if(n && partialDistance < best)
    { // обновление частичного расстояния
        double newPartialDistance = distance(key, n->key, i) -
            distance(key, partial, i);
        if(heap.getSize() < k)
        {
            HEAP_ITEM x = {n, distance(key, n->key)};
            heap.insert(x);
        }
        // использование нового частичного расстояния для проверки среза
        else if(newPartialDistance < best)
        { // инкрементное сравнение и вычисление
            double d = distance(best, key, n->key);
            if(d < best)
            {
                HEAP_ITEM x = {n, d};
                heap.changeKey(0, x);
            }
        }
        int j = (i + 1) % D;
        // обмен дочерних узлов для сохранения порядка
        Node *l = n->left, *r = n->right;
        if(!c(key, n->key, i)) swap(l, r);
        kNN(l, key, heap, k, j, partial, partialDistance, distance);
    }
```

```

        // установка частичного компонента равным компоненту n
        // n используется в качестве временного хранилища
        swap(partial[i], n->key[i]);
        kNN(r, key, heap, k, j, partial, newPartialDistance, distance);
        swap(partial[i], n->key[i]);
    }
}

```

Для  $k = 1$  размещение в очереди родителя, в которого будет вставлен запрос, позволяет отсечь больше данных. Но для  $k > 1$  так делать не стоит из-за возможной повторной вставки родителя в кучу, когда ее размер  $< k$ :

```

template<typename DISTANCE> Vector<NodeType*> kNN(KEY const& key, int k,
DISTANCE const& distance) const
{
    Heap<HEAP_ITEM> heap;
    KEY partial = key;
    kNN(root, key, heap, k, 0, partial, 0, distance);
    Vector<Node*> result; // сортировка узлов в куче по расстоянию
    while(!heap.isEmpty()) result.append(heap.deleteMin().n);
    result.reverse();
    return result;
}

template<typename DISTANCE> NodeType* nearestNeighbor(KEY const& key,
DISTANCE const& distance) const
{
    assert(!isEmpty());
    Node* parent, *result = *findPointer(key, parent);
    if(result) return result; // найден равный узел, d=0
    Heap<HEAP_ITEM> heap; // родитель помещается в кучу
    HEAP_ITEM x = {parent, distance(key, parent->key)};
    heap.insert(x);
    KEY partial = key;
    kNN(root, key, heap, 1, 0, partial, 0, distance);
    return heap.getMin().n;
}

```

Запрос расстояния аналогичен запросу  $k$ -NN. Создайте частичный ключ, который ограничивает все узлы в текущем поддереве. Он начинается с запроса  $x$  и обновляется при движении влево или вправо, что означает, что все потомки находятся хотя бы на некотором ненулевом частичном расстоянии от узла в зависимости от соответствующего измерения. Поскольку мы работаем с инкрементным расстоянием, частичный радиус равен квадрату желаемого радиуса для евклидова расстояния:

```

template<typename DISTANCE> Vector<NodeType*> distanceQuery(KEY const& x,
double partialRadius, DISTANCE const& distanceIncremental) const
{
    Vector<Node*> result;
    KEY partial = x;
    distanceQuery(x, partialRadius, result, root, 0, distanceIncremental,
        partial, 0);
    return result;
}

```



```

template<typename DISTANCE> void distanceQuery(KEY const& x,
    double partialRadius, Vector<Node*>& result, Node* n, int i,
    DISTANCE const& distanceIncremental, KEY& partial,
    double partialDistance) const
{ // попытка обрезать поддерево
    if(!n || partialDistance > partialRadius) return;
    if(distanceIncremental(n->key, x) <= partialRadius)
        result.append(n);
    i = (i + 1) % D;
    Node* nodes[] = {n->left, n->right};
    for(int j = 0; j < 2; ++j)
    { // вычисление частичного расстояния правого поддерева,
      // если x[i] находится слева от n, и наоборот.
      // Если узлы оказались равны - не страшно
      bool applyPartial = c(x, n->key, i) == (j == 1);
      double dDelta = 0;
      if(applyPartial)
      {
          dDelta = distanceIncremental(x, n->key, i) -
              distanceIncremental(x, partial, i);
          swap(partial[i], n->key[i]); // использование n в качестве
                                      // временного хранилища
      }
      distanceQuery(x, partialRadius, result, nodes[j], i,
          distanceIncremental, partial, partialDistance + dDelta);
      if(applyPartial) swap(partial[i], n->key[i]);
    }
}

```

При работе с внешней памятью требуются узлы размером  $B$  с  $k - 1$  ключами и  $k$  указателями, которые нужно дозаполнить отказа и вставить любой новый узел между соответствующими ключами.

## 18.5. Решение задач при большом числе измерений

Запросы  $k$ -NN и запросы диапазона работают медленнее при больших  $D$ , потому что для любого метрического расстояния, когда число измерений стремится к бесконечности, разница между расстояниями до ближайших и самых дальних соседей любого объекта стремится к нулю, — поэтому граничные значения не позволяют обрезать часть дерева, заставляя запросы рассматривать почти каждый узел.

Вы можете сократить время выполнения, получая при этом приближенные ответы. Для этого перед проверкой на отсеечение умножьте нижние границы на  $(1 + c)$ , где  $c$  — некоторая небольшая константа. Тогда найденные соседи окажутся не более чем в  $(1 + c)$  раз дальше от ближайшего. Но для реализации и практического использования неясно, как выбрать значение  $c$ , какая именно производительность за счет этого достигается и какая структура данных лучше всего подходит для работы.

## 18.6. Структуры данных для геометрических объектов

Самое простое решение состоит в том, чтобы создать вокруг объекта ограничивающую рамку и хранить его конечные точки в  $k$ -d-дереве. Например, интервалы или прямоугольники можно представить как многомерные точки.

Куб можно представить по его центроиде, сохраняя конечные точки в элементе узла. Это делает запросы типа  $k$ -NN более эффективными.

Как вариант, объекты можно представлять в виде изображений, где каждая ячейка содержит список указателей на объекты, которые ее касаются, и имеет не более одного указателя, при условии отсутствия объектов.

## 18.7. Точки

*Многомерную точку* можно представить в виде вектора, но обычно  $D$  известно уже во время компиляции, поэтому более эффективно использовать массив. Алгоритмы должны уметь работать и с тем и с другим, а также с евклидовым расстоянием и с деревьями. Организованная таким образом точка удобно реализует несколько арифметических операторов, используя обычную векторную арифметику:

```
template<typename KEY, int D = 2>
class Point: public ArithmeticType<Point<KEY, D> >
{
    KEY x[D];
public:
    static int const d = D;
    KEY& operator[](int i){assert(i >= 0 && i < D); return x[i];}
    KEY const& operator[](int i)const{assert(i >= 0 && i < D); return x[i];}
    int getSize()const{return D;}
    Point(){for(int i = 0; i < D; ++i) x[i] = 0;}
    Point(KEY const& x0, KEY const& x1)
    {
        assert(D == 2); // чтобы избежать проблем для случая D > 2
        x[0] = x0;
        x[1] = x1;
    }
    bool operator==(Point const& rhs)const
    {
        for(int i = 0; i < D; ++i) if(x[i] != rhs.x[i]) return false;
        return true;
    }
    Point& operator+=(Point const& rhs)
    {
        for(int i = 0; i < D; ++i) x[i] += rhs.x[i];
        return *this;
    }
    Point& operator*=(double scalar)
    {
        for(int i = 0; i < D; ++i) x[i] *= scalar;
```

```

        return *this;
    }
    friend Point operator*(Point const& point, double scalar)
    {
        Point result = point;
        return result *= scalar;
    }
    Point& operator+=(Point const& rhs){return *this += rhs * -1;}
    Point operator-(){return *this * -1;}
    double friend dotProduct(Point const& a, Point const& b)
    {
        double dp = 0;
        for(int i = 0; i < D; ++i) dp += a[i] * b[i];
        return dp;
    }
};
typedef Point<double> Point2;

```

## 18.8. Геометрические примитивы

Знак площади треугольника, заданной тремя двумерными точками, позволяет узнать, поворачивают ли точки налево, — т. е. идут ли они против часовой стрелки (CCW). Когда площадь = 0, точки не поворачиваются, но обычно этот случай считается левым поворотом. Площадь треугольника  $(a, b, c)$  против часовой стрелки равна

$$\det \begin{pmatrix} 1 & a.x & a.y \\ 1 & b.x & b.y \\ 1 & c.x & c.y \end{pmatrix} :$$

```

double triangleArea(Point2 const& a, Point2 const& b, Point2 const& c)
{return (b[0] - a[0]) * (c[1] - a[1]) - (b[1] - a[1]) * (c[0] - a[0]);}
bool ccw(Point2 const& a, Point2 const& b, Point2 const& c)
{return triangleArea(a, b, c) >= 0;} // истинно, если точки поворачиваются влево

```

Результат может оказаться неправильным для очень тонких треугольников из-за численных округлений при вычитании. Тип `double` имеет точность 52 бита, вычитание не меняет требуемой точности, а умножение удваивает ее, поэтому, если каждый член имеет  $\leq 26$  битов, результат будет правильным. В противном случае, чтобы избежать числовых ошибок, можно использовать рациональные числа:

```

Rational robustTriangleArea(Point2 const& a, Point2 const& b, Point2 const& c)
{
    return (Rational(b[0]) - Rational(a[0])) * (Rational(c[1]) -
        Rational(a[1])) - (Rational(b[1]) - Rational(a[1])) * (Rational(c[0])
        - Rational(a[0]));
}
bool robustCcw(Point2 const& a, Point2 const& b, Point2 const& c)
{return !robustTriangleArea(a, b, c).isMinus();}

```

## 18.9. Выпуклая оболочка

Для набора двумерных точек *выпуклая оболочка* — это такое их подмножество, что наложенная на него «резинка» охватывает все множество (рис. 18.3).

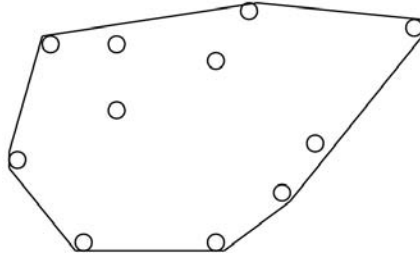


Рис. 18.3. Множество точек и его выпуклая оболочка

Алгоритм вычисления оболочки:

1. Отсортировать точки по координате  $x$ , разрывая связи по координате  $y$ .
2. Поместить первые две точки на оболочку.
3. Вычислить верхнюю часть оболочки. Для любой другой точки:
4. Поместить ее на оболочку.
5. Если последние три точки образуют левый поворот, убрать предпоследнюю точку.
6. С остальными точками сделать то же самое и удалить дубликат последней добавленной точки, чтобы вычислить нижнюю часть оболочки.

```
void processPoint(Vector<Point2>& hull, Point2 const& point)
{
    hull.append(point);
    while(hull.getSize() > 2 && ccw(hull[hull.getSize() - 3],
        hull[hull.getSize() - 2], hull[hull.getSize() - 1]))
    {
        hull[hull.getSize() - 2] = hull[hull.getSize() - 1];
        hull.removeLast();
    }
}

Vector<Point2> convexHull(Vector<Point2>& points)
{
    assert(points.getSize() > 2);
    quickSort(points.getArray(), 0, points.getSize() - 1,
        LexicographicComparator<Point2>());
    // верхняя часть оболочки
    Vector<Point2> result;
    result.append(points[0]); // инициализация первыми двумя точками
    result.append(points[1]);
    for(int i = 2; i < points.getSize(); ++i) processPoint(result, points[i]);
    // нижняя часть оболочки, удаление самой левой точки, добавленной дважды
    for(int i = points.getSize() - 2; i >= 0; --i)
        processPoint(result, points[i]);
}
```

```

    result.removeLast();
    return result;
}

```

При использовании арифметики с плавающей точкой точки на линии могут оказаться не с той стороны, и слишком резкий левый поворот может превратиться в правый поворот, что разрушает топологическую структуру. Чтобы избежать этого, можно использовать рациональную арифметику (здесь не реализовано — см. *разд. «Советы по дополнительной подготовке»*). Время выполнения составляет  $O(n \lg(n))$ , но узким местом являются выполняемые за  $O(n)$  проверки поворота, в которых много постоянных коэффициентов. Для случайных точек  $E[\text{размер корпуса}] = O(\lg(n))$  (см. [18.1]).

## 18.10. Развертка плоскости

Эта техника позволяет решить многие задачи. Например, при вычислении выпуклой оболочки можно отсортировать точки по координате  $x$ , обработать их слева направо, а затем справа налево. Каждая точка представляет собой *событие*, обработка которого задает некоторый инвариант в пространстве до следующего события.

Например, найдем площади возможно перекрывающихся зданий на изображении, каждое из которых представлено двумя верхними координатами (рис. 18.4).

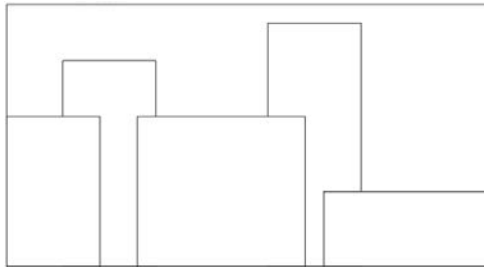


Рис. 18.4. Пример задачи о застройке

Простое решение — выполнить сортировку по координате  $x$  и обрабатывать точки слева направо, сохраняя текущую высоту. При обработке точки события увеличивайте общий объем на текущую высоту  $\times$  расстояние до предыдущего события. Чтобы отслеживать текущую максимальную высоту, используйте карту «открытых» зданий. Здание открывается, когда встречается его левая координата, и закрывается, когда встречается правая. Решение на основе STL может выглядеть так:

```

struct RoofCorner
{
    double x, y;
    bool isLeft;
    RoofCorner(double theX, double theY, bool theIsLeft): x(theX), y(theY),
        isLeft(theIsLeft){} // сравнение по x позволяет выполнять перебор
                           // слева направо
    bool operator<(RoofCorner const& rhs) const
    {return x == rhs.x ? isLeft > rhs.isLeft : x < rhs.x;}
};

```

```

double buildingArea(vector<RoofCorner> corners)
{
    sort(corners.begin(), corners.end());
    double result = 0, currentHeight = 0, lastX = corners[0].x;
    map<double, int> openBuildings; // индексирование и сортировка по высоте.
    // Здесь не возникает численных проблем, так как никаких вычислений
    // не выполняется.
    for(unsigned int i = 0; i < corners.size(); ++i)
    {
        double x = corners[i].x, y = corners[i].y;
        result += currentHeight * (x - lastX);
        lastX = x;
        // подсчет открытых зданий
        openBuildings[y] += corners[i].isLeft ? 1 : -1;
        if(openBuildings[y] == 0) openBuildings.erase(y);
        // текущая высота соответствует самому высокому зданию
        currentHeight = openBuildings.size() > 0 ?
            openBuildings.rbegin()->first : 0;
    }
    return result;
}

void testBuildingArea()
{
    vector<RoofCorner> points;
    points.push_back(RoofCorner(0, 1, true));
    points.push_back(RoofCorner(2, 1, false));
    points.push_back(RoofCorner(1, 2, true));
    points.push_back(RoofCorner(3, 2, false));
    assert(buildingArea(points) == 5);
}

```

Это сложный вопрос для собеседования, над которым многие разработчики мучаются довольно долго. На правильное решение потребуется не менее часа, поэтому лучше такую задачу не задавать.

## 18.11. Комментарии

Многомерные структуры данных и вычислительная геометрия — это обширные темы. В многомерных структурах основная проблема заключается в отсутствии балансировки и низкой производительности при больших  $D$ . На эту тему было предложено много других алгоритмов — см. работу [18.5].

VR-дерево в других источниках реализовано так, что пользователь сам задает расстояния между узлами. Идея использования расстояния до первого вставленного узла кажется оригинальной. Она правильно масштабируется для различных расстояний и хорошо показала себя в экспериментах по машинному обучению (см. главу 26. *Машинное обучение: классификация*).

Для поиска ближайшего соседа в многомерных данных можно использовать *локально-чувствительное хеширование* (Locality-Sensitive Hashing, LSH) (см. [18.5]). Несмотря на многообещающую идею, неясно, как настроить его так, чтобы оно работало как «чер-

ный ящик» даже для функций расстояния с известными хорошими хеш-функциями — такими как евклидова (представленная в некоторых других источниках). Мои попытки использовать его в задачах машинного обучения не увенчались успехом (см. *разд. «Комментарии»* в главе 26. *Машинное обучение: классификация*). Этому хешированию также нужны некоторые параметры, выходящие за предел точности  $(1 + \epsilon)$ , которые не имеют логического смысла и иногда вообще не возвращают ближайшего соседа.

В области вычислительной геометрии основной проблемой является организация надежных и эффективных вычислений. Использование арифметики больших чисел — это простое, но неэффективное решение. Более эффективным, но гораздо более сложным вариантом является использование *фильтров с плавающей точкой* (см. [18.3]). Лучше работать в формате с плавающей точкой с ограничениями ошибок и переключаться на арифметику больших чисел только в случаях, когда исключить ошибки не удастся.

В работе [18.1] приведено множество других алгоритмов. Возможно, в этой главе стоило рассмотреть алгоритмы триангуляции, результаты которых полезны при решении уравнений в частных производных методами конечных элементов (см. *разд. «Комментарии»* в главе 23. *Численные алгоритмы: работа с функциями*). Но для полезной реализации такого алгоритма нужна надежная арифметика. Вычислительная геометрия является скорее математической, чем вычислительной, как показано в недавней теоретической книге [18.2]. С компьютерной графикой она почти никак не связана.

## 18.12. Советы по дополнительной подготовке

- ◆ Реализуйте интервальный запрос для  $k$ -d-дерева.
- ◆ Реализуйте интервальную арифметику (см. *разд. «Комментарии»* в главе 22. *Численные алгоритмы: введение и матричная алгебра*) и выполните с ее помощью вычисления выпуклых оболочек. Если направление поворота неясно, можно считать точку частью прямой, поэтому точная арифметика не нужна. Множество специализированных алгоритмов описаны в работе [18.4].

## 18.13. Список рекомендуемой литературы

- 18.1. de Berg M., Cheong O., & Van Kreveld M., Overmars, M. (2008). *Computational Geometry: Algorithms and Applications*. Springer.
- 18.2. Goodman J. E., O'Rourke J., & Toth C. D. (2017). *Handbook of Discrete and Computational Geometry*. CRC.
- 18.3. Mehlhorn K., & Näher S. (1999). *LEDA: a Platform for Combinatorial and Geometric Computing*. Cambridge University Press.
- 18.4. Ratschek H., & Rokne J. (2003). *Geometric Computations with Interval and New Robust Methods: Applications in Computer Graphics, GIS and Computational Geometry*. Horwood.
- 18.5. Samet H. (2006). *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann.

# 19. Обнаружение и исправление ошибок

## 19.1. Введение

Иногда, например, во время сетевой маршрутизации, в передаваемых данных возникают ошибки. В этих случаях для их обнаружения и исправления стоит использовать специальное кодирование. То же самое касается и файлов, которые хранятся на CD, DVD и других носителях. В этой главе мы обсудим только основные алгоритмы, но существуют и многие другие, входящие в различные стандарты. Дополнительные сведения приведены в упоминаемой здесь литературе.

## 19.2. Бинарные полиномы

В качестве математической модели для некоторых из обсуждаемых алгоритмов удобно задействовать *арифметику с бинарными полиномами*. Ее эффективная реализация является необходимой частью алгоритма.

Битовая строка с установленным старшим битом  $i$  представляет собой полином с двоичными коэффициентами степени  $i + 1$ . Например, байт 00001101 представляет полином  $x^3 + x^2 + 1$ . Многословная арифметика с двоичными полиномами более эффективна, чем с числами в определенной основе, из-за отсутствия необходимости работы с переносами (табл. 19.1).

**Таблица 19.1.** Реализация операций на бинарных полиномах  $p$  и  $q$

Выражение	Реализация
$p + q$ или $p - q$	$p \wedge q$
$px$	$p \ll 1$
$p / x$	$p \gg 1$ с остатком $p$ и 1
$pq$	Почленное умножение, как с числами
$pp$	Возведение каждого члена в квадрат, потому что перекрестные члены сокращаются
$p / q$ и $p \% q$	Длинное деление путем многократного вычитания наибольшего $qx^m < p$ из $p$

## 19.3. Многочлены над конечными полями

*Поле* — это абстрактная алгебраическая структура, где для любого элемента существует аддитивный обратный, а для любого ненулевого элемента мультипликативный обратный элемент — например, вещественное число. *Конечное поле* имеет конечное



число элементов. Например, целые числа, кратные  $p$ , для простого числа  $p$ . Весьма полезными являются многочлены с коэффициентами из конечного поля — например, бинарные многочлены.

Для реализации удобно иметь конечные поля с  $2^m$  элементами, особенно если  $m = 1$  (двоичные многочлены) или 8 (об этом далее в главе), что соответствует битам и байтам соответственно.

*Неприводимый многочлен* степени  $m$  не делится ни на один другой многочлен меньшей степени. *Примитивный многочлен* — это неприводимый многочлен, который делится на  $x^n + 1$  при  $n = 2^m - 1$ , но не при меньшем  $n$  (см. [19.5]). Арифметические действия с бинарными многочленами выполняются как обычно, но конкретный примитивный многочлен модифицируется. При  $m = 1$  один такой многочлен равен 1011 ( $x^3 + x + 1$ ). Примитивный многочлен среди многочленов аналогичен простым числам среди чисел.

## 19.4. Обнаружение ошибок

Представим, что при отправке данных через сеть в битах появляются ошибки. Добавление  $k$  битов избыточности в сообщение  $m$  позволяет обнаружить ошибку с вероятностью  $1 - 2^{-k}$ , поскольку случайное сообщение содержит случайные биты. Опишем подходы к реализации такого кодирования:

- ♦ интерпретировать  $m$  как большое число и найти остаток от деления его на  $k$ -битное простое число. Это дает почти такую же вероятность, но работает неэффективно;
- ♦ использовать универсальную хеш-функцию, приведенную в теореме в главе 9. *Хеширование*. Она более эффективна, но дает худшую вероятность восстановления. Однако использование нескольких хешей сводит вероятность отказов к минимуму.

Для удобства предположим, что мы работаем с  $k$ -битными словами. На практике и в приведенной здесь реализации возьмем  $k = 32$ . Таким образом, количество избыточных битов для пользователя фиксировано. Алгоритм CRC добавляет к сообщению  $k$  нулей и возвращает  $k$ -битовый остаток от двоичного полиномиального деления результата на выбранный  $(k + 1)$ -битный полином. Хотя для алгоритма это и не обязательно, полином должен быть примитивным, чтобы значение остатка зависело от всех битов данных. Реализация игнорирует  $(k + 1)$ -й бит полинома, равный единице, и использует  $p =$  его  $k$  младших битов (рис. 19.1):

1. Прибавить  $k$  нулей к  $m$ .
2.  $w = k$  старших битов  $m$ .

$m$		Дополнение	Бит		$w$
$p$	1	1	0	0	0
	1	0	0	1	
	1	0	1	0	
	1	0	0	1	
			0	1	1
			1	1	0
			1	0	1
			0	1	1

Рис. 19.1. Длинное деление при  $m = 11$ ,  $p = 1001$ ,  $k = 3$  и соответствующем алгоритме расчетов

3. Пока в  $m$  еще есть биты:

4.  $w = (w \ll 1) \wedge$  следующий бит  $m$ .

5. Если старший бит предыдущего  $w == 1$ ,  $w \wedge= p$ .

6. Вернуть  $w$ .

Поскольку последние  $k$ -битов равны нулю,  $m$  битов влияют на  $w$  только как *управляющие биты* для решений xor-with- $p$ -or-not. Поэтому загружаем  $m$  только в управляющие биты, а не в  $w$ . Это избавляет от необходимости дополнять  $m$   $k$ -нулями:

1.  $w = 0$ .

2. Пока в  $m$  еще есть биты:

3.  $w = w (w \ll 1)$ .

4. Если старший бит предыдущего  $w \wedge$  следующий бит  $m == 1$ ,  $w \wedge= p$ .

5. Вернуть  $w$ .

Примерные результаты могут выглядеть так:

Бит	$w$ после прохода цикла
N/A	'000
1	'001
1	'011

Обрабатывать байты — более эффективный подход. Следующие 8 управляющих битов и  $p$  производят следующие 8 решений xor-with- $p$ -or-not.  $\text{MSD}_8(m) \wedge \text{MSD}_8(w)$  будет управляющим байтом, за исключением того, что для работы с ним необходимо настроить элементы управления для  $p$ . Предыдущий алгоритм эффективно работает при  $c = w \wedge m$ , причем алгоритм масштабируется до нужного размера. Кроме того,  $c = w \wedge (0 \wedge m)$ , поэтому для предварительного расчета можно принять  $w = 0$  и запустить предыдущий алгоритм на  $0_{\text{extended}} \wedge m_{\text{extended}}$ . Поскольку операция хог является кумулятивной, для данного  $p$  и любого числа байтов предварительно нужно вычислить операцию хог в одну константу и использовать управляющие байты:

1. Вычисление констант.

2.  $w = 0$ .

3. Пока в  $m$  еще есть биты:

4.  $w = w \ll 8$ .

5.  $w \wedge=$  константа[самый значащий байт предыдущего  $w \wedge$  следующий байт  $m$ ].

6. Вернуть  $w$ .

В 32-битной реализации можно использовать любое число  $p$ , что дает  $\text{Pr}(\text{обнаружения ошибки}) = 1 - 2^{-k}$ , но значение 0xFA567D89 позволяет обнаруживать все ошибки с определенным расстоянием Хэмминга (см. [19.3]). Расчет может выполняться в реальном времени за время  $O(n)$ :

```
class CRC32
{
    uint32_t polynomial, constant[256];
```

```

public:
    CRC32(uint32_t thePolynomial = 0xFA567D89u):polynomial(thePolynomial)
    {
        for(int i = 0; i < 256; ++i)
        {
            constant[i] = i << 24; // расширенное значение c
            for(int j = 0; j < 8; ++j) constant[i] =
                (constant[i] << 1) ^ (constant[i] >> 31 ? polynomial : 0);
        }
    }
    uint32_t hash(unsigned char* array, int size, uint32_t crc = 0)
    {
        assert(numeric_limits<char>::digits == 8);
        for(int i = 0; i < size; ++i)
            crc = (crc << 8) ^ constant[(crc >> 24) ^ array[i]];
        return crc;
    }
};

```

## 19.5. Каналы и коды

*Двоичный симметричный канал* (BSC, Binary Symmetric Channel) обращает каждый  $m$  бит с вероятностью  $p$ . Если  $p \neq 1/2$ , канал выдает случайные биты, а значение  $p > 1/2$  эквивалентно  $p < 1/2$ , обратному ему. Количество перевернутых битов в кодовом слове называется *расстоянием Хэмминга* между ним и принятым словом. Для небинарного алфавита расстояние равно количеству измененных символов.

Некоторые ошибки можно исправить, используя избыточность, как для обнаружения ошибок. В *коде*  $(n, k)$  используются  $n$ -символьные кодовые слова для  $k$ -символьных сообщений, где  $n \geq k$ . *Скорость* кода:  $R = k/n$ . Коды почти всегда блочные, и для повышения эффективности в них кодируются/декодируются большие блоки размером  $n$ . Один из простых вариантов кода — повторение: скопируйте каждый бит  $n - 1$  раз, что дает код  $(n, 1)$ .

*Пропускная способность*  $C$  канала равна объему взаимной информации между входом и выходом. Например, для алгоритма BSC значение  $C = 1 - H(p)$ , где  $H$  — бинарная энтропия ( $H(p) = -p \lg(p) - (1 - p) \lg(1 - p)$ ) (см. главу 15. *Сжатие*). В худшем случае  $p = 1/2$ ,  $H(p) = 0$ , и связь невозможна.

*Теорема кодирования зашумленного канала* (см. [19.4]): при  $R \geq C$  надежное исправление ошибок невозможно. В противном случае для достаточно больших значений  $n$  существуют коды, допускающие коррекцию ошибок с произвольно малой вероятностью невозможности декодирования.

Необходимо выбрать значение  $R < C$ . Обычно с помощью моделирования легко оценить значение  $C$  для рассматриваемой модели канала. Еще одна важная идея (из доказательства) заключается в том, что нужная производительность легко достигается случайными кодами, хотя декодирование случайных кодов требует больше вычислительных ресурсов. Для этого необходимо найти кодовое слово, ближайшее к полученному слову (путем измерения расстояния Хэмминга), что занимает экспоненциальное по  $n$  время, если дополнительных знаний о коде у пользователя нет. Таким образом, ис-

пользуемые на практике коды имеют некоторую структуру, которая позволяет эффективно выполнять декодирование, но не достигает желаемой эффективности.

Самый простой способ исправить ошибку — обнаружить ее с помощью CRC и повторно отправить поврежденные сообщения получателю. Этот способ хорошо работает только для каналов с большой пропускной способностью и с малым значением  $p$ , так что  $E$  [количество повторно отправленных сообщений] оказывается невелико. Но вот царапины на поверхности DVD-диска вносят ошибки, которые повторная отправка исправить не поможет, а значение  $C$  может быть низким, поэтому требуется использовать соответствующие корректирующие коды.

Существуют две распространенные стратегии декодирования:

- ◆ *наихудший случай* — предположим, что расстояние Хэмминга между любыми двумя кодовыми словами  $\geq d$  для некоторого  $d$ . Любое полученное слово становится ближайшим кодовым словом, поэтому вы можете исправить  $\leq \frac{d-1}{2}$  ошибок;
- ◆ *вероятностная* — найдем наиболее вероятное кодовое слово. Некоторые коды могут исправить гораздо больше ошибок, чем предлагает  $d$ , но не дают никаких гарантий успешного декодирования конкретного слова.

Для логики ближайшего кодового слова алгоритма BSC:

$$\begin{aligned} \Pr(\text{неудачное декодирование}) &\leq \Pr(\text{ошибка повреждает} > \frac{d-1}{2} \text{ символов}) = \\ &= 1 - \text{BinominalCDF}\left(\frac{d-1}{2}, n, p\right) \quad (\text{см. [14.9]}). \end{aligned}$$

Если  $pn < \frac{d-1}{2}$ , то при использовании нормальной аппроксимации и ограничении хвоста (см. главу 21. *Вычислительная статистика*) вероятность экспоненциально мала  $\left(\frac{d-1}{2}\right)^2$  (см. главу 21. *Вычислительная статистика*).

Логика ближайшего кодового слова является частным случаем максимальной вероятности, когда все кодовые слова равновероятны. Обычно существует возможность декодирования при большем количестве ошибок, чем  $d$ , потому что ожидаемое расстояние намного меньше, чем  $d$  в наихудшем случае. Но обычно для измерения производительности используется моделирование, которое может быть очень медленным, особенно при низком  $R$ , т. к. в этом случае требуется большее значение  $n$  и больше вычислительных ресурсов.

Оба способа могут ошибаться, даже если кодовое слово успешно найдено, т. к. настоящее кодовое слово может оказаться другим. Подобные ситуации лучше всего обнаруживаются путем добавления в коды эффективной логики обнаружения ошибок — например, CRC в каждом блоке, что увеличивает уверенность правильного исправления, а не просто коррекции.

*Одиночная граница:* расстояние  $d$  между кодовыми словами кода  $(n, k)$  удовлетворяет условию  $d \leq n - k + 1$ . Доказательство: два  $(k, k)$  кодовых слова отличаются  $\geq 1$  символом, и добавление еще  $n - k$  символов дает  $d \leq n - k + 1$ .

Код  $(n, k)$  является *линейным*, если кодирование эквивалентно интерпретации числа  $m$  размера  $k$  как вектора символов и умножению его на матрицу генератора кода  $G$  размера  $k \times n$  с использованием соответствующей арифметики. Код является *систематическим*, если первая подматрица  $k \times k$  матрицы  $G$  равна единице. Для систематических кодов сообщение является частью кодового слова, а остальная часть фактически представляет собой биты проверки четности, — это позволяет повысить эффективность и сохранить только часть  $(n - k) \times k$ . Практические коды линейны и систематичны.

Эффективность вычислений имеет важное значение, т. к. если кодирование/декодирование выполняется быстрее, то и передача данных ускоряется. Влияние скорости кодирования может перевесить изменения значения  $R$ . Кроме того, поскольку коды, как правило, реализуются на аппаратном уровне, обычно предпочтение отдается кодам с аппаратно-дружественными реализациями и возможностью параллелизма.

Эффективно использовать код с исправлением ошибок можно с помощью обновления, т. е. периодического декодирования и кодирования данных. Это позволяет исправить битовые ошибки сразу после их появления, но до чрезмерного накопления.

## 19.6. Вычисление конечного поля

В обычной арифметике не существует поля с  $m = 8$  или другим непростым значением. Для этих случаев нужны другие способы построения поля. Бинарные многочлены степени  $< m$  позволяют построить *поле Галуа*, называемое  $GF(2, m)$ . Для значений  $m \leq$  размера слова такие многочлены наиболее эффективно представляются целыми числами. Использовать бинарные полиномы как числа или даже как коэффициенты для других полиномов — это необычно. Они не являются числами, т. к. не отображаются непосредственно на диапазон  $[0, 2^m - 1]$ , но количественно их столько же, и среди них есть эквиваленты 0 (аддитивное тождественное значение) и 1 (мультипликативное тождественное значение). Поведение таких многочленов ничем не отличается от использования нормальных многочленов с простыми коэффициентами. Но вместо того, чтобы работать с наибольшим простым числом, которое соответствует заданному слову, можно работать со всем диапазоном слов и сохранять свойства многочленов с полевыми коэффициентами.

В отличие от простых чисел, арифметика полей в операциях умножения и инверсии неэффективна. Но из абстрактной алгебры известно, что каждый ненулевой элемент является степенью некоторого элемента, который не равен 0 или 1, например  $x$  (2 в поле), и обычно называется *альфа-элементом*. Поскольку число  $m$ , как правило, невелико, предварительно вычисленные таблицы сопоставляются со степенным представлением и обратно.

Для вычисления таблиц при данном  $a = x$  и примитивном многочлене  $p$  первый элемент равен 0, а следующие  $n - 1$  равны  $a^i \% p$  для  $0 \leq i < n - 1$ . В инкрементальном вычислении модуль не учитывается:  $x^i = x^{i-1} \ll 1$ , поэтому нужно вычесть  $p$  из  $x^i$ , если  $x^i \geq n$ :

```
class GF2mArithmetic
{
    int n;
    Vector<int> el2p, p2el;
```

```

public:
    int one()const{return 1;}
    int alpha()const{return 2;}
    GF2mArithmetic(int primPoly)
    { // m - это самый старший бит многочлена
        int m = lgFloor(primPoly);
        assert(m <= 16); // избегаем использования слишком большого количества памяти
        n = twoPower(m);
        el2p = p2el = Vector<int>(n - 1); // у числа 0 нет соответствующей степени
        p2el[0] = 1; // a^0 = 1, a^(n-1) // также равно 1,
            // и сохранять его не надо, а el2p[1] неявно равняется нулю
        for(int p = 1; p < n - 1; ++p)
        { // умножение на x
            int e = p2el[p - 1] << 1; // умножение на x
            if(e >= n) e = sub(e, primPoly); // сокращение при необходимости
            el2p[e - 1] = p;
            p2el[p] = e;
        }
    }
    int elementToPower(int x)const
    {
        assert(x > 0 && x < n);
        return el2p[x - 1];
    }
    int powerToElement(int x)const
    {
        assert(x >= 0 && x < n - 1);
        return p2el[x];
    } // сложение и вычитание выполняются операцией xor
    int add(int a, int b)const{return a ^ b;}
    int sub(int a, int b)const{return add(a, b);}
    int mult(int a, int b)const
    { // добавление степени и обратное преобразование
        return a == 0 || b == 0 ? 0 :
            powerToElement((elementToPower(a) + elementToPower(b)) % (n - 1));
    }
    int div(int a, int b)const
    { // вычитание степени и обратное преобразование
        assert(b != 0);
        return a == 0 ? 0 : powerToElement((elementToPower(a) + (n - 1) -
            elementToPower(b)) % (n - 1));
    }
};

```

## 19.7. Полиномы над элементами поля Галуа

Следующий шаг — определить полиномы над коэффициентами из конечного поля. Для этого не нужны никакие специальные инструменты, кроме обычной арифметики, но над полем и во всех операциях используется школьная алгебра. Как и в случае с большими числами (см. главу 17. Большие числа):

- ◆ младшие коэффициенты находятся в начале вектора хранения;
- ◆ число 0 представляется вектором размера 1, содержащим коэффициент 0;
- ◆ ведущие нули после каждой операции при необходимости обрезаются.

```
template<typename ITEM, typename ARITHMETIC>
struct Poly: public ArithmeticType<Poly<ITEM, ARITHMETIC> >
{
    Vector<ITEM> storage;
    ARITHMETIC ari;
public:
    int getSize()const{return storage.getSize();}
    int degree()const{return getSize() - 1;}
    Poly(ARITHMETIC const& theAri, Vector<ITEM> const& coeffs // значение
                                                // по умолчанию равно 0
        Vector<ITEM>(1, 0)): ari(theAri), storage(coeffs)
    {
        assert(getSize() > 0);
        trim();
    }
    static Poly zero(ARITHMETIC const& theAri){return Poly(theAri);}
    ITEM const& operator[](int i)const{return storage[i];}
    void trim()
    {while(getSize() > 1 && storage.lastItem() == 0) storage.removeLast();}
    Poly& operator+=(Poly const& rhs)
    { // сложение без переносов
        while(getSize() < rhs.getSize()) storage.append(0);
        for(int i = 0; i < min(getSize(), rhs.getSize()); ++i)
            storage[i] = ari.add(storage[i], rhs[i]);
        trim();
        return *this;
    }
    Poly& operator-=(Poly const& rhs)
    { // вычитание без переноса
        while(getSize() < rhs.getSize()) storage.append(0);
        for(int i = 0; i < min(getSize(), rhs.getSize()); ++i)
            storage[i] = ari.sub(storage[i], rhs[i]);
        trim();
        return *this;
    }
    Poly& operator*=(ITEM const& a)
    {
        for(int i = 0; i < getSize(); ++i)
            storage[i] = ari.mult(storage[i], a);
        trim();
        return *this;
    }
    Poly operator*(ITEM const& a)const
    {
        Poly temp(*this);
        temp *= a;
    }
};
```

```

        return temp;
    }
Poly& operator<=<=(int p)
{
    assert(p >= 0);
    if(p > 0)
    {
        for(int i = 0; i < p; ++i) storage.append(0);
        for(int i = getSize() - 1; i >= p; --i)
        {
            storage[i] = storage[i - p];
            storage[i - p] = 0;
        }
    }
    return *this;
}
Poly& operator>>=(int p)
{
    assert(p >= 0);
    if(p >= getSize()) storage = Vector<ITEM>(1);
    if(p > 0)
    {
        for(int i = 0; i < getSize() - p; ++i) storage[i] = storage[i + p];
        for(int i = 0; i < p; ++i) storage.removeLast();
    }
    return *this;
}
Poly& operator*=(Poly const& rhs)
{ // умножение каждого члена и суммирование
    Poly temp(*this);
    *this *= rhs[0];
    for(int i = 1; i < rhs.getSize(); ++i)
    {
        temp <<= 1;
        *this += temp * rhs[i];
    }
    return *this;
}
static Poly makeX(ARITHMETIC const& ari)
{ // x = 1 * x + 0 * 1
    Vector<ITEM> coefs(2);
    coefs[1] = ari.one();
    return Poly(ari, coefs);
}
Poly& reduce(Poly const& rhs, Poly& q)
{ // деление с остатком, выполняется аналогично числам
    assert(rhs.storage.lastItem() != 0 && q == zero(ari));
    Poly one(ari, Vector<ITEM>(1, ari.one()));
    while(getSize() >= rhs.getSize())
    { // поле гарантирует точность деления
        int diff = getSize() - rhs.getSize();

```



```

        ITEM temp2 = ari.div(storage.lastItem(), rhs.storage.lastItem());
        assert(storage.lastItem() ==
            ari.mult(temp2, rhs.storage.lastItem()));
        *this -= (rhs << diff) * temp2;
        q += (one << diff) * temp2;
    }
    return *this;
}
Poly& operator%=(Poly const& rhs)
{
    Poly dummyQ(ari);
    return reduce(rhs, dummyQ);
}
bool operator==(Poly const& rhs) const
{
    if(getSize() != rhs.getSize()) return false;
    for(int i = 0; i < getSize(); ++i) if(storage[i] != rhs[i]) return false;
    return true;
}
ITEM eval(ITEM const& x) const
{ // алгоритм Хорнера
    ITEM result = storage[0], xpower = x;
    for(int i = 1; i < getSize(); ++i)
    {
        result = ari.add(result, ari.mult(xpower, storage[i]));
        xpower = ari.mult(xpower, x);
    }
    return result;
}
};

```

*Формальная производная* многочлена вычисляется так же, как и обычная производная, но с использованием полевой арифметики, т. е.  $ax^p$  превращается в  $apx^{p-1}$ . Полиномиальные степени не являются элементами поля, поэтому нужно выполнить  $p$  сложений:

```

Poly formalDeriv() const
{
    Vector<ITEM> coefs(getSize() - 1);
    for(int i = 0; i < coefs.getSize(); ++i)
        for(int j = 0; j < i + 1; ++j)
            coefs[i] = ari.add(coefs[i], storage[i + 1]);
    return Poly(ari, coefs);
}
};

```

## 19.8. Коды Рида — Соломона

Пусть  $t = n - k$ . Код Рида — Соломона (Reed — Solomon, RS) напрямую удовлетворяет одноэлементному ограничению и позволяет исправлять  $t/2$  ошибок, поэтому предполагается нечетное значение  $k$ .  $m$  символов в некотором конечном поле интерпретируются как значения, образующие полиномиальные коэффициенты. Для простоты предполо-

жим, что поле равно  $\text{GF}(2, 8)$ , т. е. коэффициенты являются байтами. Мы также фиксируем длину блока  $n$  равной 255. Число  $k$  может быть любым нечетным значением  $< n$  в зависимости от известной информации о задаче. Обычно выбирается значение 223. Выбор примитивного полинома для поля мало что меняет, поэтому можно использовать стандартное значение 301, соответствующее  $ax^2 + 1$ , где  $a$  — это бинарный полином  $x + 1$  (Мун, 2021).

Для настройки нужно вычислить *полином генератора*  $g(x) = \prod_{0 \leq i < l} (x - a_i)$ , при этом значение  $a$  может быть любым элементом, не равным нулю или единице, — в нашем случае принято 2:

```
class ReedSolomon
{
    int n, k;
    GF2mArithmetic ari;
    typedef Poly<unsigned char, GF2mArithmetic> P;
    typedef Vector<unsigned char> V;
    P generator;
public:
    ReedSolomon(int theK = 223, int primPoly = 301): ari(primPoly),
        generator(ari, V(1, 1)), k(theK),
            n(twoPower(lgFloor(primPoly)) - 1)
    {
        assert(k < n && numeric_limits<unsigned char>::digits == 8);
        P x = P::makeX(ari);
        for(int i = 0, aPower = ari.alpha(); i < n - k; ++i)
        {
            generator *= (x - P(ari, V(1, aPower)));
            aPower = ari.mult(aPower, ari.alpha());
        }
        assert(generator.getSize() == n - k + 1);
    }
};
```

Алгоритм кодирования блока:

1. Создать полином  $m(x)$  из коэффициентов сообщения.
2.  $c(x) = (m(x) \ll t) + (m(x) \ll t) \% g(x)$ .
3. Вернуть коэффициенты  $c$ , дополненные обрезанными нулями, чтобы при необходимости достичь правильной длины блока.

```
V encodeBlock(V const& block) const
{
    assert(block.getSize() == k);
    P c(ari, block); // начальное значение c
    c <<= (n - k); // освобождение места для кода
    c += c % generator; // прибавление кода
    // будьте внимательны с обрезкой полинома, если блок равен нулю
    while(c.storage.getSize() < n) c.storage.append(0);
    return c.storage;
}
```

Произвольное сообщение разбивается на целые блоки, которые при необходимости дополняются нулями:

```
V lengthPadBlock(V block)
{
    assert(block.getSize() < k);
    block.append(block.getSize());
    while(block.getSize() < k) block.append(0);
    return block;
}

pair<V, bool> lengthUnpadBlock(V block)
{
    assert(block.getSize() == k);
    while(block.getSize() >= 0 && block.lastItem() == 0)block.removeLast();
    bool correct = block.getSize() >= 0 &&
        block.lastItem() == block.getSize() - 1;
    assert(correct);
    if(correct) block.removeLast();
    return make_pair(block, correct);
}
```

Расшифровка выполняется сложнее:

1. Вычислить полином синдрома  $s(x)$ .  $s[i] = c(a^i)$  для соответствующего  $a^i$  из  $g$ . Если  $s(x) = 0$ , ошибки не обнаруживаются (но это не значит, что их нет).
2. Найти полиномы локатора ошибок  $\Lambda(x)$  и оценщика ошибок  $\Omega(x)$  (о них расскажем позже). Нормировать оба, разделив их на  $\Lambda[0]$ . Если  $\Lambda(x) = 0$ , сообщить об ошибке.
3. Найти корни  $\Lambda(x)$ , оценивая его в каждом ненулевом элементе поля. Сообщить об ошибке, если корней не найдено. Для корня  $r$  степенное представление его обратного  $r^{-1}$  есть соответствующее место ошибки, а значение ошибки  $= -\Omega(r^{-1}) / \Lambda'(r^{-1})$ , где  $\Lambda'$  — это формальная производная от  $\Lambda$ .
4. Исправить ошибки в  $c$ , используя  $c[i] = c[i] + \text{error}[i]$ .
5. Если  $c(x) \% g(x) = 0$ , сообщить об ошибке.

На протяжении всего алгоритма используется полевая арифметика. Доказательства правильности каждого шага приведены в работе [19.4]:

```
pair<V, bool> decodeBlock(V const& code) const
{ // вычисление полинома синдрома
    assert(code.getSize() == n);
    P c(ari, code);
    int t = n - k, aPower = ari.alpha();
    V syndromes(t);
    for(int i = 0; i < t; ++i)
    {
        syndromes[i] = c.eval(aPower);
        aPower = ari.mult(aPower, ari.alpha());
    }
    P s(ari, syndromes);
    if(s == P::zero(ari)) // если истина, ошибок нет
    { // извлечение проверочных данных и восстановление удаленных ранее нулей
        c >>= t;
    }
}
```

```

    while(c.storage.getSize() < k) c.storage.append(0);
    return make_pair(c.storage, true);
} // вычисление полиномов локатора и оценщика
pair<P, P> locEv = findLocatorAndEvaluator(s, t);
if(locEv.first == P::zero(ari)) return make_pair(code, false);
// поиск корней локатора
V roots;
for(int i = 1; i < n + 1; ++i)
    if(locEv.first.eval(i) == 0) roots.append(i);
if(roots.getSize() == 0) return make_pair(code, false);
// поиск ошибочных значений
P fd = locEv.first.formalDeriv();
V errors;
for(int i = 0; i < roots.getSize(); ++i) errors.append(ari.sub(0,
    ari.div(locEv.second.eval(roots[i]), fd.eval(roots[i]))));
// исправление ошибок
while(c.storage.getSize() < n) c.storage.append(0);
for(int i = 0; i < roots.getSize(); ++i)
{
    int location = ari.elementToPower(ari.div(ari.one(), roots[i]));
    assert(location < c.getSize());
    c.storage[location] = ari.add(c.storage[location], errors[i]);
}
if(c % generator != P::zero(ari)) return make_pair(code, false);
c >= t;
return make_pair(c.storage, true);
}

```

$\Lambda(x)$  и  $\Omega(x)$  находятся модифицированной версией алгоритма Евклида:

1. Начать с  $\Lambda_0 = 1$ ,  $\Lambda_{-1} = 0$ ,  $\Omega_0 = s(x)$ ,  $\Omega_{-1} = x^t$ .
2. Пока степень  $(\Omega^i) \geq t/2$ :
3.  $\Omega_{i+1} = \Omega_{i-1} \% \Omega_i$ , с частным  $q$ .
4.  $\Lambda_{i+1} = \Lambda_i - 1 - q\Lambda_i$ .

```

pair<P, P> findLocatorAndEvaluator(P const& syndromePoly, int t) const
{
    P evPrev(ari, V(1, ari.one())), ev = syndromePoly,
    locPrev = P::zero(ari), loc = evPrev;
    evPrev <= t;
    while(ev.degree() >= t/2)
    {
        P q(ari);
        evPrev.reduce(ev, q);
        swap(ev, evPrev);
        locPrev -= q * loc;
        swap(loc, locPrev);
    } // нормализация
    if(loc != P::zero(ari))
    {
        int normalizer = ari.div(ari.one(), loc[0]);

```

```

    loc *= normalizer;
    ev *= normalizer;
}
return make_pair(loc, ev);
}

```

Коды RS являются линейными, поскольку полиномиальные операции соответствуют умножению матриц. Они также позволяют исправить *t* *опечаток* (т. е. ошибок с известным местоположением) или комбинаций ошибок и опечаток (см. [19.4]). Во множестве приложений — например, в проверке штрихкодов товаров, используется только декодирование опечаток.

Коды RS неэффективны в исправлении случайных битовых ошибок, потому что только при больших *n* можно гарантировать исправление символьных *d* ошибок. Но если вероятны *пакетные ошибки*, которые влияют на несколько последовательных битов (например, царапины на диске CD/DVD), коды RS работают хорошо.

## 19.9. Границы кодов минимального расстояния с фиксированным алфавитом

Коды RS удовлетворяют одноэлементной границе путем увеличения алфавита на *n* так, чтобы каждый символ занимал  $\lceil \lg(n) \rceil$  битов. Многие другие коды работают непосредственно с битами или другими символами фиксированного размера, независимо от *n*. Асимптотически при  $n \rightarrow \infty$  все коды минимального расстояния имеют некоторые ограничения (см. [19.4, 19.1]). Пусть  $b = d/n$ . Тогда для двоичных символов (это правило распространяется на более крупные алфавиты по правилу обобщенной энтропии) асимптотически применимо:

- ◆ Один элемент:  $R \leq 1 - b$ .
- ◆ Плоткин:  $R \leq 1 - 2b$ . Строго лучше, чем одноэлементная граница. Код соответствует одноэлементным границам, но это не является противоречием, потому что размер алфавита, стремящийся к бесконечности, приводит к тому, что лучшие границы сводятся к одному элементу.
- ◆ Гилберт — Варшамов:  $R \geq 1 - H(b)$ . Нижняя граница, но также может быть верхней границей до  $O(1)$  (см. [19.1]). Выражение это идентично тому, которое дает пропускную способность BSC.

У алгоритма BSC при  $n \rightarrow \infty$  доля ошибок равна ровно *p* (например, по неравенству Хеффдинга), т. е. требуется значение  $b > 2p$ , чтобы гарантировать исправление ошибок. Таким образом, эффективная пропускная способность для кодов с минимальным расстоянием равна  $1 - H(2b)$ . Это значение ниже *C*, и для  $p \geq 1/4$  надежная коммутация невозможна, чего нельзя сказать об общих кодах. Таким образом, все коды минимального расстояния с фиксированным алфавитом асимптотически ошибочны. Но для значений *n* до 1000 или около того некоторые двоичные коды дают хорошую производительность (например, код BCH — см. *разд. «Комментарии»*). Декодирование с максимальным правдоподобием позволяет декодировать с превышением значения *d*, и современные коды основаны на этом принципе.

## 19.10. Булевы матрицы

Булева матричная алгебра лежит в основе вероятностных кодов. Эти матрицы работают аналогично обычным матрицам (см. главу 22. *Численные алгоритмы: введение и матричная алгебра*) с булевыми коэффициентами, но реализация оказывается более эффективна с точки зрения памяти, поскольку в качестве хранилища используется набор битов. В некоторых операциях получается битовый параллелизм:

```
class BooleanMatrix: public ArithmeticType<BooleanMatrix>
{
    int rows, columns;
    int index(int row, int column) const
    {
        assert(row >= 0 && row < rows && column >= 0 && column < columns);
        return row + column * rows;
    }
    Bitset<> items;
public:
    BooleanMatrix(int theRows, int theColumns): rows(theRows),
        columns(theColumns), items(theRows * theColumns)
    {assert(items.getSize() > 0);}
    int getRows() const{return rows;}
    int getColumns() const{return columns;}
    bool operator()(int row, int column) const
    {return items[index(row, column)];}
    void set(int row, int column, bool value = true)
    {items.set(index(row, column), value);}
    BooleanMatrix operator*=(bool scalar)
    {
        if(!scalar) items.setAll(false);
        return *this;
    }
    friend BooleanMatrix operator*(bool scalar, BooleanMatrix const& a)
    {
        BooleanMatrix result(a);
        return result *= scalar;
    }
    friend BooleanMatrix operator*(BooleanMatrix const& a, bool scalar)
    {return scalar * a;}

    BooleanMatrix& operator+=(BooleanMatrix const& rhs)
    { // сложение и вычитание выполняются операцией xor
        assert(rows == rhs.rows && columns == rhs.columns);
        items ^= rhs.items;
        return *this;
    }
    BooleanMatrix& operator-=(BooleanMatrix const& rhs){return *this += rhs;}

    BooleanMatrix& operator*=(BooleanMatrix const& rhs)
    { // обычное умножение строки на столбец
        assert(columns == rhs.rows);
```

```

    BooleanMatrix result(rows, rhs.columns);
    for(int i = 0; i < rows; ++i)
        for(int j = 0; j < rhs.columns; ++j)
        {
            bool sum = false;
            for(int k = 0; k < columns; ++k)
                sum ^= (*this)(i, k) * rhs(k, j);
            result.set(i, j, result(i, j) ^ sum);
        }
    return *this = result;
}

Bitset<> operator*(Bitset<> const& v) const
{ // умножение матрицы на вектор
    assert(columns == v.getSize());
    Bitset<> result(rows);
    for(int i = 0; i < rows; ++i)
        for(int j = 0; j < columns; ++j)
            result.set(i, result[i] ^ ((*this)(i, j) * v[j]));
    return result;
} // умножение вектора на матрицу
friend Bitset<> operator*(Bitset<> const& v, BooleanMatrix const& m)
{ return m.transpose() * v; }

static BooleanMatrix identity(int n)
{
    BooleanMatrix result(n, n);
    for(int i = 0; i < n; ++i) result.set(i, i);
    return result;
}

BooleanMatrix transpose() const
{
    BooleanMatrix result(columns, rows);
    for(int i = 0; i < rows; ++i)
        for(int j = 0; j < columns; ++j) result.set(j, i, (*this)(i, j));
    return result;
}

bool operator==(BooleanMatrix const& rhs)
{
    if(rows != rhs.rows || columns != rhs.columns) return false;
    return items == rhs.items;
}

};

```

## 19.11. Коды проверки на четность с низкой плотностью

При заданном  $m$ , которое здесь интерпретируется как вектор-столбец, линейный код использует систематическую порождающую матрицу  $G$  для вычисления кодового слова  $w = Gm$ . При  $G = [X|I]^T$  (« $\gg$ » — операция дополнения)  $H = [IX]$  — это соответствующая проверочная матрица такая, что:

- ◆  $HG = 0$ ;
- ◆ для любого кодового слова  $w$   $Hw = 0$ .

Коды проверки на четность с низкой плотностью (Low-density Parity-check Codes, LDPC) являются двоичными и создаются путем конструирования  $H$  и получения  $G$  извлечением  $X$ . Разреженная проверочная матрица  $A$  создается и преобразуется в  $H$  за счет того факта, что  $G$  или  $H$ , преобразованные перестановками столбцов или элементарными операциями со строками, дают один и тот же код (с точностью до битовой перестановки, соответствующей перестановке столбцов). Результирующие значения  $H$  и  $G$  не являются разреженными, поэтому можно отбросить  $H$  и использовать для декодирования  $A$ .

Пусть  $t = n - k$ , а  $w_c$  и  $w_r$  такие, что  $\frac{n}{w_r} = \frac{t}{w_c}$ , причем деление выполняется без остатка (получается *обычный код LDPC* — см. [19.4]). Как правило, задают значение  $w_c = 3$  (меньшие значения неэффективны, а большие бессмысленны), и вычисляют  $w_r$  из приведенного ранее уравнения.

Алгоритм создания матрицы  $A$ :

1. Создать  $A'$  в строке  $r$  такой, что  $0 \leq r < \frac{t}{w_c}$ , установить  $w_r$  последовательных битов, начиная с позиции  $rw_r$ .
2. Сохранить  $w_c$  перестановок  $A'$  таких, что первая определяет идентичности, а остальные заданы случайно.

Полученная матрица  $A$  не обязательно будет полноранговой, т. е. некоторые ее строки могут оказаться линейно зависимы. Таким образом, во время исключения Гаусса с поворотом столбца при создании  $H$  ни в одном столбце в обработанной строке не должно быть единиц. То есть если строка состоит только из нулей, то она пропускается, и поэтому вместо более распространенного поворота строки используется поворот столбца. При этом  $R$  может быть немного больше, чем предполагалось, что необходимо учитывать при выборе значения  $R$ . В качестве альтернативы можно повторять генерацию до тех пор, пока не будет получен полный ранг, но для этого может потребоваться много повторов алгоритма. Гибридная стратегия заключается в повторении алгоритма до тех пор, пока не будет достигнуто минимальное количество столбцов. Подобные стратегии здесь не рассматриваются.

Еще один фокус заключается в том, что всякий раз, когда столбцы меняются местами во время поворота, тот же самый обмен должен применяться к  $A$ .

Затем из  $H$  удаляется 0 строк, что дает матрицу  $H'$  размера  $t' \times n$ . Матрица является систематичной, поэтому матрицу  $G$  можно получить из нее. Она имеет размер  $n' \times k'$ , где  $k' = n - t'$ . Несмотря на удаление строки,  $AG = 0$ , поэтому матрица  $A$  остается допустимой матрицей проверки четности для  $G$  (рис. 19.2):

```
class LDPC
{
    BooleanMatrix a, g; // разреженность матрицы A не рассматривается
    Bitset<> extractMessage(Bitset<> const &code) const
```



```

{
    int k = getNewK(), n = code.getSize();
    Bitset<> message(k);
    for(int i = 0; i < k; ++i) message.set(i, code[i + n - k]);
    return message;
}

public:
    int getNewK()const{return g.getColumns();}
    LDPC(int n, int k, int wc = 3): a(n - k, n), g(n, n - k)
    {
        int t = n - k, wr = n/(t/wc);
        assert(t % wc == 0 && n % wr == 0 && wc * n == wr * t && t < n);
        // создание a
        for(int r = 0; r < t/wc; ++r) // первое сечение
            for(int c = 0; c < wr; ++c) a.set(r, c + r * wr);
        Vector<int> perm(n);
        for(int c = 0; c < n; ++c) perm[c] = c;
        for(int i = 1; i < wc; ++i) // остальные сечения являются
                                   // перестановками первого
        { // первый
            GlobalRNG().randomPermutation(perm.getArray(), n);
            for(int r = 0; r < t/wc; ++r)
                for(int c = 0; c < wr; ++c)
                    a.set(r + i * t/wc, perm[c + r * wr]);
        } // создание H из A
        BooleanMatrix h = a;
        int skip = 0;
        for(int r = 0; r < t; ++r)
        { // поиск столбца с единицей и возврат, если таковая не найдена
            int cNow = r - skip, c = cNow;
            for(; c < n; ++c) if(h(r, c)) break;
            if(c == n) ++skip; // строка, полностью состоящая из нулей
            else if(c != cNow) // обмен столбцов cNow и c
                for(int rb = 0; rb < t; ++rb)
                {
                    bool temp = h(rb, cNow);
                    h.set(rb, cNow, h(rb, c));
                    h.set(rb, c, temp);
                    // то же самое для a
                    temp = a(rb, cNow);
                    a.set(rb, cNow, a(rb, c));
                    a.set(rb, c, temp);
                }
            for(int rb = 0; rb < t; ++rb)
                if(rb != r && h(rb, cNow))
                    for(c = cNow; c < n; ++c)
                        h.set(rb, c, h(rb, c) ^ h(r, c));
        } // удаление нулевых строк из H
        int tProper = t - skip, delta = 0;
        BooleanMatrix hNew(tProper, n);

```

```

for(int r = 0; r < tProper; ++r)
{ // у ненулевых строк должна быть правильная идентифицирующая часть
    while(!h(r + delta, r) && r < tProper) ++delta;
    for(int c = 0; c < n; ++c) hNew.set(r, c, h(r + delta, c));
} // создание g из h
int kProper = n - tProper;
g = BooleanMatrix(n, kProper);
for(int r = 0; r < n; ++r)
    for(int c = 0; c < kProper; ++c)
        if(r < tProper) g.set(r, c, hNew(r, tProper + c)); // часть x
        else g.set(r, c, r - tProper == c); // идентифицирующая часть
assert(a * g == BooleanMatrix(t, kProper));
}
Bitset<> encode(Bitset<> const& message)const
{
    assert(message.getSize() == getNewK());
    return g * message;
}
};

```

The image shows a 20x20 matrix of 0s and 1s, representing the result of a matrix calculation. The matrix is divided into three sections: a 7x7 top-left section labeled 'a', a 7x7 middle-right section labeled 'g', and a 6x6 bottom-right section labeled 'h'. The matrix is displayed on a black background with white text.

Рис. 19.2. Пример расчета матрицы  $G$  по  $A$ , где  $n = 20$  и  $k = 7$

Для декодирования алгоритмом BSC нужно знать значение  $p$ . Но точность значения алгоритму не важна, поэтому можно эвристически предположить, что  $C = R$ , и использовать численное решение уравнения, чтобы найти соответствующее  $p$ .

При декодировании используется максимальное правдоподобие (см. главу 21. *Вычислительная статистика*). Учитывая, что бит может принимать лишь два значения, нужно вычислить логарифм отношения вероятностей для любого бита. Априори:

$$l_{\text{initial}}(b) = \ln \left( \frac{\Pr(\text{bit} = 1 \mid \text{received } b)}{\Pr(\text{bit} = 0 \mid \text{received } b)} \right) = \ln \left( \frac{1-p}{p} \right) (b ? 1 : -1).$$

Сохраняйте «матрицу»  $nu$  (реализованную хеш-таблицей), в которой находятся «частичные» вероятности в позициях, где в матрице  $A$  биты установлены. Для заданного  $r$ , пусть  $c'$  будет  $c$  таким, что бит  $A[r, c]$  установлен. Также пусть  $l_c(c') = \tanh \left( \frac{nu[r, c'] - l[r]}{2} \right)$  — это «условные вероятности» (см. [19.4]). Тогда декодирование выполняется следующим образом:

1. Установить все вхождения  $nu$  равными 0.
2. Установить все вхождения  $l$  равными  $l_{\text{initial}}(\text{code}[c])$ .
3.  $\text{corrected} = \text{code}$ .
4. Пока  $A \times \text{corrected} \neq 0$  или не исчерпаны итерации:
  5. Для любой строки  $r$  из  $A$ :
  6.  $\text{temp} = \prod l_c(c')$ .
  7.  $\forall c' \ nu[r, c'] = -2 \tanh^{-1}(\text{temp}/l_c(c'))$ .
  8. Для любого столбца  $c$  из  $A$ :
  9.  $l[c] = l_{\text{initial}}(\text{code}[c]) + \sum nu[r, c]$ .
10.  $\text{corrected}[c] = (l[c] > 0)$ .

```
struct H0Functor // для численного решения p
{
    double hValue;
    double H(double p) const { return p > 0 ? p * log2(1/p) : 0; }
    double operator() (double p) const { return H(p) + H(1 - p) - hValue; }
    H0Functor(double theHValue): hValue(theHValue) {}
};

double pFromCapacity(double capacity) const // решатель гарантированно
// найдет решение
{ return solveFor0(H0Functor(1 - capacity), 0, 0.5).first; }

unsigned int uIndex(unsigned int r, unsigned int c) const
{ return r * a.getColumns() + c; }

pair<Bitset<>, bool> decode(Bitset<> const &code, int maxIter = 1000,
double p = -1) const
{
    int n = a.getColumns(), k = getNewK(), t = a.getRows();
    assert(code.getSize() == n && maxIter > 0);
    Bitset<> zero(k), corrected = code;
```

```

if(a * code == zero) return make_pair(extractMessage(code), true);
if(p == -1) p = pFromCapacity(1.0 * k/n); // найти p, если оно не дано
double const llr1 = log((1 - p)/p); // инициализация l
Vector<double> l(n);
for(int i = 0; i < n; ++i) l[i] = llr1 * (code[i] ? 1 : -1);
LinearProbingHashTable<unsigned int, double> nu; // инициализация nu
for(int r = 0; r < t; ++r) for(int c = 0; c < n; ++c) if(a(r, c))
    nu.insert(uIndex(r, c), 0);
while(a * corrected != zero && maxIter-- > 0) // основной цикл
{ // обновление nu
    for(int r = 0; r < t; ++r)
    {
        double temp = 1;
        for(int c = 0; c < n; ++c) if(a(r, c))
            temp *= tanh((*nu.find(uIndex(r, c)) - l[c])/2);
        for(int c = 0; c < n; ++c) if(a(r, c))
        {
            double *nuv = nu.find(uIndex(r, c)), product = temp/
                tanh((*nuv - l[c])/2), value = -2 * atanh(product);
            // бесконечные значения принимаются равными 100
            if(!isfinite(value)) value = 100 * (product > 0 ? -1 : 1);
            *nuv = value;
        }
    }
    // обновление l, корректировка
    for(int c = 0; c < n; ++c)
    {
        l[c] = llr1 * (code[c] ? 1 : -1);
        for(int r = 0; r < t; ++r) if(a(r, c))
            l[c] += *nu.find(uIndex(r, c));
        corrected.set(c, l[c] > 0);
    }
}
bool succeeded = maxIter > 0;
return make_pair(succeeded ? extractMessage(corrected) : code, succeeded);
}

```

Итак, итерация декодирования выполняется за время  $O(nt')$ . Алгебра разреженных матриц позволяет сократить его до  $O(n)$ . В нашем случае нужна разреженная матричная структура данных, которая допускает итерации как строк, так и столбцов.

Что касается реализации кода RS, то вместо отправки блоков сообщений определенного размера в битах можно использовать 2 байта (в более общем случае  $\lceil \lg(k) \rceil$  битов), в которых закодирована длина блока в битах, а неиспользуемые биты обнулены.

## 19.12. Примечания по реализации

Реализации выполнены по учебникам. Креатив проявлялся лишь в выборе алгоритма. К сожалению, достаточное количество тестов я добавил намного позже, что привело к ошибкам, некоторые из которых до сих пор не исправлены.

## 19.13. Комментарии

Существует множество других полезных моделей каналов. Некоторые из наиболее важных (см. [19.4]):

- ♦ *Бинарный канал стирания* (Binary Erasure Channel, BEC) — в отличие от BSC, в этой модели биты не переворачиваются, а с некоторой вероятностью становятся нераспознаваемыми;
- ♦ *Гауссовский канал* — при передаче данных используются сильноточные и слаботочные сигналы, которые соответствуют значениям «1» и «0». Ошибки лучше всего моделируются нормальным распределением с некоторой дисперсией. Декодирование гауссовского вывода непосредственно перед преобразованием обратно в двоичный формат несколько более эффективно за счет отсутствия потерь информации при дискретизации, но результирующие коды можно будет декодировать только на оборуодовании, которому эта информация известна.

Нелинейные коды изучены гораздо меньше, а их использование не дает никаких преимуществ.

Еще один часто упоминаемый способ декодирования RS — *алгоритм Берлекэмпа — Мессе*. Однако он намного сложнее, а производительность имеет примерно такую же, поэтому использовать его нет смысла. Код RS широко используется благодаря эффективному декодированию стирания. Предлагались небинарные LDPC, способные конкурировать с кодом RS, но и они не дают столь большого преимущества, как двоичные коды.

Существуют и другие важные коды (см. [19.4]):

- ♦ *BCH* — обобщенный вариант RS, в котором размер блока отделен от размера символа, и для них задаются разные поля ввода и вывода. Обычно рассматриваются только бинарные символы. *Алгоритм Гилберта — Варшамова* вообще лучше обходить стороной (подробный анализ производительности приведен в работе [19.1]). Но при малых  $n$  для соответствующих скоростей эти коды позволяют достичь отличной скорости, и к тому же просты и эффективны. Хотя алгоритм LDPC дает лучшую скорость для больших  $n$ , для некоторых приложений достаточно BCH;
- ♦ *Turbo* — срок действия их патента истек (см. [19.8]). Этот код намного сложнее, чем LDPC, медленнее выполняет декодирование и непрост в разработке. Но зато он лучше работает с каналами с малой пропускной способностью. Текущие исследования LDPC направлены на поиск попыток устранить этот недостаток.

Подробнее об *итеративных кодах* (например, LDPC и Turbo) можно почитать в работе [19.2]. В частности, обычный LDPC может быть не самым лучшим из возможных, и рекомендуется нерегулярная конструкция, которая также предотвращает циклы размера 4 (особая конфигурация битов четности). Текущие исследования LDPC сосредоточены на детерминированных конструкциях для повышения эффективности. Также существует способ использовать матрицу  $A$  для прямого кодирования без вычисления плотной матрицы  $G$ , чтобы все операции выполнялись с разреженными матрицами (см. [19.4, 19.2]), но он сложен и в моей книге не реализован. Большинство других кодов устарело, хотя они включены во многие старые стандарты и протоколы.

Большая часть кодов, которые рассматриваются в учебниках, в момент их изобретения были самыми эффективными, по крайней мере, в своих специальных задачах.

## 19.14. Советы по дополнительной подготовке

- ◆ Разрешите построение CRC из потока.
- ◆ Исследуйте 64-битные полиномы для CRC и реализуйте 64-битную CRC на их основе.
- ◆ Внесите изменение в умножение булевых матриц, перейдя на перекрестное произведение (подробности описаны в *главе 22. Численные алгоритмы: введение и матричная алгебра*). Это позволяет выполнять побитно-параллельное сложение матриц с общей производительностью  $O(n^3/w)$ .
- ◆ Используйте буфер для имитации декодирования блоков из потока с низкой частотой ошибок. Можно также включить в реализацию алгоритмы сжатия и криптографии из соответствующих глав.
- ◆ Для алгоритма LDPC выполните генерацию  $H$  несколько раз и выберите лучший результат. Исследуйте, сколько итераций позволяет получить полезный результат.
- ◆ Реализуйте разреженную булеву матрицу. Рассмотрите возможность повторного использования общего кода разреженных матриц (см. *главу 22. Численные алгоритмы: введение и матричная алгебра*), но не забывайте, что для этого нужна бинарная алгебра.

## 19.15. Список рекомендуемой литературы

- 19.1. Guruswami V. (2010). Introduction to Coding Theory. Course notes, <http://www.cs.cmu.edu/~venkatg/teaching/codingtheory/>. Accessed October 16, 2016.
- 19.2. Johnson S. J. (2009). Iterative Error Correction: Turbo, Low-Density Parity-Check and Repeat-Accumulate Codes. Cambridge.
- 19.3. Koopman P. (2002). 32-bit cyclic redundancy codes for internet applications. International Conference on Dependable Systems and Networks (pp. 459–468). IEEE.
- 19.4. Moon T. K. (2021). Error Correction Coding: Mathematical Methods and Algorithms. Wiley.
- 19.5. Rorabaugh C. B. (1996). Error Coding Cookbook: Practical C/C++ Routines and Recipes for Error Detection and Correction. McGraw-Hill.
- 19.6. Williams R. (1993). A painless guide to CRC error detection algorithms. [http://www.repairfaq.org/filipg/LINK/F\\_crc\\_v3.html](http://www.repairfaq.org/filipg/LINK/F_crc_v3.html). Accessed May 18, 2013.
- 19.7. Wikipedia (2016a). Reed–Solomon error correction. [https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon\\_error\\_correction](https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction). Accessed October 1, 2016.
- 19.8. Wikipedia (2016b). Turbo code. [https://en.wikipedia.org/wiki/Turbo\\_code](https://en.wikipedia.org/wiki/Turbo_code). Accessed October 16, 2016.
- 19.9. Wikipedia (2018). Cyclic redundancy check. [https://en.wikipedia.org/wiki/Cyclic\\_redundancy\\_check](https://en.wikipedia.org/wiki/Cyclic_redundancy_check). Accessed September 23, 2018.

## 20. Криптография

### 20.1. Введение

В этой главе приведено базовое введение в некоторые основные идеи криптографии, которая находится в конце типичного конвейера сжатия, шифрования и кодирования ошибок перед передачей файла. Криптография — весьма широкая область, требующая очень сложных реализаций, поэтому мы не будем вдаваться в подробности. В приведенной здесь литературе не описываются последние разработки и стандарты. Если вам нужны связанные с криптографией функции, следует использовать подходящую хорошую библиотеку.

Допустим, нам нужно обезопасить различные *протоколы связи* — такие как обмен секретными сообщениями между двумя сторонами. Такие протоколы обычно моделируются взаимодействием нескольких сторон. Примеры этих протоколов:

- ♦ Алиса, Боб, Кэрл, Дэвид — общающиеся стороны;
- ♦ Ева — может видеть все сообщения;
- ♦ Мэллори — может видеть и изменять все сообщения. Хотя Мэллори может также удалять или уничтожать все сообщения, важно, чтобы их взлом был Мэллори не выгоден;
- ♦ Трент — доверенный орган, способный подтвердить идентичность стороны.

Предполагается, что каждая сторона знает протокол, а безопасность регулируется наличием *ключа* — пароля, который известен только авторизованным сторонам и без которого невозможно получить какую-либо полезную информацию о связи. Например, любой, желающий прочитать вашу электронную почту, может знать вашего поставщика услуг и имя пользователя, но не знать пароль.

### 20.2. Шифрование файлов

Исторически для *шифрования* использовалось множество небезопасных методов, позволяющих выполнить *дешифровку* без знания ключа. Юлий Цезарь сдвигал алфавит на 3 буквы, превращая букву *a* в *d* и т. д. В «Пляшущих человечках» Конан Дойля буквы соответствовали танцующим фигуркам, а Шерлок Холмс — чтобы расшифровать код — использовал эвристику, заменив наиболее часто встречающиеся фигурки на наиболее вероятные буквы, предполагая, что наиболее распространенной и часто встречающейся в английских текстах буквой является *e*, второй по распространенности — *t* и т. д.

Пусть  $r$  — это случайная битовая строка размера  $n$ . Для  $n$ -битного сообщения  $m$  код ( $m$ ) равен  $m \oplus r$  и сообщение (код  $c$ ) равно  $c \oplus r$ . Невозможно декодировать  $c$ , не имея  $r$ , по-

тому что каждый  $n$ -й бит  $t$  равновероятен. *Одноразовый код* использовать непрактично, потому что если использовать  $r$  для другого сообщения,  $c_1 \wedge c_2 = m_1 \wedge m_2$ , что приводит к утечке информации.

*Поточный шифр* задает безопасный генератор случайных чисел с достаточно случайным начальным числом и создает поток случайных байтов для формирования  $r$ . Угадать будущие байты из прошлых невозможно, если для шифрования многократно не используется одно и то же начальное число. Использование контрольной суммы перед шифрованием позволяет с высокой вероятностью обнаружить битовые ошибки в зашифрованном файле. Алгоритм шифрования с использованием потокового шифра RC4 и циклического избыточного кода (CRC):

1. Создать RC4 из ключа и преобразованной Xorshift-последовательности из текущего времени.
2. Вычислить и добавить CRC в файл.
3. Зашифровать результат в потоке.
4. Добавить к нему время.

Начальное число можно использовать в дополнение к паролю, чтобы повторяющиеся сообщения таковыми не считались. Безопасно в качестве начального значения использовать время или счетчик или их сочетания:

```
void applyARC4(uint32_t seed, Vector<unsigned char> temp,
              Vector<unsigned char>& data)
{
    for(int i = 0; i < temp.getSize(); ++i)
        temp[i] ^= (seed = xorshiftTransform(seed));
    ARC4 arc4(temp.getArray(), temp.getSize());
    for(int i = 0; i < data.getSize(); ++i) data[i] ^= arc4.nextByte();
}

Vector<unsigned char> simpleEncrypt(Vector<unsigned char> data,
                                   Vector<unsigned char> const& key)
{
    uint32_t seed = time(0), s = sizeof(int);
    CRC32 crc32;
    Vector<unsigned char> theSeed = ReinterpretEncode(seed, s), crc =
        ReinterpretEncode(crc32.hash(data.getArray(), data.getSize()), s);
    for(int i = 0; i < s; ++i) data.append(crc[i]);
    applyARC4(seed, key, data);
    for(int i = 0; i < s; ++i) data.append(theSeed[i]);
    return data;
}
```

Расшифровка:

1. Считать начальное значение.
2. Удалить его байты.
3. Прочитать и расшифровать остальные значения, используя начальное.
4. Прочитать CRC.
5. Удалить его байты.
6. Убедиться, что файл соответствует CRC.



```

pair<Vector<unsigned char>, bool> simpleDecrypt(Vector<unsigned char> code,
        Vector<unsigned char> const& key)
{
    assert(code.getSize() >= 8);
    enum{s = sizeof(uint32_t)};
    Vector<unsigned char> seed, crc;
    for(int i = 0; i < s; ++i) seed.append(code[code.getSize() + i - 4]);
    for(int i = 0; i < s; ++i) code.removeLast();
    applyARC4(ReinterpretDecode(seed), key, code);
    for(int i = 0; i < s; ++i) crc.append(code[code.getSize() + i - 4]);
    for(int i = 0; i < s; ++i) code.removeLast();
    CRC32 crc32;
    return make_pair(code, crc32.hash(code.getArray(), code.getSize()) ==
        ReinterpretDecode(crc));
}

```

И то и другое выполняется за время  $O(n)$ .

Этот алгоритм прост и удобен и, вероятно, достаточно хорош для простых приложений. Существует более сложный алгоритм *AES* (см. [20.3]), который является стандартом шифрования с паролем.

## 20.3. Длина ключа

Ключ должен иметь пренебрежимо малую вероятность того, что его угадают или найдут перебором. Например, система с массовым параллелизмом может проверить все 64-битные ключи, если каждый из них проверяется достаточно быстро. Ключ должен быть таким, чтобы для наиболее эффективной атаки требовался эквивалент перебора 128-битных ключей методом полного перебора, что невозможно даже в будущем. Выводы:

- ◆ использование более коротких ключей эффективнее, но обработка данных из-за этого замедляется;
- ◆ с каждым годом аппаратное обеспечение становится быстрее, а хакеры учатся проводить более эффективные атаки;
- ◆ некоторые данные должны храниться в защищенном виде в течение многих лет, поэтому длина ключа, выбранная сейчас, должна оставаться безопасной в будущем;
- ◆ нужно учитывать значение данных и кому они важны. Например, защита вашей кредитной карты от мелких преступников требует меньшего уровня безопасности, чем защита сверхсекретной правительственной информации от других правительств, имеющих дешифровщиков и суперкомпьютеры, работающие годами.

Довольно легко перебрать типичный 8-символьный пароль, даже если все буквы в нем равновероятны. Впрочем, на практике попытаться взломать ключ методом грубой силы сложно даже для маленьких ключей. Нужно уметь распознавать правильно расшифрованные данные из мусорных данных. Например, данные могут быть закодированы на неизвестном языке. Для файловых паролей простая эвристика состоит в том, чтобы проверить, все ли символы находятся в таблице ASCII, — быстрый и точный тест во многих случаях (табл. 20.1).

Таблица 20.1. Варианты символьного шифрования

Тип символа	Количество возможных паролей	Биты безопасно- сти	Время перебора 10 <sup>9</sup> паролей в секунду	Длина, требующаяся для 128-битной безопасности
Цифры	10 <sup>8</sup>	27	0,13 секунды	39
Строчные буквы	26 <sup>8</sup>	38	275 секунд	27
Смешанный регистр + цифры	62 <sup>8</sup>	48	3,26 дня	21
Непробельные символы на клавиатуре	94 <sup>8</sup>	52	52,1 дня	20
Байты	256 <sup>8</sup>	64	585 лет	16

Пользователи склонны выбирать пароли, состоящие из конкатенаций словарных слов, безопасность каждого из которых составляет < 16 битов. Правила вроде «пароль должен содержать хотя бы один небуквенный символ» дают мало пользы, потому что пользователи будут вводить только один такой символ, а перебирать кортежи из 6 букв и 1 цифры гораздо проще, чем кортежи из 7 букв или цифр. Ограничения вроде блокировки после 10 неудачных попыток позволяют эффективно предотвратить попытки взлома (но в то же время доставляют неудобства пользователю, если целью взлома является блокировка).

Надежным и легко запоминающимся паролем является запоминающаяся необычная фраза, например: *ЕслиВамУгрожаютВНовыйГодЗвоните911*. Безопасный генератор случайных чисел с высокоэнтروпийным начальным числом генерирует наиболее безопасные ключи.

20.4. Хранилище ключей

Ключ необходимо запомнить или где-то сохранить. Если он будет утрачен, зашифрованные данные будет невозможно восстановить. Вы можете найти ключ, имея доступ к устройству дешифрования, потому что запуск его в отладчике может помочь получить ключ. К примеру, ключи извлекались из аппаратных устройств путем измерения тепловыделения или замораживания микросхем оперативной памяти. Приложение такие атаки предотвратить не может.

Самый простой способ получить ключ — подкупить кого-то или использовать методы *социальной инженерии*. Например (см. [20.1]), вы получаете штраф за нарушение правил дорожного движения, а полицейские в вашем городе находятся на учебе, которая для них важнее явки в суд. Вы звоните в участок, представляетесь адвокатом, которому нужно вызвать в суд офицера, выписавшего штраф, и узнаете, в какие даты сотрудник будет недоступен. Услужливый клерк дает вам эти даты. Затем в суде вы требуете проведения слушания в одну из этих дат. Справедливый судья соглашается. Вы приходите, а полицейский нет — штраф отменяется.

## 20.5. Криптографическое хеширование

Хеш-функция  $h$  криптографически безопасна, если:

- ◆  $h(x)$  вычисляется легко;
- ◆ любые биты  $h(x)$  вычислить сложно;
- ◆ сложно вычислить  $y$  такое, что  $h(y) = h(x)$ ;
- ◆ сложно вычислить  $y$  и  $z$  такие, что  $h(y) = h(z)$ .

Функция  $h$  в итоге выводит результаты размера более 128 битов, что гораздо эффективнее, чем универсальная хеш-функция. Рекомендуется использовать алгоритм *SHA3* (подробности описаны на веб-сайте NIST).

## 20.6. Обмен ключами

Две стороны, не знающие друг друга, не имеют общих ключей. Распространенная модель выглядит так: Алиса и Боб обмениваются сообщениями, а Ева их читает. В другой модели Мэллори перехватывает сообщения и притворяется Бобом, когда разговаривает с Алисой, и Алисой, когда разговаривает с Бобом, обмениваясь с ключами с каждым.

Алгоритм *RSA* (подробности см. в работе [20.3]) дает каждой стороне 456, известный только этой стороне, и *открытый ключ*. Сообщение шифруется с помощью открытого ключа и расшифровывается с помощью закрытого ключа. Эта система неуязвима для атаки «человек посередине», даже если открытый ключ опубликован. Например, в веб-браузерах есть списки доверенных органов — таких как Verisign. Если браузер не может распознать сертификат сайта, это может означать мошенничество или нежелание сайта платить.

При проектировании выделенного клиента и сервера, нужно дать серверу один закрытый ключ и жестко закодировать открытый ключ в клиенте.

## 20.7. Другие протоколы

Предположим, что каждая сторона не хочет быть обманутой и обманывать других (табл. 20.2).

**Таблица 20.2.** Протоколы алгоритмов

Протокол	Цель	Идея алгоритма
Аутентификация	Проверить личность пользователя и защитить сохраненные пароли	<ol style="list-style-type: none"> <li>1. В любой учетной записи хранится имя пользователя, длинная случайная строка и хеш пароля, объединенный со случайной строкой.</li> <li>2. Для аутентификации нужно извлечь случайную строку пользователя, рассчитать хеш пароля и проверить соответствие сохраненному результату.</li> </ol>

Таблица 20.2 (окончание)

Протокол	Цель	Идея алгоритма
Обязательство	Алиса хочет осуществить решение, не раскрывая его Бобу, и Боб не хочет, чтобы Алиса меняла его после совершения	1. Боб посылает Алисе длинную случайную строку. 2. Алиса добавляет к ней свое решение, шифрует результат с помощью случайного ключа и отправляет его Бобу. 3. Когда пришло время сообщить решение, Алиса посылает Бобу свой ключ
Обмен секретами	Секрет сообщается для $t$ сторон, так что любая сторона $k$ может его определить.	1. Выберите простое число $p > \max(k, \text{максимально возможный секрет})$ . 2. Создайте случайный полином степени $k - 1$ . Коэффициент при свободном члене — это сообщение. 3. Выдайте стороне $i$ полином $\% p$ , оцененный по $i$ . 4. Чтобы построить секрет, $k$ сторон решают соответствующие $k$ уравнений.

20.8. Примечания по реализации

Для хорошей портативной реализации криптографии на C++ нужны функциональные возможности, выходящие за рамки STL. В частности, нужен безопасный источник случайных чисел. Так что, несмотря на мое желание поместить в эту главу намного больше информации, многие реализации я сократил, когда пришел к этой мысли во время экспериментов с AES. Вы можете инкапсулировать криптографию, чтобы в качестве входных данных требовалось безопасное случайное начальное число — по крайней мере, в форме функтора «черного ящика», что позволило бы сделать хорошие реализации многих алгоритмов. Я решил не идти по этому пути и вместо этого разработал простой потоковый шифр, основанный на стратегии заполнения на основе пароля.

20.9. Комментарии

Криптография — это обширная сфера. Из-за наличия особых случаев и эффективных средств решения сложных задач необходимо проявлять особую осторожность, чтобы в них не попали случайно сгенерированное начальное число или другая информация. У хорошо зарекомендовавших себя протоколов, таких как RSA, простые случаи хорошо известны, хотя во многих источниках они игнорируются. Таким образом, сложность реализации всегда необходима.

Невозможно создать протокол, в котором не существует особых случаев. Теоретически, чтобы предотвратить легкое решение сложных задач, необходимо доказать отсутствие эффективных рандомизированных или фиксированных параметров алгоритма и, возможно, показать, что с большой вероятностью происходит фазовый переход и т. д. (см. главу 16. Комбинаторная оптимизация). Безопасность в основном строится на предположении о том, что такие случаи маловероятны. Многие протоколы используют дополнительное скремблирование на случай, если проблема обнаружится. Например, в задачах хеширования старые алгоритмы MD5 и SHA1 считаются неработоспособны-

ми, несмотря на то, что много лет назад они казались полностью безопасными. Алгоритм SHA3 был разработан именно потому, что появилось ощущение, будто безопасный в то время SHA2 скоро будет взломан.

Качество генератора случайных чисел и само начальное число имеют решающее значение. Использование в качестве начального значения сочетания пароля и времени является хорошей эвристикой, но в идеале лучше иметь случайный источник на специальном устройстве. Иногда это единственный вариант, потому что многие алгоритмы, такие как безопасные хеш-функции, работают без пароля.

Возможность обновления протокола очень важна. Одной из причин поражения Германии во Второй мировой войне мог быть взлом машины Enigma. Она работала на усовершенствованном варианте шифра пляшущих человечков, который не поддавался грубой силе человеческих атак, но не выдержал напора одного из первых компьютеров. Даже специалисты в этой области не знают наверняка, безопасны ли существующие протоколы. В лучшем случае можно создавать библиотеки надежных реализаций и активно отражать известные атаки. Хороший протокол должен в течение многих лет противостоять атакам, чтобы считаться пригодным для использования. Многие злоумышленники пытаются пробить защиту предложенных протоколов, потому что наградой за успех будет слава и хорошие возможности для работы.

В конвейере передачи/хранения данных обычно задействуются все функции сжатия, обнаружения ошибок, шифрования и исправления ошибок. Используется именно этот порядок, потому что зашифрованные данные плохо сжимаются. Нам нужно обнаружить фальсификацию зашифрованных данных и сделать так, чтобы после исправления ошибок мы восстановили все зашифрованные данные.

## 20.10. Совет по дополнительной подготовке

Усовершенствуйте портативность кода дешифрования, не предполагая конкретного размера слов.

## 20.11. Список рекомендуемой литературы

- 20.1. Mitnick K. D., & Simon W. L. (2001). *The Art of Deception: Controlling the Human Element of Security*. Wiley.
- 20.2. Schneier B. (1995). *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley.
- 20.3. St Denis T. (2006). *Cryptography for Developers*. Syngress.

## 21. Вычислительная статистика

Все модели ошибочны, но некоторые из них полезны.

*Джордж Бокс*

Имея статистику, наврать легко, но без нее проще.

*Фредерик Мостеллер*

Никогда не давайте оценку, не имея доверительного набора.

*Ларри Вассерман*

Консультация со статистиком после завершения эксперимента часто равносильна проведению вскрытия.

*Сэр Рональд Фишер*

Любой, кто ведет спор, апеллируя к авторитету, пользуется не интеллектом, а памятью.

*Леонардо да Винчи*

Нет более распространенной ошибки, чем предполагать, что длительные и точные математические расчеты позволяют применить эти результаты к природным явлениям.

*Альфред Норт Уайтхед*

Совершенно невероятно, чтобы математическая формула могла предвидеть будущее, и те, кто думает, что это возможно, когда-то поверили бы в колдовство.

*Якоб Бернулли*

Нас беспокоит не то, чего мы не знаем, а то, что мы знаем, что это не так.

*Уилл Роджерс*

### 21.1. Введение

Эта глава представляет собой полезный обзор математической статистики, но без математики. Основное внимание уделяется процедурам анализа данных и теории, необходимой для их понимания. Чтобы разобраться в материале этой главы, нужно предварительно освоить курс математической статистики. Я уделяю особое внимание оценке и доверительным интервалам и преуменьшаю значение проверки гипотез. Процедуры с минимальными предположениями, если таковые есть, использовать предпочтительнее, и мы обсудим практическое применение общей процедуры начальной загрузки, где я добавлю кое-что от себя.

Основная идея статистики заключается в том, что если точки данных следуют некоторому распределению, то их функция следует другому распределению, определение которого позволяет делать всевозможные выводы.

### 21.2. Оценки

Теория вероятностей описывает поведение известных случайных явлений. Статистика — это обратный процесс, т. е. определение свойств *генератора данных* на основе данных, которые он генерирует. Для этого нужны (см. [21.82]):

- ◆ *модель* — для ее выбора требуется научное обоснование, и она не обязана даже близко соответствовать реальному генератору. В работе мы исходим из того, что соответствие есть;
- ◆  $N$  независимых случайных точек данных, которые, предположительно, созданы в соответствии с моделью и часто бывают уже доступны, но иногда для их сбора необходимы дополнительные эксперименты. В некоторых случаях также разрешены данные, не являющиеся независимыми;
- ◆ хорошо бы знать параметры модели, чтобы получить ее полную спецификацию, но часто требуется узнать лишь некоторые конкретные *параметры*. Параметры не обязательно должны быть частью какой-либо математической спецификации модели, хотя обычно являются таковыми.

Например (конкретные методы обсуждаются позже в этой главе и в *главе 27. Машинное обучение: регрессия*):

- требуется определить текущий средний (медианный) доход страны, имея данные опроса (настоящий средний — без учета миллиардеров, искажающих статистику);
- требуется оценить стоимость дома и, предполагая, что цену описывает модель линейной регрессии, использовать данные о прошлых продажах для учета частичного вклада таких факторов, как налог на имущество и рейтинги близлежащих школ.

Для этой задачи можно использовать модель случайного леса, в которой для каждого дерева регрессии оцениваются как структура его узлов, так и параметры каждого узла.

Вопрос, верна ли модель, возникает редко, но обычно некорректная модель приводит лишь к небольшой ошибке аппроксимации (о ней поговорим в *главе 25. Основы машинного обучения*), так что ее истинность не столь важна. Любая модель, какой бы бессмысленной она ни была, может быть оценена по любым данным, если для этой модели такие данные возможны. В худшем случае вы дадите хорошую оценку плохой модели. Модель абстрактна и не связана с реальностью — даже для примера со средним доходом допущение о независимости и случайности не обязательно должно быть верным, потому что респонденты опроса могут хвастаться или недоговаривать чего-то. Впрочем, обычно получают полезные модели, которые оцениваются по качеству предсказания (см. *главу 25. Основы машинного обучения*). Цель состоит в том, чтобы определить, по крайней мере, качественное поведение — что-то верное в модели, вероятно, будет правдой и в реальности.

Вполне допустимо использовать приближенные модели. Они часто соответствуют жизненным ситуациям — например, законы являются приближением к какой-то идеальной системе правосудия, правила дорожного движения (особенно ограничения скорости) тоже довольно хорошо моделируют то, как должны ездить водители. В приближенных моделях можно делать дополнительные приближения, упрощая вычисления или принимая незначительные допущения. Вернемся к приведенным ранее примерам:

- ◆ для вычисления среднего дохода можно предположить распределение с некоторой медианой, что всегда верно для непрерывного распределения (и большинства дискретных с разрывом в медиане);

- ♦ модель цен на жилье со случайным лесом предполагает, что механизм точно соответствует случайному лесу деревьев регрессии с некоторой структурой и значениями узлов;
- ♦ линейная регрессия обычно более приближительна, чем случайный лес, но все же полезна из-за своей простоты. Хотя вывод расчетов часто делается без каких-либо вероятностных допущений, это особый случай применения максимального правдоподобия (обсуждается далее в главе) к нормально распределенным ошибкам.

В некоторых случаях модель можно вывести из свойств предметной области. К примеру, в физике новый важный параметр, скорее всего, окажется революционным, поэтому вероятно, не будет иметь статистического значения. Кроме того, например, если вы имеете дело с данными без памяти, можно рассмотреть модель экспоненциального распределения.

Параметры могут быть *структурными*, как в модели случайного леса, — чаще всего они представляют собой любую информацию, необходимую для определения конкретного генератора данных. Для деревьев регрессии это могут быть точки разделения, определенные условно для того или иного узла на узлах его предков. Выделим некоторые важные классы параметров:

- ♦ *неопределимые* — не поддающиеся оценке. Например, когда квадратные нормальные выборки теряют по крайней мере знак среднего. Обычно можно оценить только агрегатные параметры, описывающие типичное поведение, — например, среднее значение, квантили или значение разделения узла дерева. Несмотря на существование множества алгоритмов, статистический анализ занимается не тем, что вы хотите, а тем, что вы можете извлечь из имеющейся у вас информации;
- ♦ *мешающие* — не относящиеся к тем, которые мы пытаемся определить, но влияющие на оценку. Например, при оценке среднего значения и его доверительного интервала обычно также необходимо оценивать дисперсию.

Когда требуемые параметры известны, статистика не нужна. Но если их нет, она позволяет, насколько это возможно из имеющихся данных:

- ♦ оценить значения параметров — цели оценки также называются *оценщиками*;
- ♦ с разумной вероятностью исключить маловероятные значения — т. е. сделать ошибки из выводов более редкими. Это делается с помощью доверительных интервалов и проверок гипотез (о них позже в этой главе).

Иногда требуется получить функции оценки — например, разницу. Удачность выбора оценок может зависеть от оценщиков — так, если разность средних и разность медиан одинаково полезны, выбрать нужно ту, у которой оценка лучше всего подходит для рассматриваемой ситуации. Чтобы избежать глупых применений, лучше сначала подумать о том, как бы вы использовали параметр  $\theta$ , если бы знали его точно, а затем задействовать любую оценку таким же образом, но с корректировками.

В большинстве задач нельзя сделать какую-либо оценку без предположений, и некоторые из них бывают более разумны, чем другие. Во многих алгоритмах делаются некоторые эвристические выборы, вследствие чего такие алгоритмы и сами могут рассматриваться как эвристические. В частности, вместо предположения о том, что все проблемы разрешимы (а это не так), лучше предположить, что ни одна из них не разрешима, и обнаружить затем, что некоторые все же разрешимы при определенных



предположениях. То же самое относится и к алгоритмам в численных методах и оптимизации и машинном обучении, где для полезных решений требуется какая-то особая структура.

Наиболее интересным вариантом статистического вывода является случай, когда для данных предполагается конкретное распределение при полной или частично известной спецификации параметров. Типичное предположение состоит в том, что при заданном  $\theta$  есть нормальное распределение ( $\theta$  — известная или неизвестная дисперсия). Это приводит к *логарифмической вероятности*:

$$LL(\{x_i\} | \{\theta\}) = \sum_{0 \leq i < n} \log(PDF(x_i | \{\theta\})).$$

Это совместное ненормализованное распределение вероятности, преобразованное с помощью логарифма для математического и числового удобства.

Когда не предполагается никакого конкретного распределения, можно выполнить повторную выборку, равномерно извлекая из него новые данные, что приводит к *эмпирической функции распределения* (Empirical Distribution Function, EDF):

$$F(x) = \sum_i \frac{X > S_i}{n}.$$

Отсюда можно сделать много выводов, причем без существенных дополнительных предположений.

Нет смысла изучать без дополнительных ограничений (например, при размере выборки  $n$ ) статистику типа длины доверительного интервала, потому что для разумных построений она стремится к нулю, т. е. параметр не существует.

## 21.3. Оценщики

Оценщик  $f$  является функцией данных, значение которых должно быть близко к  $\theta$ . Сам по себе он является случайным, зависит от данных и имеет *выборочное распределение*  $S$ . Свойства  $S$  определяют качество оценщика. В частности, никогда (за исключением дискретных параметров или искусственных данных) не бывает так, чтобы  $\hat{\theta} = f(\text{данные})$  было равно  $\theta$ . В лучшем случае, когда  $S$  достаточно узкое,  $\hat{\theta}$  может совпадать с  $\theta$  с точностью до нескольких знаков после запятой (и определенно не с полной точностью вычисления).

Существует несколько общих принципов, помогающих получить хороший оценщик. Каждый из них создает некоторую сходимость к  $\theta$ . В качестве рабочего примера рассмотрим оценку параметра Бернулли  $p$ :

$$LL(\{x_i\} | p) = \text{count}(1) \log(p) + (n - \text{count}(1)) \log(1 - p).$$

- ♦ *Принцип подключения* — если  $\theta = g(\text{некоторое распределение } T)$  для некоторой функции  $g$ , то  $f = g(\text{EDF(выборка данных из } T))$  является *подключаемой оценкой* (см. [21.79]). Например, выборочное среднее и медиана являются подключаемыми. Для примера Бернулли:

$$\bar{x} = \frac{\text{count}(1)}{n}.$$

- ♦ *Метод моментов* — некоторые параметры данных являются функциями параметров, которые можно оценить с помощью принципа подключения, — в частности, моменты (например,  $E[X^j]$ , со случайной величиной  $X$ , соответствующей механизму генерации). В случае нескольких простых параметров — таких как нормальное распределение с неизвестным средним значением и дисперсией, это приводит к системе уравнений. Но этот метод не является общим.
- ♦ *Максимальная вероятность* — поиск  $\operatorname{argmax}_{\{\theta\}} LL(\{x_i\}|\{\theta\})$ . Например, если  $\{x_i|\{\theta\}\}$  распределена нормально,  $\hat{\theta} = \bar{x}$  является оценкой (в работе [21.82] приведены примеры производных). Если заменить нормальное распределение на распределение Лапласа, вместо этого вы получите медиану.  $S$  оценок максимального правдоподобия распределены нормально асимптотически при некоторых условиях регулярности (в работе [21.79] «регулярность» означает, что для выявления исключений необходимы сложные математические вычисления). Обычно находят максимум, устанавливая производную логарифмического правдоподобия, равную 0. Для примера Бернулли:

$$\frac{d}{dp} LL(\{x_i\} | p) = \frac{\text{count}(1)}{p} - \frac{(n - \text{count}(1))}{1 - p} = \frac{\text{count}(1) - np}{p(1 - p)},$$

установка которого равным 0 дает  $\bar{x}$ .

- ♦ *Оценка на основе расстояния* — выбрать функцию расстояния  $d$  между предполагаемым распределением выходного сигнала генератора, учитывая параметры, и EDF(данные), а затем найти минимизирующий  $d$  параметр. Например,  $d$  может быть распределен по Колмогорову — Смирнову в одном измерении, и в большем числе измерений звездой в зависимости от наименьшего гиперкуба, содержащего данные. Для дискретных распределений можно использовать расстояния хи-квадрат (все распределения обсуждаются далее в этой главе). Но, несмотря на интуитивную привлекательность, этот параметр сложно вычислить, и используется он мало. Если интересно, в работе [21.6] можно найти больше подробностей.
- ♦ *Байесовская оценка* — предположим, что существует состояние доверия, в котором случайным является  $\theta$ , а не наблюдаемые данные. Учитывая разумно обоснованное (часто называемое *объективным*) *априорное распределение*  $\theta$ , известное до просмотра данных, и сами данные, нужно вывести *апостериорное распределение* с состоянием доверия, которое отражает всю известную информацию (в главе 26. *Машинное обучение: классификация* приведен пример использования теоремы Байеса). Далее, как и в случае с максимальной вероятностью, выберите  $\hat{\theta}$  — точку максимальной PDF (Probability Density Function, функции плотности вероятности). Например, для примера Бернулли можно использовать точку  $\frac{1 + \text{count}(1)}{2 + n}$ . Выбор основан на бета-биномиальной установке с бета(1, 1) априорно (см. [21.97]). К преимуществам этой оценки можно отнести вычислительную стабильность (не допускает деление на 0) и отсутствие чрезмерного оптимизма, когда один из показателей оказывается очень мал.
- ♦ *Принцип минимальной длины описания* (Minimum Description Length, MDL) — поиск кратчайшего представления данных и генератора с заданными параметрами. Число  $x$  может быть представлено с любой фиксированной точностью, используя биты  $O(\log(x))$  и, например, гамма-код (см. главу 15. *Сжатие*). Любые постоянные факто-

ры при выборе лучшего кода не учитываются. По сравнению с параметром максимальной вероятности есть также дополнительный член  $\sum \log(\theta_i)$ . В отличие от байесовской оценки, мы ищем код, а не априорную оценку, а это проще. В работе [21.62] приведена подробная информация. Этот параметр, как и эквивалентная байесовская оценка, имеет преимущества при  $D > 1$ .

Вы можете ориентироваться на любой из приведенных параметров, но лишь на функциях данных, где при известном  $\{\theta\}$  сообщается некоторая дополнительная информация, например  $\text{sign}(x_i - \theta)$  или некоторый рабочий диапазон значений, что позволяет ослабить предположения о распределении  $S$ . Но для преобразования данных в ранги это не работает, т. к. ранги являются глобальным свойством, основанным сразу на всех данных, и поэтому им нельзя присвоить вероятность на основе одного наблюдения и модели.

В принципе, существуют наборы параметров, к которым эти критерии сходятся. Но решение не обязательно должно существовать, быть уникальным, совпадать с истинным параметром в пределе или поддаваться вычислительному анализу. Впрочем, с конкретными оценщиками такие проблемы обычно не возникают. Тем не менее в целом оценка всегда должна подтверждаться — независимо от того, какой принцип использовался для ее получения.

При данных  $\theta$  и  $\hat{\theta}$  существует возможность, по крайней мере теоретическая, задать потери  $L(\theta, \hat{\theta})$  для количественной оценки стоимости нахождения неточного значения  $\theta$ . Обычно используют *среднеквадратичную ошибку* (Mean-Squared Error, MSE) потерь, где  $L(\theta, \hat{\theta}) = (\theta - \hat{\theta})^2$ . Бывают асимметричные значения  $L$ , когда недооценка хуже, чем переоценка, или наоборот. Требуется, чтобы выполнялись следующие условия:

- ◆  $L(\theta, \theta) = 0$ ;
- ◆  $L(\hat{\theta} \neq \theta) > 0$ ;
- ◆  $L$  была выпуклой или квазивыпуклой.

Первые два условия говорят сами за себя, а последнее основано на том, что большие ошибки не должны стоить меньше, чем более мелкие. На выпуклости иногда настаивают из-за ее математических свойств — например, суммы выпуклых функций также выпуклы. Масштабирование  $L$  константой не влияет на получаемые выводы, поэтому нужно выбирать наиболее удобный масштаб. Обычно предполагается, что незначительные изменения в  $L$  мало что меняют в основанных на ней выводах. Практически невозможно предопределить все последствия использования того или иного значения  $\hat{\theta}$ , поэтому любая разумная стратегия должна работать.

Чтобы точно определить подходящую функцию  $L$ , требуются знания предметной области, но обычно использование нестандартного  $L$ , относящегося к пользователю, не позволяет применять общие полезные алгоритмы. Полученный вывод может оказаться некорректным, если  $L$  конечного пользователя сильно отличается от стандартной. Может не существовать аналитических методов для работы с необычными  $L$ , но обычно достаточно моделирования или численной оптимизации. Иногда  $L$  можно использовать для получения оценки, которая минимизирует некоторую совокупную меру риска (обсуждается позже), аналогично выборочному среднему для оценок, основанных на MSE. Но это сложная тема (например, нужно найти некоторое представление оценки и т. д.), и здесь она не обсуждается.

Значения  $\hat{\theta}$  зависят от  $S$  и могут определять риск  $R(\theta) = E_{\hat{\theta}}[L(\theta, \hat{\theta})]$ . С ожиданием легко работать, но в остальном тут нет ничего особенного — можно было бы использовать медиану или 90-й перцентиль, чтобы получить другое разумное определение. При использовании потери  $L_1$  (абсолютная разница)  $R$  становится ожидаемой ошибкой, но даже если она оценивается по данным, то оказывается не столь полезна, как доверительный интервал. Тем не менее при использовании высокого перцентиля риск легче интерпретировать, чем ожидание.

Некоторые желаемые свойства оценщиков не основаны на риске. Оценщик является:

- ♦ *согласованным*, если  $\hat{\theta} \rightarrow \theta$  при  $n \rightarrow \infty$ , т. е. увеличение числа выборок делает оценщик  $f$  точным в пределе. Статистики обожают согласованность, как программисты оптимизацию, но поведение для конечных значений  $n$  может быть совершенно другим. Согласованная оценка не обязательно хороша на практике, но для использования несогласованной оценки требуется хорошее оправдание. Имейте в виду, что термин «асимптотический» в статистике означает, что  $n \rightarrow \infty$ , что не то же самое, что нотация «большого  $O$ », которая игнорирует постоянные множители. Хотя асимптотические аппроксимации обычно достаточно точны, как, например, ряд Тейлора, все процедуры, обладающие только асимптотическими известными хорошими свойствами, нужно проверять с помощью моделирования;
- ♦ *(средне) несмещенным*, если  $E[\hat{\theta}] = \theta$ , и ожидание должно превышать  $S$ . Медианная несмещенность определяется аналогично, и оценщик не может иметь оба вида несмещенности, если  $S$  асимметрична. Несмещенность не зависит от  $\theta$ , но смещенная оценка может иметь разное смещение для каждого  $\theta$ . Значение  $\theta$  переоценивается или занижается в соответствии с  $S$ , и смещение обычно невелико. Байесовские и MDL-оценки почти всегда слегка смещены, но редко настолько, чтобы обращать на это внимание. Для примера Бернулли  $x$  является несмещенным, а байесовская оценка имеет смещение 0 при  $p = 0$  и положительное в противном случае (см. [21.97]). Соотношения оценок вообще не могут быть несмещенными. Среднее смещение легче исправить, потому что на него не влияют, например, монотонные функции оценки;
- ♦ *инвариантным* — преобразование данных разумным набором функций, таких как изменение единиц измерения, должно преобразовать  $\hat{\theta}$  таким же образом (во многих источниках употребляется термин «эквивариантный» с таким же значением). В то время как оценки максимального правдоподобия обладают этим свойством (см. [21.46]), байесовские и MDL-оценки обычно не инвариантны. Вы можете настаивать на приближительной инвариантности, т. е. когда преобразования дают лишь небольшие изменения. Для примера Бернулли байесовская оценка инвариантна к масштабированию и сдвигу.

Оценщики, полученные на основе общих принципов, обычно согласованы, но корректируются в каждом конкретном случае, чтобы получить некоторые другие свойства, которые редко существенно нарушаются. Иногда на устойчивость вычислений вносятся поправки, которые обычно имеют байесовскую интерпретацию.

У несмещенных оценщиков использование в качестве метрики эффективности дисперсии соответствует потерям MSE, потому что  $MSE = \text{смещение}^2 + \text{дисперсия}$  (см. [21.79], а также главу 25. *Основы машинного обучения*, где описаны аналогичные соотношения для других  $L$ ).

Поскольку сходимость связана с пределами функций, а не с числами, в отношении нее вводятся некоторые определения. Все они относятся к асимптотическому распределению выборки  $S(n)$ , когда  $n \rightarrow \infty$  (см. [21.79]):

1.  $L_2$  (в среднем квадратичном):  $E[(\hat{\theta} - \theta)^2] \rightarrow 0$ ;
2. С вероятностью 1 (сильная сходимость):  $\Pr(\hat{\theta} \rightarrow \theta) = 1$ ;
3. В вероятности (слабая сходимость):  $\Pr(|\hat{\theta} - \theta| < \varepsilon) \rightarrow 1$ ;
4. В распределении: распределение  $g(\hat{\theta} - \theta)$  стремится к некоторому предельному распределению для некоторой нормализующей функции  $g$  (например,  $1/\sqrt{n}$ ).

Согласованность определяется с помощью пункта (3). Из (1) и (2) следует (3), а из (3) следует (4), а все остальные следования в общем случае не выполняются. В частности, согласованность не подразумевает асимптотическую несмещенность. Чтобы такое следование было справедливо, нужна более сильная версия согласованности, основанная на сходимости  $L_2$  для желаемого вывода.

При заданном  $\theta$  пусть  $b(\theta)$  — это смещение, а  $FI(x_0, \theta)$  — информация Фишера для одного наблюдения ( $x_0$  без ограничения общности;  $FI$  — вторая производная от  $LL$ . Некоторые примеры приведены в работе [21.79]). В условиях регулярности имеют место неравенства Крамера — Рао (см. [21.51, 21.47]): для любой  $f$ , которая использует  $n$  точек данных:

$$\text{var}(S) \geq \frac{1 + b'(\theta)^2}{nFI(x_0)}.$$

Например, рассмотрим постоянную оценку  $f(\text{data}) = 42$ . При  $\theta = 42$  она идеальна, но  $b(\theta) = 42 - \theta$  и  $b'(\theta) = -1$ . Таким образом, как и ожидалось, дисперсия ограничена 0. Как и в случае с CLT, граница задается для абсолютного количества и не связана с  $\hat{\theta}$  или  $\theta$ .

Некоторые частные случаи выходят за рамки условий регулярности, но этим обычно можно пренебречь. Пусть  $\theta$  находится на границе, т. е. когда есть  $n$  выборок из  $\text{uniform}(0, \theta)$ , и вы хотите оценить  $\theta$ . Здесь  $\hat{\theta} = \frac{n}{n-1} \max(x_i)$  является несмещенной оценкой (см. [21.47]), где дисперсия составляет  $O(n^{-2})$ .

Неравенство порождает ряд выводов:

- ♦ поскольку  $b(\theta)^2 \geq 0$ , это предельное значение MSE;
- ♦ для несмещенного  $f$ ,  $b'(\theta) = 0$ , что дает небольшое упрощение формулы;
- ♦ пока значение  $b'(\theta)$  отделено границей от  $-1$  (например, когда  $b$  липшицево (от Lipschitz) с константой  $< 1$ ), стандартное отклонение  $S$  равно  $O(n^{-1/2})$ . Поскольку смещение обычно составляет  $O(n^{-1})$  (см. [21.79]), это дает, по сути, значение точности работы  $\hat{\theta}$ . Методы MDL позволяют кодировать оценку с этой точностью (см. [21.62]);
- ♦ все оценщики имеют пределы эффективности (по крайней мере, для обычных случаев) по дисперсии  $S$ .

Неравенство редко бывает равенством — исключение составляют только *экспоненциальные семейства* (см. [21.46]). Например, для среднего нормального распределения с известной дисперсией выборочное среднее точно удовлетворяет неравенству, но так не происходит в случае неизвестной дисперсии. Значение границы в основном заключается в существовании предела производительности и асимптотического стандартного отклонения  $O(n^{-1/2})$  для  $S$ .

Граница значительно обобщена. Даже значения разбиения дерева каким-то образом оцениваются. Например, для расчета выборочной дисперсии значения конкретного узла можно использовать начальную загрузку и не ожидать уменьшения лучше, чем  $O(n^{-1})$  (деталь реализации: нужно повторить выборку данных, которые попали в узел, и делать так для каждого узла дерева, которое оценивается на основе всех данных).

Рассмотрение смещения и дисперсии подтверждается *неравенством Чебышева* ([21.79]):

для распределения с конечным  $\mu$  и конечным  $\sigma > 0$ ,  $\Pr(|x - \mu| > k\sigma) < \frac{1}{k^2}$ . Чтобы приме-

нить оценку, замените  $\mu$  на  $\theta$ . В частности, если смещение  $\leq$  фиксированной доли  $\sigma$  и  $\sigma \rightarrow 0$ , оценки приближаются к  $\theta$ . Таким образом, конвергенция в MSE, также называемая конвергенцией в  $L_2$ , для многих других  $L$  означает конвергенцию в рисках.

Но обратная оценка, как правило, неверна — для распределения Коши, которое представляет собой  $S$  для выборочного среднего значения переменных Коши, наличие произвольно малой дисперсии не обязательно ограничивает другие  $L$ . Нужна дополнительная информация, например:

- ◆  $L \geq$  функции дисперсии. Так, если использовать  $MSE^2$ , значение уменьшится как минимум как  $O(n^{-2})$ , что не есть самый лучший выбор. Это говорит о том, что функции  $L$ , такие как  $\text{linex}$  (далее в главе), полезны для расчета оценок, но не обязательны для ранжирования. Параметр MSE полезен, потому что он связан со стандартным отклонением, которое, в свою очередь, связано с длиной универсально применимой конструкции доверительных интервалов Вальда (обсуждается далее в этой главе). А вот при вычислении медианы средняя величина длины доверительного интервала зависит от других оценок, поэтому выбор  $L$  на основе построения интервала в общем случае не работает;
- ◆ предположим, что задано конкретное  $S$  — например, нормальное распределение, где все положения квантилей определяются или ограничиваются функцией стандартного отклонения  $S$ . Любое  $L$  является квазивыпуклым и  $> 0$ , поэтому даже хвосты размера  $a$  имеют по крайней мере частичный вклад размера  $a(L(\text{левое расположение}(a, \text{дисперсия}) + L(\text{правое расположение}(1 - a, \text{дисперсия})))$  в  $R$ . Можно вычислить максимум от этой функции по всем  $a$ . Например, если эти зависящие от местоположения  $L$  линейны по стандартному отклонению, они также асимптотически ограничены сверху  $O(n^{-1/2})$ . Нелинейная зависимость приводит к различным асимптотикам, которые трудно вычислить, но даже при неизвестной, но конечной скорости  $L$  не может сколь угодно быстро убывать по  $n$ .

Асимптотическое поведение может несколько отличаться от поведения с конечным  $n$ :

- ◆ согласованность означает, что  $R \rightarrow 0$  (нужны некоторые условия регулярности — такие как функция  $L$  по Липшицу);
- ◆ граница Крамера — Рао удовлетворяется большим количеством оценок, потому что константы сходятся.

Интересное свойство оценщиков состоит в том, что их работа основана на *достаточной статистике* — т. е. на функции выборки, которая содержит всю информацию о ней относительно  $LL$  (технически должна быть возможностью определенным образом учитывать влияние  $LL$  — примеры есть в работе [21.82]). Например, для оценки среднего значения и дисперсии нормального распределения могут применяться сумма и сумма квадратов. В общем случае для использования свойства достаточности нужна параметрическая модель. В лучшем случае нужно задействовать:

- ◆ всю статистику упорядоченной выборки (т. е. по отсортированным данным). Сортировка позволяет, например, рассчитать медиану. Для  $D > 1$  выполнить сортировку нельзя, поэтому есть только данные;
- ◆ число уникальных значений в дискретных распределениях.

Главная мысль состоит в том, что нужно каким-то образом использовать все данные, и это требование более базовое, чем достаточность относительно некоторого распределения. Например, усеченного среднего, которое является заведомо хорошей оценкой, в общем случае недостаточно, хотя при его расчете используются все данные.

Оценщики обычно выполняют производные оценки — такие как доверительные интервалы и тесты гипотез, позволяющие оценить те или иные аспекты распределения  $S$  лучше, чем его дисперсия. Но сами по себе они не имеют смысла, потому что не нужны, если известно  $\theta$ .

Выбранный оценщик должен:

- ◆ удовлетворять любым внешним требованиям — особенно если результат является промежуточным и передается другому расчету. С некоторыми оценщиками, такими как выборочное среднее, связана специальная теория (например, CLT или неравенства конечной выборки, которые рассматриваются далее в этой главе), но зато они могут быть более значимы для анализа.
- ◆ быть статистически эффективным (подробнее далее в этой главе);
- ◆ обладать общими качествами полезного алгоритма.

Интересно, а что нам дает оценка объекта, если оценщик не удовлетворяет условиям? Полученная *описательная статистика* часто бывает полезна сама по себе, независимо от составляющих ее параметров. Например, усеченное среднее для асимметричного распределения не совпадает со средним или медианой. При выполнении сравнений любая метрика местоположения (обсуждаемая далее в этой главе) полезна, а усеченное среднее обычно является наиболее эффективно оцениваемой статистикой, поэтому в первую очередь можно попробовать рассмотреть ее.

Имея выборку данных из известного распределения, можно выполнить моделирование для проверки свойств любой оценки (включая доверительные интервалы и тесты).

Любые выводы, основанные на данных, столь же полезны, сколь и сами данные. Подобно тому, как тестирование программного обеспечения не может гарантировать отсутствие ошибок, статистическая процедура, основанная на независимых случайных данных, не может гарантировать абсолютно точную оценку. Но даже на законодательном уровне статистические результаты являются допустимым доказательством, хотя и не единственно достаточным. Статистические процедуры работают в определенных условиях, которые могут отличаться от желаемых, даже если результаты интерпретируются желаемым образом. Часто в задаче нет формально сформулированных условий

корректности, что усложняет работу и потенциально увеличивает затраты времени на исследование. Если в нескольких книгах о таких условиях ничего не говорится, то их, скорее всего, не существует хотя бы в общем виде. Имеющиеся сегодня инструменты доказательства статистической теории ограничены в основном гладкими задачами и асимптотическими результатами, и исследователь должен сам вывести более полезные теоремы. Практическое применение некоторых процедур, таких как начальная загрузка (обсуждается далее в этой главе), опирается в первую очередь на практический опыт.

## 21.4. Поиск наиболее эффективных оценщиков

При выборе оценщика будем учитывать:

1. Существование и уникальность оценщика с наименьшим риском.
2. Возможность понять, какой оценщик лучше.
3. Возможность найти лучшего оценщика.

Ответ на любой из этих вопросов должен сводиться к конкретному  $S$  или распределению данных, и в общем случае каждое отдельное распределение среди неисчислимого их множества может дать свой ответ. Нужно выбрать предположительно репрезентативное  $S$ , которое обычно оказывается удобным с математической или иной точки зрения, и эвристически (зачастую необоснованно) предположить, что выводы в целом верны, по крайней мере, для текущей задачи. Обычно  $S$  — это нормальное распределение, потенциально преобразованное некоторой функцией.

Даже с учетом общих ограничений в общем случае могут возникнуть проблемы, поэтому сначала стоит сосредоточиться на несмещенных оценках. Для пункта (2) есть *теорема Лемана — Шеффе* (см. [21.46]): оценка является несмещенной с наименьшим риском  $\forall \theta$  при использовании выпуклой  $L$ , если она:

- ♦ является функцией достаточной статистики;
- ♦ *полная* — математическое требование, которое делает несмещенную оценку уникальной (см. [21.66]).

Но лучшая несмещенная оценка не обязательно должна быть лучшей в целом. Например, для примера Бернулли при  $p = 0,5$  байесовская оценка имеет нулевое смещение и меньшую дисперсию (см. формулу в [21.97]), чем  $x$ , которая является полной (см. [21.46]), поэтому и применяется теорема Лемана — Шеффе. Для близкого значения  $\theta$  MSE байесовской оценки снижается, пока не будет достигнуто достаточное удаление в сторону хвостов. Эта байесовская оценка далеко не уникальна — например, у вас могут быть даже дробные приращения, и все они инвариантны к смещениям местоположения и к масштабированию. Полная оценка, основанная на достаточной статистике, может не существовать, и в этом случае может не существовать и хорошая несмещенная оценка. Ее существование, по сути, ограничено классическими параметрическими нормальными оценками.

Функция  $L$  может быть очень асимметричной функцией вроде *linex*, у которой  $L(d) = e^{ad} - ad - 1$ , где  $d = \hat{\theta} - \theta$ , а  $a$  — это параметр масштабирования (он не должен быть равен 0 и может быть инвертирован, что ведет к перевороту графика), обычно равный 1 (см. [21.66]). В работе [21.26] приведено еще несколько распространенных вариантов. Наилучшая несмещенная оценка не меняется, потому что она не зависит



от  $L$ . Таким образом, для использования только несмещенных оценок требуется хорошее обоснование. Тем не менее определить точное значение  $L$  для предметной области практически невозможно, поэтому приходится работать с аппроксимацией, позволяющей выполнять приближенную оптимизацию.

В общем случае основная проблема заключается в том, что, как и у байесовской оценки в примере Бернулли, для разных значений  $\theta$  могут существовать разные риски. То есть нужно оценить совокупный риск по всем  $\theta$ . Обычно используют следующие метрики:

- ◆ *максимальный риск* — защита от наихудшего случая;
- ◆ *байесов риск* — при некотором распределении по  $\theta$  риск равен  $= E\theta[R(\theta)]$ . Например, дельта плотности вероятности по значению максимального риска дает максимальный риск, поэтому байесовский риск описывает более общий случай. Также можно использовать *объективные распределения*: например, для ограниченных диапазонов (когда есть ограничения, связанные с предметной областью) — однородное, а для неограниченных распределений — с толстым хвостом вроде распределения Коши (важно определить ожидание).

Если используется максимальный риск, общая теорема теории игр (равновесие Нэша) говорит о существовании *минимаксной оценки*:

- ◆ вычисляется с наименьшим максимальным риском;
- ◆ может быть рандомизированной, т. е. создавать разные  $\hat{\theta}$  из одних и тех же данных (как и в случае с камнем-ножницами-бумагой, где рандомизация приводит к безопасной стратегии);
- ◆ игра заключается в том, что у вас есть распределение с неизвестными параметрами и  $n$ , и выбирается оценщик. Затем выбирается  $\theta$  (после этого выполняется получение данных, но здесь это не имеет значения).

Использование рандомизации выглядит подозрительно, поскольку при анализировании одного и того же набора данных можно получить разные результаты. Но сами данные должны быть гораздо большим источником случайности. Можно представить, что мы связываем некоторую случайную информацию с каждым конкретным набором данных. Для научной воспроизводимости генерация другого набора данных также должна генерировать другую случайную информацию. Но все же неясно, как проверить выполненный ранее анализ.

Минимаксные оценщики часто не существуют или неочевидны даже в простых случаях. Например, в примере Бернулли оценка  $x$  не является минимаксной, а вот более

сложная байесовская оценка  $\frac{n\bar{x} + \sqrt{n/2}}{n + \sqrt{n}}$  является (см. [21.70]). Минимаксный оценщик

определить сложно. Существует полезная теорема, которая гласит, что байесовские оценки с постоянным риском являются минимаксными (см. [21.46]). Именно так была найдена приведенная ранее оценка). Интересная идея состоит в том, чтобы имитировать логику алгоритмов аппроксимации и определить минимаксные оценки, которые отличаются от минимаксной производительности не более чем на постоянный множитель  $c$ , но в литературе эта идея не рассматривалась.

Байесовский риск немного нечестный в том смысле, что оптимальной оценщик окажется байесовским оценщиком, использующим распределение риска в качестве априорной

оценки. Но сам по себе байесовский оценщик не дает уникальную оценку, потому что необходимо каким-то образом взвесить данные. Например, в случае бета-биномиала любая бета(счет, счет) является равномерной.

При использовании максимального риска можно получить удовлетворительный результат в худшем случае и избежать мошенничества — такого как размещение дельта плотности вероятности на значении наименьшего риска. Но байесовский риск также полезен:

- ♦ оценщики с одинаковым максимальным риском могут иметь разные риски во всех остальных точках, а минимаксные оценки могут оказаться недопустимыми (обсуждается позже). Тем не менее оценки с большими различиями между значениями риска для разных значений  $\theta$  вызывают подозрение, поэтому обработка таких случаев не имеет практического значения. Возможно, лучше всего работать с этой ситуацией как с интерполяцией полиномами Чебышева или минимаксными полиномами в численном анализе (см. главу 23. Численные алгоритмы: работа с функциями) — первые выигрывают в среднем, но немного проигрывают в максимуме;
- ♦ байесовский риск легко оценить с помощью моделирования, используя мультидельта плотности вероятности для случайно или определенным образом выбранных точек. В простых случаях, например, для нормального среднего или медианы, можно работать только с одним репрезентативным значением  $\theta$  — в нашем случае 0 (и не использовать глупые оценки, такие как  $f(\{x_i\}) = 0$ );
- ♦ выбор  $L$  также субъективен, и важно понимать общий эффект этой самой функции и метода агрегирования.

Например, в случае минимизации совокупного риска функция  $\text{linex}$  выберет меньшее значение вместо большего, насколько позволяет неопределенность. Здесь для нормаль-

ной модели с известным  $\sigma$  оценщик  $\bar{x}$  уступает место  $\bar{x} - \frac{\sigma^2}{2n}$  (см. [21.66]). При рабо-

те с необычными  $L$  помогают численные методы. Например, вы можете оценить оценщик для конкретной  $L$  и функцию агрегирования для конкретного распределения данных. Проще всего это сделать, когда одно значение  $\theta$  является репрезентативным, — например, если вокруг него предполагается нормальное распределение (но другие распределения тоже подходят). Это даже позволяет искать наилучших оценщиков численно в некотором функциональном представлении — например, веса для линейной комбинации порядковых статистик с фиксированным  $n$ . Возможно, вам удастся обобщить результаты аналитически и найти таким образом хорошего оценщика.

Найти и даже распознать наилучшего оценщика сложно, а часто и невозможно даже в несмещенном случае. Рассмотрим более простое сравнение оценщиков по значению риска. Вместо агрегирования можно определить границу Парето (см. главу 16. Комбинаторная оптимизация) оценщиков с точки зрения риска, который на ней является допустимым. Использование недопустимых оценщиков требует хорошего обоснования, но допустимость автоматически не подразумевает качество — например, оценщик-константа допустим, но не является качественным.

Кроме того, при использовании MSE дисперсия является недопустимым оценщиком, потому что вместо применения максимального правдоподобия делитель  $1/n$  в формуле дисперсии теряет тем больше, чем увеличивается систематическая ошибка<sup>2</sup> (см. [21.66]). Но использование этого делителя, по сути, является известной ошибкой

из-за внешнего ограничения, которое оценивает дисперсию не для обнаружения истинного значения, а для получения доверительного интервала на основе  $t$ -распределения (обсуждается далее в этой главе), которое предполагает несмещенную оценку. Интересно, что полученное стандартное отклонение выборки на самом деле оказывается смещенным (см. [21.82]), но это не имеет значения. Таким образом, атрибуция ошибки не связана с выбором несмещенных оценок. Например, в ЕМ-кластеризации (см. главу 28. *Машинное обучение: кластеризация*) намеренно используется смещенный оценщик максимального правдоподобия, чтобы теория работала. Кроме того, при реализации доверительного интервала начальной загрузки ВС (обсуждается далее в этой главе) используется байесовский оценщик параметра Бернулли вместо среднего — чтобы избежать выхода за границы диапазона. Как правило, несмещенная оценка приводит к нарушению ограничений диапазона, а исправление этой ситуации в итоге приводит к смещению. В некоторых случаях хороших несмещенных оценок не существует (примеры есть в работе [21.51]).

Свойства, внешние по отношению к риску, такие как несмещенность и инвариантность, по-прежнему желательно получить, даже если они не диктуются внешними ограничениями, и ради этого можно даже допустить небольшой риск:

- ◆ получить точные доверительные интервалы для несмещенных оценщиков проще — например, в начальной загрузке (обсуждается далее в этой главе) самые быстрые методы, основанные на нормальном распределении, предполагают, что смещение достаточно мало, и не утруждают себя его исправлением. Это важно, потому что у оценщика всегда есть доверительный интервал;
- ◆ определенные типы инвариантности помогают в случаях, если позже каким-то образом нужно преобразовать оценки;
- ◆ оба свойства обеспечивают некоторую регуляризацию (подробнее об этой концепции рассказано в главе 25. *Основы машинного обучения*) выбора  $L$  для обеспечения низкого совокупного риска. Точное выполнение этих свойств дает такой же эффект.

Количественные сравнения более значимы, чем качественные, поэтому сравнение обычно выполняют с использованием *относительного совокупного коэффициента риска* без единиц измерения. В асимптотическом случае, если MSE и смещения достаточно малы, чтобы их можно было игнорировать, отношение дисперсии  $S$  двух оценок является *асимптотической относительной эффективностью* (Asymptotic Relative Efficiency, ARE) (см. [21.93]). Аналогичным образом можно использовать отношение размера выборки, необходимого для получения равной дисперсии. Этот метод также хорошо работает для проверки гипотез, доверительных интервалов и других функций  $L$ . Теоретически эти свойства уже определены для многих сравнений оценок. Например, когда несколько функций данных оценивают один и тот же параметр, например среднее, медиану и усеченное среднее (обсуждается далее в этой главе), оценивается среднее симметричного распределения (там, где оно существует). Хотя отношения могут иметь конечное значение  $n$ , в частности, рассчитанное с помощью моделирования, асимптотические легче вывести аналитически, и они все же достаточно предсказуемы.

На практике выбирается оценщик, который редко бывает слишком плохим, а для типичных данных работает хорошо. Оценщики, которые работают в редких случаях очень хорошо, а в остальных случаях — плохо, могут быть полезны только при условии наличия знаний в предметной области. То же самое справедливо для доверительных интервалов и тестов.

## 21.5. Некоторые особенности асимптотики

В принципе, асимптотическая аппроксимация выполняется прямолинейно. Например, CLT показывает все нужное. Но бывают и сюрпризы:

- ♦ возможность и эффективность приближений более высокого порядка — например, CLT можно сделать более точным, используя третий и четвертый моменты (подробности см. в работе [21.20]).
- ♦ некоторые математические проблемы — например, согласованность не означает более сильной сходимости. Обычно такие ситуации не считаются статистически важными и далее рассматриваться не будут.

В случае с CLT ошибка аппроксимации по *теореме Берри — Эссена* составляет  $O(1/\sqrt{n})$  (зависит от третьего момента — см. [21.20]). Аппроксимации с такой ошибкой имеют *первый порядок точности*. Включение большего количества моментов снижает ошибку аппроксимации в  $\sqrt{n}$  раз. Например, аппроксимация *второго порядка* имеет ошибку аппроксимации  $O(1/n)$ , а ошибку оценки  $O(1/\sqrt{n})$ .

Поскольку ошибка оценки не уменьшается, а моменты более высокого порядка или другая информация должны существовать и быть оценены, их используют редко. Второй порядок в общем случае лучше, чем первый, но это зависит от конкретного случая и проверяется моделированием. Ошибка аппроксимации характерна только для асимптотического объекта и может не наблюдаться при конечном  $n$ , точно так же, как для ряда Тейлора требуется, чтобы  $x \rightarrow 0$  компенсировало влияние конкретной неизвестной аппроксимируемой функции и приводило ее к известному многочлену. Как и в случае с рядом Тейлора, использование произвольного количества членов для приближения с конечным  $n$  может привести к расхождению, но на практике точность не выше второго порядка таких рисков не создает.

## 21.6. Оценка нормальной CDF

Нормальное распределение встречается чаще всего, особенно асимптотическое, как видно из *центральной предельной теоремы* (Central Limit Theorem, CLT). Но иногда использования параметра  $z = 2$  для доверительных интервалов недостаточно. Поэтому необходимо вычислить значение  $z$ , соответствующее определенному уровню достоверности, и наоборот, особенно при поправке на множественное тестирование (обсуждается далее в этой главе).

Расчет сводится к вычислению функции ошибки из библиотеки `<cmath>`, добавленной в C++ 11. В статье [21.86], если любопытно, можно найти несколько простых и разумных приближений, но стандартная реализация должна быть лучше. Однако нужно масштабировать аргумент, а односторонний и двусторонний случаи немного отличаются:

```
double approxNormalCDF(double x)
{
    double uHalf = erf(abs(x)/sqrt(2))/2;
    return 0.5 + (x >= 0 ? uHalf : -uHalf);
}

double approxNormal2SidedConf(double x){return 2 * approxNormalCDF(x) - 1;}
```

Чтобы найти значение  $z$ , соответствующее определенной достоверности, используйте метод инверсии. Он гарантированно работает, потому что CDF (Cumulative Distribution Function, совокупные функции распределения) монотонны. Эффективность метода тоже достаточно хороша, потому что у нормального распределения тонкие хвосты, поэтому на практике оно редко выходит за рамки нескольких стандартных отклонений:

```
double find2SidedConfZ(double conf)
{
    assert(conf > 0 && conf < 1);
    return invertCDF([](double x){return approxNormalCDF(x);}, 0.5 + conf/2);
}
```

## 21.7. Оценка CDF Т-распределения

Т-распределение с  $\nu = n - 1$  степенями свободы моделирует среднее значение нормальных выборок размера  $n$  с поправкой на оценку  $\sigma^2$  (см. [21.95]). Хотя при  $n > 30$  нормальное приближение считается точным, может потребоваться меньшее значение  $n$  или более высокая точность.

Для  $\nu \geq 3$  аппроксимация Хилла преобразует значение  $t$  в значение  $az$  и имеет наихудшую ошибку  $O(10^{-5})$  (где в «О» скрывается небольшая постоянная составляющая).

$$z = w + \frac{w^3 + 3w}{b} - \frac{(4w^7 + 33w^5 = 240w^3 = 855w)}{10b(b + 0.8w^4 + 100)},$$

где

$$w = \sqrt{a \log\left(1 + \frac{t^2}{n}\right)}, a = n - 0.5 \text{ и } b = 48a^2 \text{ (см. [21.11])}.$$

Для  $\nu \leq 2$ :

$$TCDF(t, 1) = \frac{1}{2} + \frac{1}{\pi} a \tan(t) \text{ и } TCDF(t, 2) = \frac{1}{2} + \frac{t}{2\sqrt{(2 + x^2)}} \text{ (см. [21.103])}.$$

Для чуть более простой аппроксимации Глисона рекомендуются более высокие значения ошибок для  $\nu = 3$ ):

```
double approxTCDF(double t, int v)
{ // Метод Hill3 всегда дает результат < 10-4?
    assert(v > 0);
    if(v == 1) return 0.5 + atan(t)/PI(); // Точное значение
    if(v == 2) return 0.5 + t/2/sqrt(2 + t * t); // Тоже точное
    double a = v - 0.5, b = 48 * a * a, w = sqrt(a * log(1 + t * t/v)), w27[6];
    w27[0] = w * w;
    for(int i = 1; i < 6; ++i) w27[i] = w * w27[i - 1];
    double z = w + (w27[0] + 3) * w/b - (4 * w27[5] + 33 * w27[3] +
        240 * w27[1] + 855 * w)/(10 * b * (b + 0.8 * w27[2] + 100));
    return approxNormalCDF(t > 0 ? z : -z);
}

double approxT2SidedConf(double x, int v){return 2 * approxTCDF(x, v) - 1;}
```

Чтобы найти значение, соответствующее определенной уверенности, используйте метод инверсии, как и для нормального распределения:

```
double find2SidedConfT(double conf, int v)
{
    assert(conf > 0 && conf < 1 && v > 0);
    return invertCDF([v](double x){return approxTCDF(x, v);}, 0.5 + conf/2);
}
```

## 21.8. Подробнее о доверительных интервалах

Еще одной хорошей мерой точности оценки является размер соответствующего *доверительного интервала*. В более общем случае существует *доверительный набор*, который представляет собой непересекающийся интервал или *доверительную область* для многомерных параметров, но эти параметры менее полезны. Определяемый зависимыми от выборки границами  $l$  и  $u$ , доверительный интервал — это такой интервал, что  $\Pr(\theta \in [l, u](\text{sample})) \geq 1 - \alpha$ . Почти всегда используют значение  $\alpha = 0,05$ , что дает доверительный интервал 95%. Эта допустимая погрешность накладывается на статистические методы извне как максимальная общепринятая в науке ошибка. С точки зрения понимания доверительный интервал — это место, где, «по мнению» данных, находится значение  $\theta$ , притом что у разных данных могут быть разные «мнения». Часто в качестве единственной меры точности  $\hat{\theta}$  ищут только  $u - l$  или  $\max(u - \hat{\theta}, \hat{\theta} - l)$ . Любая возникающая неопределенность связана с самими данными, но иногда также возникает из-за рандомизации в оценщике (например, в результате моделирования). В частотной статистике речь идет о долгосрочных свойствах процедур, а не об их индивидуальных результатах. Хотя существуют плохие конструкции доверительных интервалов, частотная валидация никогда не вызывает сомнений.

Например, если известна дисперсия  $S$ , функция  $f$  является несмещенной и предполагается, что она вычисляет среднее значение, неравенство Чебышева дает  $(l, u) = \hat{\theta} \pm \sigma \sqrt{\frac{2}{\alpha}}$ .

Даже если оценка  $\sigma$  взята по данным и  $f$  имеет небольшую погрешность или результат сильно отличается от среднего, как правило, получается очень консервативный интервал, потому что неравенство Чебышева обычно очень слабое. Таким образом, если можно оценить дисперсию  $S$ , часто делают более сильное предположение, что  $S$  является нормальным распределением, и используют  $(l, u) = \hat{\theta} \pm 2s$ . Это *интервал Вальда*. На практике многие  $\hat{\theta} \sim$  нормальные или почти нормальные. Дисперсию можно оценить с помощью начальной загрузки, если не удастся рассчитать ее напрямую.

Так как интервал  $(-\infty, \infty)$  допустимый, но бесполезный, желательно, чтобы доверительные интервалы были «короткими», что обычно измеряется одним из следующих параметров:

- ♦ ожидаемая длина — для односторонних интервалов используется расстояние до желаемой границы от  $\theta$ . Для более общих областей возьмите соответственным образом определенный объем. Обычно односторонние интервалы не используются, и часто лучше всего вычисляются путем сведения к двусторонним интервалам, поэтому им уделяется мало внимания. Средняя длина иногда является лучшим критерием, но мы ее не рассматриваем;

- ♦ *точность* — вероятность покрытия различных значений  $\hat{\theta}$  при заданном  $\theta$ . Она позволяет, среди прочего, определить допустимость вероятностным включением множества (детерминированное включение невозможно из-за случайного характера доверительных интервалов) — доверительный *интервал равномерно более точен*, чем другой, если  $\forall \hat{\theta} \neq \theta$  точность выше. В нескольких частных случаях можно получить *равномерно наиболее точный* (Uniformly Most Accurate, UMA) интервал — т. е. односторонние интервалы для однопараметрических экспоненциальных семейств, таких как биномиальные (см. [21.70]).

Существуют и другие желательные требования к доверительным множествам, но они не являются обязательными:

- ♦ требование получить неразрывную, а для  $D > 1$  — даже выпуклую область. Но для задач вроде полета птицы вокруг препятствия есть бимодальное распределение, где имеет смысл формировать доверительный набор из двух частей;
- ♦ двусторонние интервалы требуют симметричного интервала с ошибкой  $\leq a/2$  на каждой стороне. Для  $D > 1$  это должно выполняться в каждом измерении.

К счастью, общее определение обычно ведет к автоматическому выполнению этих требований:

- ♦ покрытие становится сбалансированным из-за убывающей величины  $\partial a / \partial \text{length}$ , когда  $a$  мало. Для  $D > 1$  можно определить баланс в определенном направлении.
- ♦ отмеченные интервалы дешевле закрыть, чем расширять их в стороны дальше;
- ♦ для задач вроде полета птицы (см. главу 6. *Генерация случайных чисел*) эта логика не работает, потому что центральные области характеризуются малыми вероятностями.

Интервал, скорее всего, охватывает значение  $\hat{\theta}$ , которое обычно является точкой максимального правдоподобия и может выступать в качестве оценки ошибки. Но в целом доверительные интервалы и оценки связаны слабо и не напрямую. Например, оценщик может быть центроидой или наиболее часто включаемой точкой доверительного набора, но не иметь отношения конечной выборки.

Несмотря на то что с точки зрения статистической эффективности предпочтение отдается равным значениям  $a$ , в конкретной задаче может потребоваться предположение о том, что одна сторона должна иметь большую ошибку из-за асимметричных затрат. Но большинство API (включая описанные в этой главе) этого не поддерживают. Тот же эффект можно получить, вычислив два интервала, соответствующих искомому, и объединив их. Тем не менее это решение потенциально затратно с вычислительной точки зрения, а при рандомизированных построениях смысла от него мало.

Использование ожидаемой длины в качестве меры помогает избежать дыр, но при определенной точности форма не имеет значения. Первое также легче изучать с помощью моделирования, потому что последнее в основном является относительной величиной. Оба параметра не следуют общей стратегии сравнения оценок, основанной на функциях потерь и рисков, но некоторые интуитивно понятные функции потерь будут рассмотрены позже в этой главе. Если вы управляете значением  $a$ , ожидаемая длина может рассматриваться как риск и сравниваться с оценщиками, ориентируясь на минимакс, допустимость и другие величины, зависящие от  $\theta$ . Тем не менее этот метод лучше всего

задействовать при моделировании с использованием байесовского риска с репрезентативными точками. В литературе оптимальность в основном проверяется по точности.

Поскольку в некоторых случаях можно получить интервалы UMA, например у оценщиков, можно рассмотреть ограниченные классы, наложив следующие требования (см. [21.70]):

- ◆ *несмещенность* — для доверительных интервалов это зависит от точности, а смещение равняется наиболее часто включаемой точке  $\theta$ . При выполнении оценки может появиться небольшое случайное смещение, которое мало на что влияет. Чтобы рассчитать его с помощью моделирования, нужно сгенерировать множество доверительных интервалов и вычислить результирующую гистограмму, используя проход слева направо по отсортированным конечным точкам (это хороший вопрос для собеседования). Середина области с наибольшим количеством значений — это оценка наиболее часто включаемой точки;
- ◆ *инвариантность* — по отношению к взаимно однозначным преобразованиям  $\theta$ , особенно к операциям переноса и масштабирования. Это более общий критерий, чем оценка.

Как и в случае с оценками, эти свойства имеют интуитивно понятный смысл, но не обязательно должны давать хороший результат — например, слегка смещенный интервал может лучше охватывать как истинные, так и ложные значения. Но многие классические конструкции доверительных интервалов являются *несмещенными по UMA* (UMA Unbiased, UMAU) (см. [21.70]), в том числе:

- ◆ T-интервал распределения среднего по выборке;
- ◆ интервал на основе критерия знака для медианы выборки.

Ожидаемая длина и точность связаны *тождеством Госта — Прамта* (см. [21.14]), которое применяется к объемам конечной длины с разрешенными отверстиями. Оно означает, что интервалы UMAU имеют наименьшую ожидаемую длину среди несмещенных интервалов.

Значения  $\alpha < 0,05$  в общем случае не работают по следующим причинам:

- ◆ любые незначительные отклонения от предположения — такие как достаточно быстрая сходимость к норме у интервалов на основе CLT, также делают интервал приблизительным. Ошибки аппроксимации тоже обычно увеличиваются ближе к хвостам, т. е. при малых  $\alpha$ . Таким образом, увеличение доверительного интервала дополнительной выгоды не приносит;
- ◆ если вычисленный интервал не соответствует  $\theta$ , маловероятно, что изменения интервала помогут исправить ситуацию. Итак, если использовать размер доверительного интервала как оценку ошибки  $\hat{\theta}$ , значение  $\alpha = 0,05$  или близкое значение дает тот же порядок результата.

Меньшие значения  $\alpha$  обычно мало влияют на интервал. Например, для интервалов на основе CLT значение  $\alpha = 0,0027$  приводит к увеличению интервала на 50% по сравнению с  $\alpha = 0,05$  ( $z = 2$  против  $z = 3$ ). Таким образом, для больших  $n$  и интервалов с минимальными предположениями, таких как интервалы, основанные на квантилях (обсуждаемые далее в этой главе), значения  $\alpha \leq 0,01$  могут быть оправданы.

Не для каждой оценки распределение  $S$  асимптотически нормально. Например, оно может быть мультимодальным. Но даже в этом случае нормальный интервал полезен



как грубая оценка ширины для непротиворечивой оценки  $u - 1 \rightarrow 0$ . Для промежуточных шагов, когда оценка будет использоваться где-то еще, доверительные интервалы не нужны.

$\hat{\theta}$  получает мажоритарные значения в диапазоне  $(S_a, S_{1-a})$ , но это не доверительный интервал для  $\theta$  — он ограничивает только значения  $\hat{\theta}$  (рис. 2.1).

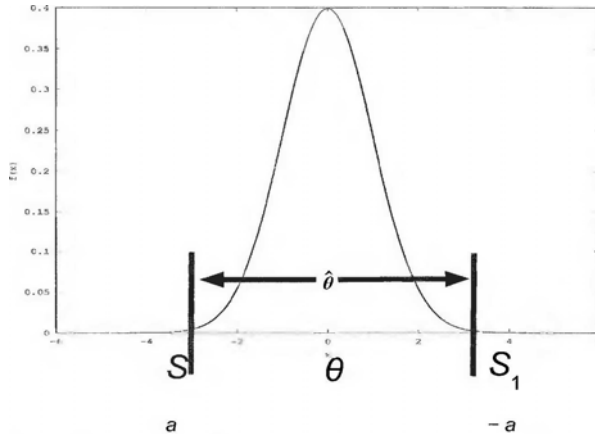


Рис. 21.1. Границы  $S$

Наилучший (точный) доверительный интервал для  $\theta$  равен  $(\theta - (S_{1-a} - \hat{\theta}), \theta + (\hat{\theta} - S_a)) = (\hat{\theta} + \theta - S_{1-a}, \hat{\theta} + \theta - S_a)$  (см. [21.54]). На левой/правой границе значения  $\theta$  соответственно занижены/завышены. То есть если  $\hat{\theta} = S_a$  по отношению к другим выборкам  $(\theta - (S_{1-a} - S_a), \theta)$  образует правильный интервал, то же самое делает  $(\theta, \theta + (S_{1-a} - S_a))$  при  $S_{1-a} = \hat{\theta}$ . Даже при неизвестном  $\theta$  эта конструкция полезна для сравнения различных доверительных интервалов теоретически и путем моделирования.

Доверительные интервалы дают информацию о значении  $\theta$  только в рамках выбранной модели, что не может быть хорошим приближением. Например, доверительный интервал для медианы всегда корректный. ARE и другие меры сравнения также имеют отношение только к модели. Только тестирование модели с другими данными позволяет понять, как нарушения предположений влияют на доверительные интервалы. Например, доверительный интервал для коэффициента линейной регрессии измеряет только ошибку оценки. Он ничего не говорит о влиянии на некоторую переменную других переменных (см. главу 27. *Машинное обучение: регрессия*), даже если ошибка прогноза очень мала. В центре внимания статистики находятся свойства оценок параметров модели, независимо от ее прогностической полезности. Если вычисленные доверительные интервалы слишком широки, нужно больше данных, и размер выборки надо выбрать более тщательно (см. далее в этой главе)

Учитывая доверительные интервалы правильного значения  $a$ , нужно выбрать интервал с наименьшей ожидаемой длиной. Здесь эквивалентом допустимости является вероятностное включение, т. к. включенный интервал явно будет хуже. Но если процедура немножко выходит за границы  $a$ , это значение может оказаться приемлемым, если про-

изведенный интервал намного короче, чем у конкурентов. Это можно сделать с помощью функции потерь, но стандартная функция вроде  $l = \text{функция (длина)} + \text{достигнутое } a$  (см. [21.46]) не является ни популярной, ни интуитивно понятной.

Интуитивно *нормальная потеря* определяется выражением  $l = E[\text{длина}] / (\text{показатель } z, \text{ соответствующий достигнутому } a)$ . Длину можно сделать относительной путем деления на длину точного интервала, если последний уже известен или получен с помощью моделирования. У такой функции потери есть проблемы:

- ◆ она не учитывает запрошенное значение  $a$  — например, для интервалов, основанных на нормальной теории, интервал 95% так же хорош, как 99,73%, но при этом выбирается последний, который в 1,5 раза больше;
- ◆ у функции нет штрафов за асимметричное покрытие — желательно, чтобы с каждой стороны достигалось значение  $a/2$ , чтобы избежать смещения, поэтому нужно снижать вес процедур, которые пытаются сократить длину, сдвигаясь к более щедрой стороне;
- ◆ нормальное распределение несправедливо по отношению к очень малым значениям  $a$ .

Последняя проблема решается с помощью *потерь по Чебышеву*:

$$l = \frac{E[\text{длина}]}{\sqrt{1 / a_{\text{достигнутое}}}},$$

которые вычисляются по инвертированному неравенству Чебышева. Для асимметричного покрытия лучшим вариантом является:

$$l = \frac{E[\text{длина слева}]}{\sqrt{0.5 / a_{\text{слева}}}_{\text{достигнутое}}} + \frac{E[\text{длина справа}]}{\sqrt{0.5 / a_{\text{справа}}}_{\text{достигнутое}}}.$$

Длина стороны вычисляется от  $\theta$  и в качестве дополнительного незначительного штрафа равняется нулю, если значение  $\theta$  с этой стороны не охвачено. Тем не менее асимметричная потеря по Чебышеву является лишь хорошей мерой эффективности отказа от ожидаемой длины.

Правильная функция потерь должна учитывать тот факт, что значение  $a$ , хотя и выбирается произвольно, его тем не менее следует соблюдать. Потеря по Чебышеву предлагает сделать штраф мультипликативным. Таким образом, экспоненциальная зависимость:

$$l = E[\text{длина}] \exp \left( \max \left( \frac{a_{\text{достигнутое}}}{a_{\text{целевое}}} - 1, 0 \right) \right)$$

кажется разумным вариантом. Асимметричная версия тоже может быть вычислена по Чебышеву.

## 21.9. Границы среднего значения в конечной выборке

Теорема о центральном пределе является асимптотической, и даже без редких событий сходимость к нормальному распределению может занять некоторое время — в некото-

рых простых случаях требуется  $n \geq 600$  (см. [21.102]). Если данные принадлежат диапозону  $[0, 1]$ , на  $\mu$  накладываются ограничения для любого  $n$ . В более общем случае для  $x_i \in [a, b]$  выполняется отображение  $x_i$  на диапазон  $[0, 1]$ , вычисляется оценка, и она уже отображается на диапазон  $[a, b]$ , что работает за счет линейности ожидания.

**Неравенство Хёффдинга:** Пусть  $\Delta(a) = \sqrt{\frac{\ln(1/a)}{2n}}$ . Тогда с вероятностью  $\geq 1 - a$  существует любое из:

- ◆ верхний интервал:  $\mu \leq \bar{x} + \Delta(a)$ ;
- ◆ нижний интервал:  $\mu \geq \bar{x} - \Delta(a)$ ;
- ◆ двусторонний интервал:  $\mu \in \bar{x} \pm \Delta(a/2)$ .

Двусторонний интервал составляется из двух других и границы объединения (вариант *неравенства Бонферрони*, которое будет рассмотрено далее в этой главе). Нижняя граница получается из верхней с помощью преобразования  $\mu = 1 - \mu$ .

Границы Хёффдинга не зависят от данных и могут рассматриваться в качестве критерия остановки. Они используются в основном теоретически из-за математической простоты, особенно в машинном обучении (см. главу 25. Основы машинного обучения). Границы конечной выборки — это лучшее, что можно применить на практике, но этого не следует делать без таких допущений, как ограниченность, потому что в противном случае очень редко будут встречаться большие значения вроде  $10^9$ .

Согласно численным исследованиям, приведенным в работе [21.4], *исходное неравенство Хёффдинга* дает наилучшую оценку. Верхняя граница  $\Delta$  неявно задается решением уравнения

$$\left( \left( \frac{1 - \bar{x} - \Delta}{1 - \bar{x}} \right)^{1 - \bar{x}} \left( \frac{\bar{x} + \Delta}{\bar{x}} \right)^{\bar{x}} \right) = a.$$

В работе не приведено деталей реализации, но можно отметить несколько хитростей:

- ◆ нижняя граница симметрична за счет выражения  $\mu = 1 - \mu$ ;
- ◆ с помощью логарифмов можно получить:

$$\log \left( 1 + \frac{\Delta}{1 - \bar{x}} \right) (1 - \bar{x}) + \log \left( 1 + \frac{\Delta}{\bar{x}} \right) \bar{x} = \frac{\log(a)}{n},$$

чтобы избавиться от дорогостоящего вычисления степени крайних случаев  $\bar{x} = 0$  или 1, а также улучшить вычислительную стабильность;

- ◆ можно найти решение уравнения для  $\Delta \in [0, 1 - \bar{x} - \varepsilon)$  или  $[0, \bar{x} - \varepsilon)$ , чтобы получить соответственно верхнюю или нижнюю границу, где  $\varepsilon$  — числовой предел точности. Если  $x \notin (\varepsilon, 1 - \varepsilon)$ , соответствующая оценка равна 0.

```
struct HoefFuncTor
{
    double ave, a;
    int n;
    double getTerm(double t, double d) const
    { // Предельное значение 0 достигнуто
        if (d < 0) assert(d <= t);
```

```

        double result = log(1 + d/t) * t;
        return isfinite(result) ? result : 0;
    }
public:
    double operator() (double d) const
    {
        assert(d >= 0 && d <= 1);
        return (getTerm(1 - ave, -d) + getTerm(ave, d)) * n - log(a);
    }
    static pair<double, double> conf(double ave, int n,
        double confidence = 0.95)
    {
        assert(ave >= 0 && ave <= 1 && confidence > 0 && confidence < 1 &&
            n > 0);
        HoefFunc f = {ave, (1 - confidence)/2, n};
        double e = numeric_limits<double>::epsilon();
        double upperD = ave < 1 - e ? solveFor0(f, 0,
            1 - ave - e).first : 0;
        f.ave = 1 - ave;
        double lowerD = ave > e ? solveFor0(f, 0,
            ave - e).first : 0;
        return make_pair(ave - lowerD, ave + upperD);
    }
};

```

## 21.10. Доверительные интервалы для общих метрик местоположения

*Метрикой местоположения* является любая линейная функция квантилей. В эту категорию попадают среднее, медиана и многие другие оценки.

Для среднего значения обычно используют интервал, основанный на  $t$ -распределении, где для небольших выборок вместо обычного применяют значение  $t$  с  $n - 1$  степенями свободы. Это точно для нормальных выборок и на практике наиболее полезно для малых  $n$  из ограниченного распределения:

```

pair<double, double> getTConf(IncrementalStatistics const& s, double a = 0.05)
{
    assert(s.n > 1 && a > 0 && a < 1);
    double ste = s.getStandardErrorSummary().stddev() *
        find2SidedConfT(1 - a, s.n - 1);
    return make_pair(s.getMean() - ste, s.getMean() + ste);
}
pair<double, double> getTConf(Vector<double> const& data, double a = 0.05)
{
    IncrementalStatistics s;
    for(int i = 0; i < data.getSize(); ++i) s.addValue(data[i]);
    return getTConf(s, a);
}

```

Медиана чаще всего представляет собой квантиль 50-го перцентиля. Логически  $k$ -й квантиль — это такая точка, что  $k$ -я часть данных не больше, а  $1 - k$  не меньше. У квантилей

выборки уникальные значения могут не достигаться. Таким образом, для медианы выборки с четным  $n$  соглашение состоит в том, чтобы усреднить два средних значения, хотя любое промежуточное значение тоже будет удовлетворять логическому свойству. В общем случае квантиль усредняет нижнее и верхнее значения, когда  $k$  попадает на разрыв EDF. По соглашению при  $k \notin [0, 1]$  результат равен  $\pm\infty$ . В частности, это помогает при работе с доверительными интервалами:

```
double quantile(Vector<double> data, double q, bool isSorted = false)
{
    assert(data.getSize() > 0);
    if(q < 0) return -numeric_limits<double>::infinity();
    else if(q > 1) return numeric_limits<double>::infinity();
    int n = data.getSize(), u = q * n, l = u - 1;
    if(u == n) u = l; // проверка крайних случаев
    else if(u == 0 || double(u) != q * n) l = u; // и граничных значений
    if(!isSorted)
    {
        quickSelect(data.getArray(), n, u);
        if(l != u) quickSelect(data.getArray(), u, l);
    }
    return (data[l] + data[u])/2;
}

double median(Vector<double> const& data, bool isSorted = false)
{return quantile(data, 0.5, isSorted);}
```

Чтобы получить непараметрические доверительные интервалы для квантильных оценок, нужно инвертировать *одновыборочный тест знаков*. Метод инверсии будет рассмотрен позже в этой главе. Идея теста знаков заключается в том, что для данных  $n$  выборок из распределения с известным  $k$ -м квантилем  $\theta$  ожидается, что  $kn$  из них  $< \theta$ . Таким образом, нулевое распределение является биномиальным  $(n, k)$  или приближенно нормальным  $(kn, nk(1 - k))$ . Затем, взяв  $z = 2$  (или другое разумное значение), нужно получить решение  $\theta \in (\text{квантиль}_{k-d}, \text{квантиль}_{k+d})$  с  $d = z \sqrt{\frac{k(1-k)}{n}}$  (в работе [21.28]

используется необязательная коррекция тонкости). Например, для набора данных  $\{1, 2, \dots, 99, 100\}$  медиана = 50,5, где  $(l, u) = (41, 60)$ . Для среднего нормальный интервал равен  $50,5 \pm 5,8$ . Это ожидаемо, потому что медиана менее эффективна для нормального распределения. Для усеченного среднего (обсуждается позже) получим оценку  $50,5 \pm 7,7$ :

```
pair<double, double> quantileConf(Vector<double> const& data, double q = 0.5,
    bool isSorted = false, double z = 2)
{
    double d = z * sqrt(q * (1 - q)/data.getSize());
    return make_pair(quantile(data, q - d, isSorted),
        quantile(data, q + d, isSorted));
}
```

Имейте в виду, что у дискретных распределений квантили могут быть неопределенными — например, для бинома с  $p = 0,5$  медиана не равна ни 0, ни 1. Доверительный интервал по-прежнему используется, хотя он будет равен  $[0, 1]$  даже в пределе.

Неверно думать, что в логике теста знаков используются только знаки данных и теряется слишком много информации. Точки данных являются случайными и по отдельности несут информацию только как часть выборки. Любая статистика получает лишь то, что ей нужно для расчета некоторого значения — например, среднего или медианы. Свойства  $S$  определяют полезность распределения.

Рассмотрим *усеченное среднее значение выборки* (см. [21.102]): извлечем  $\lfloor cn/2 \rfloor$  наименьших и наибольших значений и вернем среднее значение по остальным. Обычно  $c = 0,2$ . Имея массив данных, можно вычислить его за ожидаемое время  $O(n)$  с помощью двух вызовов функции `quickselect`, чтобы отделить середину от боков (возможен расчет в реальном времени, как и для медианы, но вместо двух куч используются три динамически отсортированные последовательности):

```
double trimmedMean(Vector<double> data, double c = 0.2,
    bool isSorted = false)
{
    int n = data.getSize(), trim = c * n;
    assert(n > 0 && c >= 0 && c < 0.5);
    if(!isSorted)
    {
        quickSelect(data.getArray(), n, n - trim - 1);
        quickSelect(data.getArray(), n - trim - 1, trim);
    }
    double sum = 0;
    for(int i = trim; i < n - trim; ++i) sum += data[i];
    return sum / (n - 2 * trim);
}
```

Значение  $c = 0$  эквивалентно среднему, а  $c = 0,5$  — медиане. Для распределения Коши усеченное среднее при оценке *параметра местоположения* работает эффективнее, чем медиана (см. [21.73]). Для симметричных распределений среднее, медиана и усеченное среднее совпадают. Асимптотически усеченное среднее значение выборки нормально (см. [21.102]), и его стандартное отклонение можно оценить аналитически. *Виндсоризация* выборки аналогична обрезке, за исключением того, что вместо удаления хвостов они заменяются копиями оставшихся экстремальных значений. Тогда

$$SE(\text{выборочное усеченное среднее}) = \frac{SE(\text{виндсоризованная выборка})}{1 - 2c}.$$

```
double trimmedMeanStandardError(Vector<double> data, double c = 0.2,
    bool isSorted = false)
{
    int n = data.getSize(), trim = c * n;
    assert(n > 0 && c >= 0 && c < 0.5);
    if(!isSorted)
    {
        quickSelect(data.getArray(), n, n - trim - 1);
        quickSelect(data.getArray(), n - trim - 1, trim);
    } // Виндсоризация хвостов
    for(int i = 0; i < trim; ++i) data[i] = data[trim];
    for(int i = n - trim; i < n; ++i) data[i] = data[n - trim - 1];
    IncrementalStatistics s; // вычисление отклонений значений
```

```

for(int i = 0; i < n; ++i) s.addValue(data[i]);
return s.getStandardErrorSummary().stddev()/(1 - 2 * c);
}
pair<double, double> trimmedMeanConf(Vector<double> const& data, double z = 2)
{
    double tm = trimmedMean(data), ste = trimmedMeanStandardError(data) * z;
    return make_pair(tm - ste, tm + ste);
}

```

Эта оценка на практике хорошо работает при  $c = 0,2$ , но менее точна для гораздо меньших или гораздо больших значений, особенно при расчете медианы (см. [21.102]).

Обрезка помогает повысить эффективность, поскольку точность выборочного среднего зависит от усреднения, а обрезка отбрасывает наблюдения, которые снижают точность из-за того, что они слишком велики или малы, — т. е. точность выборочного среднего чувствительна к распределениям с толстым хвостом.

Для медианы выборки относительно среднего значения выборки *асимптотическая относительная эффективность* (ARE) составляет (см. [21.28]):

- ◆  $\infty$  — для распределения Коши;
- ◆  $1,5/\pi$  — для нормального распределения;
- ◆  $1/3$  — для равномерного распределения (наихудший случай).

Среднее значение выборки всегда эффективно для ограниченных распределений, что следует из неравенств конечной выборки. У непрерывного неизвестного распределения безопаснее использовать усеченное среднее значение выборки или медиану выборки.

Для асимметричных распределений среднее  $\neq$  усеченное среднее  $\neq$  медиана, поэтому необходимо решить, какую метрику местоположения использовать. Например, для цен на жилье предпочтение отдается медиане, но решение это принимается исходя из знаний предметной области, а не из каких-либо конкретных свойств оценщика. Даже при одних и тех же данных могут потребоваться разные параметры, если данные используются для разных целей. Например, мы можем получить данные о доходах, средний доход для социального обеспечения и средний доход для оценки налогов. Кроме того, когда экспоненциально большие значения экспоненциально маловероятны, среднее значение может быть большим, а медиана — малой. Здесь большое среднее значение не имеет значения, потому что большие числа фактически невозможны. Для данных о зарплатах среднее значение информативно для работодателей, но медиана лучше для сотрудников, потому что мало кто становится генеральным директором. Для задачи Бернулли с  $p \neq 0,5$  имеет смысл только среднее: медиана равна 0 или 1, а усеченное среднее смещено по отношению к среднему. Наконец, многие распределения не имеют хороших показателей местоположения в том смысле, что ни одно число не отражает нормально типичное поведение. Усеченное среднее и другие местоположения, которые соответствуют медиане в симметричных распределениях, называются *псевдомедианами*.

## 21.11. Выпадающие значения и надежные выводы

Почти все статистические процедуры предполагают наличие независимых случайных данных. Но практические данные могут содержать наблюдения, не соответствующие

этому условию, т. к. всегда бывают ошибки в представлении данных или измерениях. Такие данные являются *выпадающими*. Чтобы отличить выпадающее значение от правильного в распределении с толстым хвостом, нужна дополнительная информация, например:

- ◆ знание предметной области. Невозможные значения являются выпадающими — например, возраст не может быть отрицательным;
- ◆ сильное априорное убеждение в существующей модели. Например, если данные являются нормальными с известными параметрами, значения, отличающиеся намного от стандартных отклонений, экспоненциально маловероятны. Потенциально это выпадающие значения. Но модель может быть плохой аппроксимацией, что чаще всего влияет на хвосты. Таким образом, для идентификации выпадающих значений нужно использовать только проверенные хорошие модели.

Выпадающие значения могут быть интересны сами по себе — например, они могут интерпретироваться как аномалии, которые стоит учитывать (подробнее об этом сказано в главе 29. *Машинное обучение: другие задачи*), но мы не будем рассматривать такие случаи. Основные способы снижения влияния выпадающих значений на оценку:

- ◆ выявить и удалить предполагаемые выпадающие значения перед применением процедур оценки. Иногда считается, что задача поставлена некорректно и не существует хорошей конкретной общей стратегии или способов измерения производительности;
- ◆ использовать *надежных оценщиков*, влияние выпадающих значений на которые ограничено. Это предпочтительная стратегия, и оценки, которые эффективны для распределений с толстым хвостом, обычно изначально являются надежными. Например, для оценки среднего значения симметричного распределения усеченное среднее и средние выборки достаточно надежны. Если вы хотите усреднить возраст в модели нормального распределения, значение «-1» будет иметь сильный эффект смещения в небольшой выборке. Но при использовании усеченного среднего или медианы влияние на оценку незначительно.

В этой логике стоит остерегаться только выпадающих значений с очень большими/маленькими значениями, потому что значения, близкие к другим данным, не могут иметь большого влияния. Говоря более формально, мы рассматриваем величину *влияния* — т. е. какое влияние может оказать одно наблюдение. Например, среднее выборки имеет влияние  $= \infty$ , если имеется наблюдение  $= \infty$ . Но у медианы влияние конечно. Медиана выборки остается конечной даже при бесконечных значениях, отсекая 50% данных. Таким образом, ее *точка пробоя*  $= 50\%$ . У усеченного среднего точка пробоя равна  $c$ . Разбивка имеет для небольших выборок большее значение, чем для больших, — например, даже при 10%-ном загрязнении данных часть образцов при  $n = 20$  будет непригодна для использования даже при умеренно надежной оценке (см. [21.41]). Однажды я был в составе присяжных, которым нужно было согласовать сумму возмещения ущерба, не зная ничего о достоверной статистике. Поэтому, естественно, я предложил использовать среднее значение, и все согласились. Один присяжный дал 0 долларов, а другой — 10 миллионов долларов, т. е. выпадающее значение, и была присуждена более высокая сумма, чем если бы вместо этого использовалась медиана (разница не менее 1 миллиона долларов).

Увеличение надежности может повредить стабильности. Например, для бинома с  $p = 0,5$  выборочная медиана очень надежна, но дает значение  $p$ , равное 0 или 1, независимо от



того, насколько велико  $n$ . Нечто подобное может произойти и при использовании усеченного среднего значения выборки.

Если забыть о выпадающих значениях, еще одним преимуществом надежных оценок является меньшая *чувствительность к спецификации модели* — например, когда данные имеют некоторое распределение, близкое к нормальному, но не совсем нормальное. Даже при небольших отклонениях от предположений о распределении надежная оценка может быть гораздо более достоверной и/или эффективной. В случае распределения с толстым хвостом это наверняка будет справедливо. В работе [21.102] приведено много примеров. Но обычно более сложные вычисления при использовании надежных оценок неудобны. Например, иногда не удастся выполнять инкрементальные вычисления, что важно для больших наборов данных.

Например, стандартное отклонение выборки неэффективно для распределений с толстым хвостом. Надежной альтернативой является *нормализованное среднее абсолютное отклонение* (Normalized Median Absolute Deviation, MADN). Примените медиану абсолютных разностей к медиане и выполните нормализацию, чтобы сделать значение равным стандартному отклонению нормального распределения (см. [21.102]):

```
pair<double, double> medianMADN(Vector<double> data)
{ // масштабирование для соответствия нормальному стандартному отклонению
  assert(data.getSize() > 2);
  double med = median(data);
  for(int i = 0; i < data.getSize(); ++i) data[i] = abs(data[i] - med);
  return make_pair(med, 1.4826 * median(data));
}
```

Для нормальных данных эффективность составляет всего 37 %, а точка пробоя — 50 %, а для других распределений эффективность может быть намного выше.

## 21.12. Функции оценок

Чтобы сделать вывод о  $g(\theta)$  для некоторого  $g$ , обычно лучше всего выполнить оценку напрямую. Но иногда это проблематично.

Для доверительных интервалов полезным инструментом является *дельта-метод*

(см. [21.79]): асимптотически, если  $\hat{\theta} \sim \text{normal}\left(\theta, \frac{\sigma^2}{n}\right)$ ,  $g'(\theta)$  существует и  $\neq 0$ ,

$$g(\hat{\theta}) \sim \text{normal}\left(g(\theta), \frac{(|g'(\theta)|\sigma)^2}{n}\right).$$

Поэтому для оценки нужно умножить  $s$  на  $|g'(\theta)|$ . Это также работает для  $D > 1$  с использованием  $\mathbb{V}g$  и ковариационной матрицы.

Если  $g'(\theta) = 0$ ,  $g''(\theta)$  определяет асимптотическое поведение, а предельное распределение — хи-квадрат (см. [21.20]). В более общем случае используется разложение  $g$  в ряд Тейлора, насколько это необходимо. Это свойство является асимптотическим, поэтому для проверки конкретных случаев необходимо моделирование, особенно с учетом того, что для  $g$ , такого как возведение в степень, распределение конечной выборки явно не является нормальным.

В остальных случаях нужно выполнить расчет вручную. Например, рассмотрим  $g(x) = \begin{cases} x, & \text{if } x < 0 \\ 2x, & \text{else} \end{cases}$  и  $f$  = среднее значение выборки. При  $x = 0$  функция  $g$  недифференцируема, поэтому нельзя использовать этот метод напрямую. Можно рассмотреть распределения близких значений  $\theta$ , чтобы сделать вывод о поведении  $S$  при преобразовании с помощью  $g$ . Если истинное среднее значение равно 0,  $g(\bar{x})$  — это распределение, в котором левая половина является нормальной, а правая половина — сжатая нормальная (результат кажется, но не является 50-процентной смесью нормального  $(0, 1)$  и нормального  $(0, 2)$ ).

Для  $g(x) = |x|$  даже моделирование  $S$  не может дать доверительный интервал, который будет содержать  $\theta = 0$  в 95% случаев, потому что оно находится на границе.

Иногда требуется *инвариантность преобразования*, т. е. преобразование данных эквивалентно преобразованию доверительных интервалов или других оценок таким же образом. Но она либо есть у процедуры, либо нет, и выбирать процедуру по этому свойству мало смысла.

## 21.13. Измерение времени выполнения алгоритма

Обычно алгоритм запускают несколько раз, выполняют некоторые измерения и вычисляют требуемую статистику. Зачастую это работает достаточно хорошо, но нужно быть осторожным. Уточним некоторые источники шума:

- ◆ рандомизация в алгоритме;
- ◆ смещение часов (дисперсия здесь не так важна);
- ◆ дисперсия нагрузки на систему.

Шум всегда только увеличивает время выполнения, поэтому распределение асимметрично. Даже если часы работают точно, нам нужно узнать  $E[\text{время выполнения алгоритма}]$ , но вместо этого замеряется время работы самого алгоритма + время работы системы. Система с небольшой вероятностью может вносить существенные задержки — т. е. время работы системы потенциально имеет распределение с толстым хвостом.

Простая процедура получения более качественных измерений заключается в запуске нескольких пакетов — чтобы свести на нет шум от часов процессора и рандомизации в алгоритме. Используйте в отношении времени выполнения пакета усеченное среднее выборки. Разделите его на количество запусков в пакете, чтобы оценить типичное время выполнения. Это позволяет узнать время выполнения алгоритма + типичное время выполнения системы, избегая экстремальных, менее вероятных случаев серьезных системных задержек. Будем надеяться, что различия между системами окажутся пропорциональны их скорости, а относительные различия в производительности будут связаны только с различиями алгоритмов.

Квантиль 90% может в некоторых случаях быть лучшим показателем местоположения, поэтому управление им для большинства пользователей достаточно в полунаихудшем случае. Но если это имеет значение, нужно провести сравнение на предмет *стохастичности*.

ческого доминирования — т. е. когда одна альтернатива превосходит остальные по всем квантилям распределения, а не только по какому-то конкретному показателю, такому как медиана. Для однопараметрических распределений разность средних обычно подразумевает стохастическое преобладание — например, даже для нормального распределения большая дисперсия добавляет больше массы хвостам, так что сравнение среднего и 90%-ного квантиля может дать разностные результаты.

## 21.14. Корреляционный анализ

Требуется обнаружить взаимоотношения между данными. Корреляция Пирсона обнаруживает линейные отношения с использованием коэффициента корреляции:

$$r = \frac{1}{n-1} \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{s_x s_y},$$

чтобы оценить коэффициент линейной корреляции  $\rho$ . Ответ лежит в диапазоне  $[-1, 1]$ , где 1 означает совершенную линейную зависимость, а  $-1$  — совершенную отрицательно-линейную зависимость. Величины вне этих значений означают только отсутствие линейной зависимости — например, для выборок из  $y = x^2$ ,  $\rho = 0$ .  $\rho \geq 0,95$  считается сильным:

```
double PearsonCorrelation(Vector<pair<double, double> > const& a)
{
    int n = a.getSize();
    assert(n > 1);
    IncrementalStatistics x, y;
    for(int i = 0; i < n; ++i)
    {
        x.addValue(a[i].first);
        y.addValue(a[i].second);
    }
    double covSum = 0;
    for(int i = 0; i < a.getSize(); ++i)
        covSum += (a[i].first - x.getMean()) * (a[i].second - y.getMean());
    covSum /= n - 1;
    double result = covSum/sqrt(x.getVariance() * y.getVariance());
    return isfinite(result) ? result : 0; // проверка деления на 0
}
```

Вычисление доверительного интервала для  $\rho$  аналитически требует допущений. Достаточно точный асимптотический нормальный интервал работает хорошо, если сначала сделать преобразование. В частности (см. [21.98]), для данных от двумерного нормального распределения с корреляцией  $\rho$  асимптотически:

$$\tanh^{-1}(r) \sim \text{normal}\left(\tanh^{-1}(\rho), \frac{1}{n-3}\right).$$

```
pair<double, double> PearsonCorrelationConf(double corr, int n, double z = 2)
{
    assert(0 <= corr && corr <= 1 && n > 3);
    double stat = atanh(corr), std = 1/sqrt(n - 3);
```

```

    return make_pair(tanh(stat - z * std), tanh(stat + z * std));
}

```

Корректировка на 3 исходит не из математики, но известно, что числовое значение работает лучше (см. [21.52], примеры расчета также приведены в работе [21.24]). Преобразование асимптотически верно только для двумерного нормального распределения (см. [21.102]), но это не помешало его широкому распространению.

Поскольку переменные могут быть связаны монотонно, но не линейно, *корреляция Спирмена* сначала преобразует их в ранги, а затем вычисляет *корреляцию Пирсона*. Вместо этого можно непосредственно вычислить среднее значение ранга и дисперсию, но это усложняется при наличии связей:

```

Vector<double> convertToRanks(Vector<double> a)
{ // создать массив индексов, отсортировать и его и превратить индексы в ранги
    int n = a.getSize();
    Vector<int> indices(n);
    for(int i = 0; i < n; ++i) indices[i] = i;
    IndexComparator<double> c(a.getArray());
    quickSort(indices.getArray(), 0, n - 1, c);
    for(int i = 0; i < n; ++i)
    { // ранжирование при просмотре, поиск совпадений и изменение записей
        int j = i;
        while(i + 1 < n && c.isEqual(indices[j], indices[i + 1])) ++i;
        double rank = (i + j)/2.0 + 1;
        for(; j <= i; ++j) a[indices[j]] = rank;
    }
    return a;
}

double SpearmanCorrelation(Vector<pair<double, double> > a)
{
    Vector<double> x, y;
    for(int i = 0; i < a.getSize(); ++i)
    {
        x.append(a[i].first);
        y.append(a[i].second);
    }
    x = convertToRanks(x), y = convertToRanks(x);
    for(int i = 0; i < a.getSize(); ++i)
    {
        a[i].first = x[i];
        a[i].second = y[i];
    }
    return PearsonCorrelation(a);
}

```

Даже при наличии связей асимптотически  $r \sim normal\left(p, \frac{1}{n-1}\right)$ , что считается точным при  $n \geq 30$  (см. [21.28]):

```

pair<double, double> SpearmanCorrelationConf(double corr, int n, double z = 2)
{
    assert(0 <= corr && corr <= 1 && n > 1);
    double std = 1/sqrt(n - 1);

```

```
return make_pair(corr - z * std, corr + z * std);
}
```

Корреляционный анализ редко полезен, если это не исследовательский инструмент. Любая корреляционная мера должна иметь четкое представление о том, что она измеряет. В общем случае ни одно число не дает хорошего показателя степени зависимости. Обычно требуется иметь прогностическую модель, которую лучше всего составлять с помощью регрессии. В финансах, например, исторические корреляции между ценными бумагами очень полезны для расчета рисков портфелей, но здесь на общую зависимость не обращают внимания. Корреляции Пирсона и Спирмена ненадежны — например, даже небольшое вращение данных существенно меняет корреляцию Пирсона (см. [21.102]). В работе [21.78] приведена дополнительная информация о корреляции. Любая интерпретация высокой корреляции не должна быть причинно-следственной — например, потребление мороженого и уровень убийств коррелируют, но лишь потому, что и то и другое происходит чаще летом.

Также следует обратить внимание, что доверительные интервалы в обоих случаях не зависят от данных, поэтому даже при  $n = 100$  будет получен 95%-ный доверительный интервал при  $r \pm 0,2$ , что встречается нередко.

Альтернативой корреляции Спирмена является *корреляция Кендалла* (см. [21.101]), которая для каждого наблюдения подсчитывает количество согласованных пар, где для каждого другого наблюдения  $x$ , и  $y$  меньше или больше, чем эталонное наблюдение. Результатом является общее количество согласованных пар, деленное на выбор ( $n, 2$ ). Некоторые отличия:

- ◆ вычисление на самом деле выполняется за  $O(n \lg(n))$  с использованием сортировки;
- ◆ результат более надежен, чем у Спирмена;
- ◆ нормальное приближение для полученного биннома лучше.

Обычно формулы корректируются исходя из того, что среднее значение известно. Например, для данных временных рядов, таких как дивиденды от акций, общая модель состоит в том, чтобы различать данные и смотреть на ежедневные изменения, у которых известно среднее значение, равное 0, или некоторое предполагаемое среднее значение долгосрочного тренда. Формулу Пирсона легко подстроить — нужно подставить известные средние значения и удалить некоторую степень свободы из нормальной достоверности для каждого известного среднего значения.

В случае со Спирменом все сложнее — нужно использовать ранг известного среднего, но исключить этот ранг из расчета. Распределение также меняется, и необходимо изучить общий способ настройки нормального приближения, если таковой необходим. У Кендалла простая эвристика, которая менее точна для других методов, состоит в том, чтобы включить известное среднее значение в качестве наблюдения  $i$ , возможно, дополнительно скорректировать расчет, чтобы не использовать его с другими точками данных в качестве эталонных наблюдений.

## 21.15. Начальная загрузка

Вы можете оценить точность оценщика-«черного ящика»  $f$ , сделав предположения путем повторной выборки набора данных. EDF  $F(x)$  аппроксимирует реальное распре-

ление  $T$ . В одном измерении получается *неравенство Дворецкого — Кифера — Вольфовица* (DKW):

$$\Pr(\max_x |(T(x) - F(x))| > \varepsilon) \leq 2e^{-2n\varepsilon^2}$$

для ошибки  $\varepsilon > 0$ , и такое, что это значение  $p \leq 0,5$  (см. [21.20]). Для векторного значения  $f$  нет многомерного DKW — только асимптотическая сходимость  $F$  к  $T$ , но этого достаточно для начальной загрузки.

Начальная загрузка (bootstrap) создает выборки  $f_i = f(\text{resample}_i)$ , где  $\text{resample}_i$  состоит из  $n$  выборок из  $F$  из распределения  $B$ .  $B \approx S$ , когда  $F \approx T$  (если только функция  $f$  не проблематичная). Поэтому  $f_i$  можно считать независимыми случайными данными из  $S$  и использовать их, чтобы найти доверительный интервал для  $\theta$ :

1.  $b$  раз:
2. Выбрать  $n$  случайных элементов из данных с помощью замены:
3.  $f_i$  заменяется на  $f(\text{resample})$ .
4. Расчет доверительного интервала для  $\hat{\theta}$  из  $b$  значений  $f_i$ , используя один из описанных далее методов.

Если  $f$  такова, что увеличение  $n$  не увеличивает ошибку выборки из  $F$  вместо  $T$ , эвристически значение  $B \approx S$  становится сколь угодно точным при  $n \rightarrow \infty$ . В общем случае следует ожидать, что начальная загрузка будет работать при выполнении моделирования. Начальная загрузка не сработает, если моделирование:

- ♦ пусть  $f = g$  (среднее выборки), где  $g$  — ступенчатая функция, а истинное среднее = 0. Тогда  $g(\text{среднее выборки}) \sim \text{Бернулли}(0,5)$  независимо от  $n$ , и ни один доверительный интервал не уменьшается с увеличением  $n$ ;
- ♦ если дисперсия  $(S) = \infty$ , например, при использовании среднего значения выборок Коши,  $B \approx S$ , но доверительные интервалы на основе дисперсии недействительны (интервалы на основе квантилей действительны, но бесполезны).

Как будет показано далее, ни один метод не позволяет надежно определить, когда начальная загрузка дает неверные доверительные интервалы. Что касается максимальной вероятности, то нужны условия регулярности. Как и с максимальной вероятностью, на практике эти значения проверяются редко.

Все методы доверительного интервала работают с ресемплером. Для одиночной выборки  $f$  используйте следующее:

```
template<typename FUNCTION, typename DATA = double> struct BasicBooter
{
    Vector<DATA> const& data;
    Vector<DATA> resample;
    FUNCTION f;
    BasicBooter(Vector<DATA> const& theData, FUNCTION const& theF = FUNCTION())
        :data(theData), f(theF), resample(theData){assert(data.getSize() > 0);}
    void boot()
    {
        for(int i = 0; i < data.getSize(); ++i)
            resample[i] = data[GlobalRNG().mod(data.getSize())];
    }
}
```

```
double eval()const{return f(resample);}
// для интервала начальной загрузки-t
BasicBooter cloneForNested()const{return BasicBooter(resample, f);}
};
```

Предположим, что распределение  $S$  является нормальным. Тогда вычисляется стандартное отклонение от  $B$ , чтобы сформировать *нормальный доверительный интервал начальной загрузки*. Достаточно выполнить 25 операций ресемплинга, а для большей точности рекомендуется выполнить 200 (см. [21.24]):

```
template<typename BOOTER> double bootstrapStde(BOOTER& booter, int b = 200)
{
    assert(b > 2);
    IncrementalStatistics s;
    for(int i = 0; i < b; ++i)
    {
        booter.boot(); // ресемплинг
        s.addValue(booter.eval());
    } // внимание — здесь вы получите стандартное отклонение, а не стандартную ошибку
    return s.stdev();
}

template<typename BOOTER> pair<double, double> bootstrapNormalInterval(
    BOOTER& booter, int b = 200, double z = 2)
{
    double q = booter.eval(), stde = bootstrapStde(booter, b);
    return make_pair(q - z * stde, q + z * stde);
}
```

Например, в конкретном прогоне выборки из 1000 равномерно распределенных, где  $f$  = среднее значение выборки, нормальный интервал начальной загрузки дает  $\hat{\theta} = 0,493 \pm 0,019$  с достоверностью 95%. Если  $f$  = медиана, непараметрический интервал дает  $\hat{\theta} = 0,512 \begin{smallmatrix} +0,035 \\ -0,029 \end{smallmatrix}$ , а нормальный интервал начальной загрузки дает  $0,512 \pm 0,032$ .

По сути, мы вычисляем интервал Вальда, но без необходимости математических выводов для стандартных ошибок неизвестных значений  $f$ . У этого метода много преимуществ по сравнению с более точными интервалами, обсуждаемыми позже:

- ◆ точность обычно достаточно высока — для малых  $n$  ошибка оценки предположения о нормальности мала, а для больших это хорошее приближение, потому что большинство оценщиков вычисляют *концентрации*, подобные среднему значению или медиане;
- ◆ вычисление оказывается на несколько порядков эффективнее, чем вычисление более точных интервалов. Для большого  $n$  и дорогого  $f$  это вычислительное решение может оказаться единственно возможным;
- ◆ теоретически более точные интервалы являются более точными только для гладких  $f$ , тогда как начальная загрузка более полезна для определяемого пользователем «черного ящика»  $f$ , который может вообще не быть гладким;
- ◆ метод легко объяснить, сочетание оценки и ошибки четко интерпретируется;
- ◆ легко вносить поправки для выполнения многократного тестирования.

Тем не менее предположение о нормальности оказывается несколько неточно для асимметричных  $S$  — например, для среднего логарифмически нормального распределения. Асимптотически интервал, основанный на неравенстве Чебышева с использованием  $z = \sqrt{1/a}$ , согласован для выборочного среднего для любого  $S$  с конечной дисперсией, поскольку  $s \rightarrow \sigma$ . Но известно, что он консервативен, поэтому следует использовать смешанный интервал Чебышева с  $z = cz_{normal} + (1-c)\sqrt{1/a}$ . Это небольшое отклонение от нормального интервала компенсирует несколько проблем:

- ◆ в общем случае, например, при наличии нескольких выборок, значение  $n$  неизвестно, поэтому не удастся напрямую использовать  $t$ -распределение;
- ◆ нормальность — это лишь приближительная модель, особенно для малых  $n$  — интервал Вальда не обязательно должен быть даже асимптотически правильным, поскольку  $f$  может не иметь CLT;
- ◆ начальная загрузка слегка занижает  $s$  (см. [21.24]), используя оценку максимального правдоподобия;
- ◆ истинный интервал, вероятно, будет асимметричным;
- ◆ функция  $f$  может быть слегка смещенной.

Метод эффективен даже при небольшом смешивании при  $c = 0,9$ . Например, для стандартной достоверности 95% получаем  $z \approx 2,21$ , что соответствует  $t$ -распределению с 10 степенями свободы. Для достоверности 99,73%, соответствующих  $z = 3$ , для  $c = 0,9$  получаем  $z \approx 4,62$ . При  $c = 0,8$  получаем  $z \approx 2,46$ , что соответствует  $t$ -распределению с 6 степенями свободы, а для 99,73% получаем  $z \approx 6,25$ :

```
double getMixedZ(double a = 0.05, double c = 0.9)
{
    assert(a > 0 && a < 1 && b > 0 && b < 1);
    double z = find2SidedConfZ(1 - a), chebZ = sqrt(1/a),
    return z * c + chebZ * (1 - c);
}

template<typename BOOTER> pair<double, double> bootstrapMixedInterval(
    BOOTER& booter, int b = 200, double a = 0.05)
{ // используется смешивание на 10 процентов
    assert(b > 2 && a > 0 && a < 1);
    return bootstrapNormalInterval(booter, b, getMixedZ(a));
}
```

Асимптотически  $f(F) \rightarrow f(T) = \theta$ , предполагая, что  $f$  непротиворечиво. Также  $f(F) \approx \hat{\theta}$  для разумных значений  $f$ , поскольку использование набора данных, состоящего из нескольких копий исходных данных, не должно иметь значения, если  $f$  каким-либо образом не учитывает  $n$ . Технически  $f$  может дублировать данные для собственных целей столько раз, сколько необходимо, но это не проблема, если в итоге достигается какой-то полезный результат. Это единственное отклонение от идеи выполнения всех вычислений в «мире начальной загрузки»  $F \approx T$ , связанное с существенной экономией вычислительных ресурсов. В противном случае пришлось бы моделировать большое количество выборок для вычисления  $f(F)$ . Напомним, что теоретически идеальный интервал для  $\theta$  равен  $(l, u) = (\hat{\theta} + \theta - S_{1-a}, \hat{\theta} + \theta - S_a)$ . Подставьте  $B$  вместо  $S$  и  $f(F)$  вместо  $\theta$ . Предполагая, что  $f(F) \approx \hat{\theta}$  (что на самом деле точно, если  $f$  является подключаемой оцен-



кой — см. [21.54]), получаем  $(l, u) = (2\hat{\theta} + B_{1-a}, 2\hat{\theta} - B_a)$ . Этот интервал называется *опорным* (согласно терминологии [21.79] или *основным, перевернутым и отраженным* в других источниках) и фактически предполагает только  $f(T, n) - \theta \approx f(B, n) - \hat{\theta}$ .

Хотя значение  $b = 1000$  считается достаточным для таких интервалов, основанных на хвостовом квантиле (см. [21.24]), с современной вычислительной мощностью значение 10 000 кажется лучше (см. [21.18]), а по моим экспериментам получается, что 3000 — хорошая золотая середина (обсуждается позже). Меньших чисел, таких как 200, для нормального интервала уже недостаточно, потому что стандартное отклонение оценить легче, чем хвостовые квантили. Например, при  $b = 100$  бессмысленно запрашивать 99,5%-ный интервал на основе квантилей. Количество различных повторных выборок равно  $\binom{2n-1}{n}$  и превосходит любой возможный на практике выбор  $b$ . Таким образом, при некотором достаточно большом  $b$  все различные значения будут использованы примерно в равной пропорции, и увеличение  $b$  не улучшит ни точность, ни разрешение. Увеличение  $b$  до высоких значений ничего не дает (несмотря на важность для теории), а  $n$  ограничивает точность.

Хотя базовый интервал является согласованным (с условиями регулярности, как и для других интервалов), не стоит ожидать, что  $\hat{\theta} \approx \theta$ , потому что последнее фиксировано, а первое случайно. Но различия должны быть такими, что  $\hat{\theta}$  отклоняется от  $\theta$  в том же направлении, что и квантили  $B^*$  от  $S^*$ . Кроме того, практическая эффективность этого интервала плоха из-за перекоса  $S$ . В работе [21.35]) это называется *неправильным разворотом вперед* и подразумевает движение в неправильном направлении, чтобы исправить перекося из-за чрезмерного влияния  $\hat{\theta}$  в расчетах.

Лучшие опорные точки получаются из стандартного отклонения, что дает *интервал начальной загрузки- $t$* . Название связано с тем, что сводная точка выглядит как статистика  $t$ , но в остальном не имеет к ней никакого отношения. Опорный интервал можно представить как  $(l, u) = (\hat{\theta} - (B - \hat{\theta})_{1-a}, \hat{\theta} - (B - \hat{\theta})_a)$ , где вычисленные значения  $B$  сразу же корректируются с помощью  $\theta$ . Чтобы улучшить его, используйте зависимость  $(l, u) = (\hat{\theta} - s(B)P_{1-a}, \hat{\theta} - s(B)P_a)$ , где  $P$  — это распределение  $p(\text{ресемпл}) = \frac{f(\text{ресемпл}) - \hat{\theta}}{\text{std}(f(\text{ресемпл}))}$ .

Вычислять стандартные отклонения для повторной выборки аналитически можно только в особых случаях, например, когда  $f$  — среднее значение (но этот метод тем не менее полезен — в работе [21.102] приведены примеры). В общем случае начальная загрузка применяется рекурсивно — т. е. вычисляются стандартные отклонения для ресемплов, используя вложенную начальную загрузку, которая задействует ресемплы в качестве данных. Получается еще один уровень начальной загрузки. Этот метод требует больших вычислительных ресурсов, поэтому для вложенных загрузок используйте параметры  $b = 1000$  и  $b_2 = 50$ . Чтобы отслеживать квантили, возьмите две кучи размером  $t \approx ba$  для хранения только хвостов с экстремальными значениями. Затраты ресурсов составляют  $O(fbb_2 \ln(t))$  времени и  $O(n + t)$  пространства.

В результате производительность значительно увеличивается, как теоретически, так и практически. Интервал получается согласованным, как и опорный, и использование

стандартных отклонений для повторной выборки имеет важное значение, что подтверждается моделированием. Теоретически (см. [21.24]) начальная загрузка- $t$  имеет второй порядок точности для гладких  $f$  — т. е. одностороннее покрытие =  $a/2 + o(n^{-1})$ . Нормальный и опорный интервалы имеют точность только первого порядка — т. е. одностороннее покрытие =  $a/2 + o(n^{-1/2})$ . Они относятся к ошибке асимптотической аппроксимации и для нормального интервала неявно предполагают дополнительные условия, такие как наличие CLT для  $f$ .

К сожалению, известно, что начальная загрузка- $t$  нестабильна при малых  $n$ , поэтому в работе [21.24] рекомендовано не использовать ее, за исключением случаев, когда действуют метрики местоположения — например, среднее значение или квантили. В ходе экспериментов (о них поговорим позже) я обнаружил две нестабильности:

- ◆ по длине — например, для корреляции Пирсона при  $n = 5$ ;
- ◆ по охвату — например, для корреляции Спирмена и нормального стандартного отклонения при  $n = 5$  и  $10$ .

Простое решение состоит в том, чтобы использовать консервативное (для среднего значения выборки) неравенство Чебышева, ограничивающее вычисляемые опорные точки диапазоном  $[1, z \text{ Чебышева}]$ . Здесь  $1$  — это значение  $z$ , соответствующее максимальному  $a = 1$ , а интервалы стандартной ошибки по-прежнему  $\leq 1$ . Это позволяет значительно увеличить производительность в проблемных случаях и делает метод в целом безопасным. Неясно, существует ли на этот счет лучшая общая стратегия ограничения:

```
template<typename BOOTER> pair<double, double> bootstrapTIntervalCapped(
    BOOTER& booter, int b = 1000, int b2 = 50, double confidence = 0.95)
{
    assert(b > 2);
    double a = (1 - confidence)/2;
    int tailSize = b * a;
    if(tailSize < 1) tailSize = 1;
    if(tailSize > b/2 - 1) tailSize = b/2 - 1;
    // максимальная куча для работы с наибольшими значениями
    Heap<double, ReverseComparator<double> > left;
    Heap<double> right; // минимальная куча для работы с наименьшими значениями
    double q = booter.eval();
    IncrementalStatistics s;
    for(int i = 0; i < b; ++i)
    {
        booter.boot(); // ресемплинг
        double valStat = booter.eval();
        if(isfinite(valStat)) s.addValue(valStat);
        else continue;
        BOOTER booter2 = booter.cloneForNested();
        double stdeInner = bootstrapStde(booter2, b2),
            value = (valStat - q)/stdeInner;
        if(isnan(value)) continue; // возможно деление на 0, но это нормально
        if(left.getSize() < tailSize)
        { // кучи не заполнены
            left.insert(value);
            right.insert(value);
        }
    }
}
```

```

else
{ // кучи заполнены – выполнить замену при необходимости
  if (value < left.getMin()) left.changeKey(0, value);
  else if (value > right.getMin()) right.changeKey(0, value);
}
} // осторожно – здесь вы получите стандартное отклонение, а не стандартную ошибку
// защита от плохих данных
double normalZ = 1, chebZ = sqrt(1/(2 * a)), stde = s.n > 2 ? s.stdev() :
  numeric_limits<double>::infinity(), leftZ = right.getSize() > 0 ?
  min(max(right.getMin(), normalZ), chebZ) : chebZ, rightZ =
  left.getSize() > 0 ? min(max(-left.getMin(), normalZ), chebZ) : chebZ;
return make_pair(q - stde * leftZ, q + stde * rightZ);
}

```

Еще один способ получить интервал второго порядка — использовать *калибровку начальной загрузки* (см. [21.9]), которая работает с любым интервалом:

1. Вместо того чтобы вычислять интервал из исходных  $b$  повторных выборок, создайте  $b_2$  повторных выборок  $B$  и вычислите интервал для каждой из них с желаемым значением  $a$ .
2. Часть интервалов  $b_2$  будет находиться справа, а часть — слева от  $\theta$  и не будет содержать его. Поэтому повторите пункт (1) со скорректированной боковой целью  $a$ , чтобы получить нужные данные с помощью числового двоичного поиска.
3. Вычислите первичный интервал из исходных повторных выборок  $b$ , но со скорректированным значением  $a$ .

Некоторые детали реализации, зависящие от используемого интервала:

- ◆ диапазон для бинарного поиска, который сводится к решению уравнения для 0. Может использовать экспоненциальный поиск, но в интервалах, обсуждаемых далее, границы заданы хорошо;
- ◆ что делать, если нет решения в указанном ранее диапазоне (конечные точки не имеют разных знаков)? Нужно взять верхнюю границу равной  $a$ , если ее достаточно, и нижнюю границу в противном случае.

Простой метод, очень похожий, по сути, на  $t$ -интервал начальной загрузки, — это *калиброванный интервал Чебышева*. В нем используется *калибровка с интервалом в виде неравенства Чебышева*. В отличие от обычного интервала, он дает диапазон автоматического поиска, равный  $[a_{target}, 1]$ , который является границами ограниченной начальной загрузки- $t$ . По умолчанию по-прежнему используются значения  $b = 1000$  и  $b_2 = 50$ . Поскольку основная часть ресурсов уходит на вычисления, а не на бинарный поиск, оба интервала вычисляются одинаково. На выполнение требуется  $O(b)$  памяти.

Опорный интервал и калибровка, в принципе, дают аналогичные результаты, и источники это подтверждают, но мое моделирование немного благоприятствует первому варианту. Тем не менее идея полезна для лучшего понимания начальной загрузки и калибровки:

- ◆ для неограниченной начальной загрузки- $t$  не выполняется разделение на  $n = 5$  или 10;
- ◆ причина такой разбивки, по-видимому, заключается в том, что дисперсии подвыборки немного меньше, чем дисперсии полной выборки, что приводит к немного

более длинным интервалам, чем необходимо. Таким образом, проблема в первую очередь заключается во вложенных оценках, а не в опорных точках.

Технически может получиться так, что у многих интервалов  $\hat{\theta} \notin (l, u)$ , но это маловероятно, поскольку некоторые повторные выборки должны приводить к более экстремальным значениям, чем исходная, но, если это произошло, значит что-то пошло не так. Нормальная, исходная и ограниченная начальная загрузка- $t$ , а также калиброванные интервалы Чебышева, не имеют этой проблемы.

Еще одна важная конструкция для более точных интервалов основана на том, что существует преобразование, при котором  $S$  выглядит нормально. В частности (см. [21.24]), это приводит к *перцентильному интервалу*:  $(l, u) = (B_a, B_{1-a})$ . Для него требуется то же значение  $b$ , что и для опорного интервала, а также точность первого порядка. Многие пользователи выбирают этот метод, поскольку:

- ◆ он обманчиво прост — для его использования не требуется никакого аналитического мышления, включая расчеты распределения;
- ◆ на практике он работает лучше, чем опорный интервал, т. к. сдвигается в правильном направлении в случае перекоса  $S$ ;
- ◆ у него есть некоторые хорошие свойства — такие как соблюдение ограничений диапазона и инвариантность преобразования.

Между выводами опорных и перцентильных интервалов нет связи — они случайно оказываются симметричны относительно  $\hat{\theta}$ . Но это говорит о том, что перцентиль согласован для симметричного  $S$ , когда существует нормализующее преобразование.

Охват не связан с инвариантностью, и, если вы хотите преобразовать доверительный интервал, в первую очередь вычислять его не рекомендуется. Даже нормальный интервал не является инвариантным. Вы можете обрезать вычисляемые интервалы, чтобы соблюсти ограничения диапазона, — величина обрезки обычно слишком мала, чтобы заметно увеличить длину или уменьшить охват. Таким образом, преимущество инвариантности незначительно.

Для интервалов на основе квантилей, по сравнению с использованием функции квантилей для интерполяции близких значений в общем случае, округление для получения более экстремальных значений является простым, слегка консервативным выбором, который также сохраняет инвариантность преобразования.

Можно добиться улучшения, добавив коррекцию смещения (*интервал BC*) и коррекцию асимметрии (*интервал BCa*). Подробнее об этом можно почитать в работах [21.24 и 21.23]. Интервал BCa имеет точность второго порядка для гладкой статистики, но, к сожалению, доступен только в виде «черного ящика» для одиночной выборки  $f$  (а не для нескольких выборок — таких как разница медиан). Кроме того, при вычислении его «ускорения» используется *складной нож* (jackknife), который не работает для негладких  $f$  — таких как выборочная медиана, и поэтому не является надежным для получения 2-го порядка. Перцентиль имеет плохое покрытие для небольших выборок (см. [21.35]), а BC строго улучшает его. Асимметричные размеры хвостов BC и BCa создают проблемы, если значение  $b$  недостаточно велико для правильной оценки соответствующих квантилей, но в моих экспериментах не возникло проблем ни с длиной, ни с охватом.

Калибровка также эффективна с перцентильным интервалом. По умолчанию выбираются значения  $b = 1020$  и  $b_2 = 98$  при выполнении 100 000 оценок (см. [21.9]). Диапазон поиска =  $[0, 1]$ , где 1 — это наибольшее значение для получения действительного интервала из-за ограничений симметрии перцентильного интервала. Производительность получается лучше, чем у ВС и ВСа. Затраты памяти равны  $O(bb_2)$ , что намного больше, чем у калиброванного интервала Чебышева, который позволяет хранить только вложенные дисперсии:

```
class BTPCFunctor
{
    Vector<Vector<double> > sets;
    bool findLeft;
    double q, aTarget;
    pair<double, double> makeInterval(int i, double a) const
    { // перцентиль
        assert(0 <= i && i < sets.getSize());
        double b = sets[i].getSize();
        int left = max<int>(0, a/2 * b),
            right = min<int>(b - 1, (1 - a/2) * b);
        return make_pair(sets[i][left], sets[i][right]);
    }
    double operator() (double a) const
    { // возвращает <0, если значение ниже целевого
        int missCount = 0, b = sets.getSize();
        for(int i = 0; i < b; ++i)
        { // убедитесь, что не перепутали промах слева и промах справа
            pair<double, double> conf = makeInterval(i, a);
            missCount += (findLeft ? q < conf.first : conf.second < q);
        }
        return aTarget/2 - missCount * 1.0/b;
    }
public:
    BTPCFunctor(double theQ, double a): q(theQ), aTarget(a), findLeft(true){}
    void addSet() {sets.append(Vector<double>());}
    void addValue(double value) {sets.lastItem().append(value);}
    void closeSet()
    {quickSort(sets.lastItem().getArray(), sets.lastItem().getSize());}
    void flipSide() {findLeft = !findLeft;}
    double findA() const
    {
        double left = 0, right = 1, yLeft = this->operator()(left);
        return haveDifferentSign(yLeft, this->operator()(right)) ?
            solveFor0(*this, left, right).first :
            yLeft < 0 ? left : right;
    }
};

template<typename BOOTER> pair<double, double>
bootstrapDoublePercentileInterval(BOOTER& booter, int b = 1020,
int b2 = 98, double confidence = 0.95)
{
    assert(b > 2 && confidence > 0 && confidence < 1);
    double q = booter.eval();
```

```

BTPCFunctor f(q, 1 - confidence);
Vector<double> values(b);
for(int i = 0; i < b; ++i)
{
    booter.boot(); // ресемплинг
    values[i] = booter.eval();
    BOOTER booter2 = booter.cloneForNested();
    f.addSet();
    for(int j = 0; j < b; ++j)
    {
        booter2.boot(); // вложенный ресемплинг
        f.addValue(booter2.eval());
    }
    f.closeSet();
}
quickSort(values.getArray(), b);
int left = max<int>(0, f.findA()/2 * b);
f.flipSide(); // по умолчанию слева, а сейчас выполняется поиск справа
int right = min<int>(b - 1, (1 - f.findA()/2) * b);
// ограничение q
return make_pair(min(q, values[left]), max(q, values[right]));
}

```

Приведенные далее результаты моделирования говорят, что и двойной перцентиль, и ограниченные интервалы начальной загрузки- $t$  дают хорошие результаты.

Любой из этих вариантов является удачным выбором для больших бюджетов вычислений, хотя определение лучшего из них требует дальнейшего изучения. Последний кажется более безопасным для «черного ящика»  $f$  из-за его согласованности, хотя также можно утверждать, что первый обладает устойчивостью к неизвестным, основанной на инвариантности преобразования (рис. 21.2).

Корреляционная функция здесь:  $f(x) = x - 0.1x^2 + N(0, 0.1^2)$ , где  $x \sim$  стандартная форма. Метрики:  $a_{left}(aL)$ ,  $a_{right}(aR)$ , процент длины относительно точного интервала ( $\%L$ ) и асимметричные потери по Чебышеву основаны на относительных длинах (ACL). Точные значения и точные интервалы рассчитываются путем моделирования выборки из  $10^7$  образцов, когда каждая начальная загрузка запускалась 5000 раз. Значения  $b$  соответствуют значениям по умолчанию, но для опорного и некалиброванного перцентиля было выбрано значение 3000, чтобы увеличить вычисление примерно в 15 раз по сравнению с нормальным и интервалом второго порядка.

При больших  $n$  все интервалы работают одинаково во всех задачах — асимптотическая нормальность проявляется достаточно хорошо. В моделировании из методов второго порядка тестировалась только начальная загрузка- $t$ . Смешанная выборка дает меньшую частоту ошибок, чем остальные, за счет удлинения интервалов. Что еще более важно, у опорной точки охват хуже, чем у нормального, который, в свою очередь, не хуже начальной загрузки- $t$  как по охвату, так и по длине (рис. 21.3).

Использование значения  $n = 5$  является хорошим тестом на стабильность (рис. 21.4).

При  $n = 10$  нестабильность начальной загрузки- $t$  для коэффициента корреляции не исчезает, хотя и уменьшается. Проблема с толстым хвостом у большинства интервалов также не исчезает (рис. 21.5).

Распределение и статистика	Опорная точка				Перцентиль				Нормальное			
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL
Нормальное среднее	0.030	0.035	-0.025	0.250	0.030	0.034	-0.025	0.248	0.029	0.034	-0.007	0.251
Лог-нормальное среднее	0.002	0.152	-0.127	0.250	0.008	0.118	-0.127	0.294	0.003	0.131	-0.095	0.278
Медиана Леви	0.012	0.189	0.240	0.368	0.030	0.032	0.240	0.312	0.004	0.043	0.449	0.289
Нормальное отклонение	0.027	0.064	-0.087	0.278	0.006	0.109	-0.087	0.242	0.011	0.079	-0.067	0.253
2-нормальное среднее значение смеси	0.027	0.034	-0.027	0.242	0.032	0.036	-0.027	0.254	0.027	0.033	-0.009	0.243
Нормальная ошибка	0.135	0.001	0.001	0.243	0.072	0.011	0.001	0.290	0.094	0.003	0.028	0.268
Нормальная ошибка по Спирмену	0.052	0.000	0.422	0.175	0.000	0.038	0.422	0.093	0.005	0.000	0.465	0.096
Средние ранги	8.857		3.000	4.286	8.000		2.571	5.000	5.571		5.286	3.571
Смешанная выборка	BC				BCa				Начальная нагрузка -t			
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL
	0.018	0.021	0.098	0.217	0.033	0.034	-0.026	0.255	0.033	0.035	-0.024	0.256
	0.001	0.112	0.000	0.268	0.013	0.111	-0.099	0.317	0.027	0.088	-0.006	0.360
	0.003	0.036	0.589	0.289	0.021	0.040	0.158	0.310	0.020	0.039	0.165	0.306
	0.006	0.062	0.033	0.238	0.021	0.072	-0.093	0.272	0.031	0.053	-0.057	0.280
	0.015	0.020	0.095	0.203	0.033	0.038	-0.027	0.260	0.033	0.039	-0.024	0.264
	0.076	0.001	0.136	0.245	0.045	0.020	0.066	0.284	0.037	0.021	0.109	0.280
	0.002	0.000	0.619	0.066	0.016	0.017	0.224	0.224	0.014	0.024	0.273	0.229
	2.857	8.000	1.286		7.429	2.000	6.286		6.857	4.429	7.000	
Начальная нагрузка -t с ограничением	Двойной перцентиль				Двойной Чебышев							
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL
	0.023	0.023	0.073	0.232	0.025	0.027	0.041	0.239	0.024	0.027	0.038	0.236
	0.011	0.064	0.210	0.356	0.020	0.066	0.151	0.363	0.014	0.067	0.178	0.363
	0.020	0.030	0.626	0.378	0.028	0.031	0.274	0.317	0.034	0.047	0.273	0.372
	0.024	0.027	0.161	0.264	0.031	0.033	0.045	0.268	0.050	0.024	0.122	0.284
	0.024	0.027	0.070	0.244	0.029	0.032	0.054	0.261	0.028	0.033	0.035	0.256
	0.028	0.017	0.266	0.282	0.027	0.020	0.260	0.284	0.030	0.021	0.221	0.287
	0.022	0.008	0.239	0.229	0.019	0.042	0.091	0.238	0.037	0.006	0.110	0.238
	2.000	8.286	5.286		5.143	6.143	6.857		5.571	6.000	8.143	

Рис. 21.2. Характеристики нескольких интервалов начальной загрузки для нескольких задач при  $n = 30$ 

Распределение и статистика	Опорная точка				Перцентиль				Нормальное			
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL
Нормальное среднее	0.023	0.024	0.001	0.154	0.025	0.024	0.001	0.157	0.021	0.023	0.019	0.151
Лог-нормальное среднее	0.011	0.044	-0.009	0.177	0.016	0.036	-0.009	0.165	0.012	0.039	0.009	0.171
Медиана Леви	0.019	0.062	0.010	0.216	0.026	0.023	0.010	0.158	0.019	0.029	0.034	0.162
Нормальное отклонение	0.022	0.033	-0.003	0.166	0.019	0.035	-0.003	0.168	0.020	0.032	0.014	0.167
2-нормальное среднее значение смеси	0.021	0.019	0.001	0.142	0.020	0.021	0.001	0.144	0.019	0.021	0.019	0.144
Нормальная ошибка	0.039	0.012	0.000	0.170	0.032	0.018	0.000	0.160	0.033	0.012	0.016	0.160
Нормальная ошибка по Спирмену	0.037	0.011	0.028	0.170	0.017	0.034	0.028	0.169	0.024	0.018	0.046	0.152
Средние ранги	0.027	0.025	0.000	0.162	0.027	0.028	0.000	0.166	0.026	0.022	0.030	0.159
Смешанная выборка	5.125	2.500	5.000		5.375	2.500	5.500		2.625	5.125	4.125	
Нормальное среднее	BC				BCa				Начальная нагрузка -t			
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL
	0.013	0.013	0.128	0.129	0.025	0.025	0.000	0.157	0.024	0.025	0.000	0.156
	0.007	0.025	0.115	0.152	0.019	0.033	-0.008	0.163	0.027	0.027	0.005	0.165
	0.012	0.018	0.143	0.141	0.024	0.024	0.004	0.156	0.025	0.023	0.005	0.155
	0.010	0.018	0.124	0.137	0.024	0.033	-0.004	0.169	0.028	0.027	-0.001	0.166
	0.014	0.010	0.126	0.124	0.022	0.020	-0.001	0.144	0.021	0.020	-0.001	0.143
	0.023	0.007	0.126	0.147	0.029	0.020	0.000	0.158	0.027	0.021	0.002	0.156
	0.017	0.011	0.156	0.138	0.028	0.024	0.012	0.163	0.026	0.025	0.014	0.162
	0.014	0.015	0.136	0.136	0.027	0.027	-0.002	0.164	0.027	0.028	-0.002	0.166
	1.000	7.000	1.000		5.625	1.625	4.875		5.000	2.625	4.125	

Рис. 21.3. Производительность при  $n = 1000$ , по 3000 начальных загрузок



Распределение и статистика	Опорная точка				Перцентиль				Нормальное						
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL			
Нормальное среднее	0.082	0.083	-0.172	0.345	0.083	0.085	-0.172	0.348	0.075	0.077	-0.143	0.342			
Лог-нормальное среднее	0.010	0.302	-0.347	0.299	0.015	0.266	-0.347	0.343	0.010	0.273	-0.303	0.329			
Медиана Леви	0.016	0.296	###	###	0.032	0.031	###	###	0.006	0.048	###	###			
Нормальное отклонение	0.113	0.143	-0.275	0.393	0.0000	0.353	-0.275	0.186	0.004	0.202	-0.196	0.286			
2-нормальное среднее значение смеси	0.066	0.079	-0.191	0.313	0.078	0.105	-0.191	0.353	0.064	0.082	-0.163	0.324			
Нормальная ошибка	0.389	0.006	0.771	0.416	0.156	0.000	0.771	0.896	0.109	0.001	1.451	0.639			
Нормальная ошибка по Спирмену	0.305	0.002	0.389	0.384	0.305	0.000	0.389	1.030	0.223	0.001	0.689	0.699			
Средние ранги	8.143		2.000	3.286	7.714		2.000	5.143	5.571		4.000	3.429			
Смешанная выборка	BC				BCa				Начальная загрузка -t						
aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL
0.060	0.063	-0.053	0.338	0.077	0.091	-0.168	0.349	0.076	0.091	-0.143	0.358	0.024	0.027	1.170	0.493
0.006	0.250	-0.231	0.334	0.014	0.268	-0.346	0.344	0.017	0.249	-0.273	0.394	0.002	0.103	3.607	1.920
0.004	0.043	###	###	0.032	0.189	1.361	1.436	0.032	0.189	1.361	1.436	0.004	0.034	###	###
0.001	0.181	-0.111	0.280	0.033	0.314	-0.398	0.306	0.035	0.306	-0.414	0.306	0.132	0.023	2.683	0.864
0.048	0.064	-0.074	0.312	0.076	0.114	-0.188	0.359	0.080	0.120	-0.159	0.382	0.014	0.029	1.288	0.473
0.103	0.001	1.705	0.675	0.043	0.007	3.159	1.186	0.045	0.008	2.794	1.107	0.035	0.011	###	###
0.222	0.001	0.869	0.758	0.002	0.309	1.149	0.230	0.000	0.309	1.152	0.056	0.305	0.000	0.768	0.699
3.857	6.429	3.571	7.714	4.429	4.857	7.286	5.143	5.286	2.000	9.286	8.857				
Начальная загрузка -t с ограничением				Двойной перцентиль				Двойной Чебышев							
aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL				
0.025	0.028	0.485	0.343	0.044	0.048	0.156	0.355	0.027	0.030	0.467	0.350				
0.002	0.135	0.168	0.423	0.007	0.184	-0.100	0.407	0.002	0.137	0.159	0.427				
0.004	0.033	###	###	0.062	0.059	###	###	0.013	0.042	###	###				
0.098	0.062	0.101	0.420	0.067	0.306	-0.452	0.333	0.104	0.061	0.096	0.419				
0.015	0.031	0.429	0.301	0.038	0.075	0.130	0.379	0.016	0.034	0.412	0.312				
0.074	0.005	2.783	1.049	0.127	0.012	1.523	1.219	0.131	0.004	1.182	0.535				
0.305	0.000	0.696	0.732	0.305	0.000	0.997	1.506	0.280	0.000	0.774	0.761				
2.714	8.000	5.714	6.000	5.571	8.286	3.857	7.000	6.429							

Рис. 21.4. Производительность при  $n = 5$  по 30 000 начальных загрузок. Значение «####» означает на несколько порядков большее число, чем остальные; BC и BCa случайно преуспевают в медиане Леви

Выводы по полученным результатам, простоте, вычислительной эффективности и симметричной интерпретации ошибок:

- ◆ смешанное распределение кажется хорошим выбором по умолчанию, особенно для очень маленьких или больших  $n$ . Нормальное приближение достаточно точное, или вычисления  $f$  достаточно требовательны, чтобы использовать малое  $b$ ;
- ◆ можно попробовать добиться точности второго порядка ограниченной начальной загрузки или двойного перцентиле. При необходимости можно обрезать смешанную и ограниченную начальную загрузку- $t$ , чтобы соблюсти ограничения, связанные с предметной областью;
- ◆ некалиброванные интервалы на основе перцентилей кажутся бесполезными. Но интервал BC почти всегда работает лучше перцентиле и выигрывает по ожидаемой длине среди всех рассматриваемых интервалов. Последнее, возможно, связано с тем, что фактор смещения сдвигается в один хвост и удлиняет его меньше, чем обрезается другой.

Если нужно получить достаточное количество повторных выборок из распределения с толстым хвостом, никакое  $f$  не поможет достичь результата. Заманчивой идеей кажется возможность «зафиксировать» нормальные интервалы, используя оценку с надежной



Распределение и статистика	Опорная точка				Перцентиль				Нормальное							
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL				
Нормальное среднее	0.051	0.051	-0.078	0.297	0.051	0.049	-0.078	0.296	0.049	0.047	-0.057	0.295				
Лог-нормальное среднее	0.003	0.230	-0.245	0.274	0.010	0.191	-0.245	0.325	0.005	0.202	-0.202	0.307				
Медиана Леви	0.007	0.226	####	1.618	0.042	0.022	####	2.470	0.003	0.034	####	####				
Нормальное отклонение	0.062	0.100	-0.179	0.346	0.001	0.223	-0.179	0.232	0.012	0.143	-0.145	0.297				
2-нормальное среднее значение смеси	0.042	0.048	-0.090	0.276	0.051	0.058	-0.090	0.304	0.042	0.050	-0.069	0.285				
Нормальная ошибка	0.239	0.001	0.164	0.281	0.102	0.005	0.164	0.417	0.119	0.000	0.315	0.352				
Нормальная ошибка по Спирмену	0.105	0.000	0.599	0.238	0.007	0.009	0.599	0.197	0.007	0.000	0.718	0.131				
Средние ранги	8.571		2.286	4.143	7.571		2.143	4.429	5.857		5.571	4.000				
Смешанная выборка	BC				BCa				Начальная загрузка -t							
aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL	
0.036	0.034	0.043	0.278	0.051	0.049	-0.075	0.297	0.052	0.050	-0.063	0.302	0.026	0.022	0.247	0.275	
0.002	0.180	-0.117	0.305	0.012	0.178	-0.216	0.343	0.021	0.155	-0.118	0.399	0.005	0.088	1.050	0.746	
0.002	0.028	####	####	0.025	0.054	0.983	0.640	0.026	0.053	1.004	0.644	0.006	0.017	####	####	
0.006	0.123	-0.055	0.286	0.029	0.161	-0.226	0.318	0.040	0.132	-0.202	0.335	0.047	0.033	0.655	0.447	
0.030	0.036	0.029	0.266	0.054	0.061	-0.086	0.315	0.061	0.066	-0.069	0.337	0.030	0.038	0.251	0.326	
0.107	0.000	0.457	0.355	0.042	0.016	0.527	0.412	0.034	0.015	0.593	0.392	0.015	0.019	2.386	0.604	
0.007	0.000	0.898	0.134	0.007	0.010	0.568	0.194	0.007	0.010	0.675	0.207	0.105	0.001	0.594	0.580	
3.857	6.143	2.714	7.714	2.429	4.429	7.286	4.286	6.143	2.714	8.714	8.286					
Начальная загрузка -t с ограничением				Двойной перцентиль				Двойной Чебышев								
aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL					
0.026	0.023	0.237	0.274	0.030	0.029	0.181	0.288	0.028	0.026	0.198	0.279					
0.005	0.093	0.193	0.398	0.012	0.106	0.058	0.395	0.006	0.097	0.173	0.405					
0.005	0.017	####	####	0.035	0.024	####	2.446	0.011	0.030	####	####					
0.046	0.037	0.249	0.354	0.052	0.120	-0.183	0.349	0.111	0.033	0.159	0.365					
0.030	0.037	0.228	0.318	0.041	0.046	0.198	0.356	0.033	0.041	0.191	0.327					
0.030	0.015	0.901	0.435	0.020	0.023	1.643	0.533	0.034	0.020	0.773	0.440					
0.013	0.001	0.604	0.213	0.007	0.019	2.163	0.397	0.007	0.001	1.356	0.253					
2.286	8.000	6.000	5.143	6.714	7.143	3.857	7.857	7.714								

Рис. 21.5. Производительность при  $n = 10$  по 15 000 начальных загрузок

шкалой — например, MADN, но такие оценки менее стабильны, несмотря на надежность. Например, для набора  $\{0, 0, 0, 1\}$  среднее значение более значимо, чем медиана, и для медианы не обязательно иметь точность второго порядка. Кроме того, для надежной оценки, такой как медиана выборки, оценка дисперсии с начальной загрузкой будет иметь неограниченную функцию влияния, поскольку небольшая доля повторных выборок может стать проблемой. Таким образом, в некоторых случаях использование MADN может иметь смысл, но в литературе это не рассматривалось.

Точность второго порядка нужна только для сглаженных статистик (см. [21.71]), но даже у негладких начальных загрузок  $-t$  и калибровки, по-видимому, ошибка аппроксимации улучшается даже без точности второго порядка. Двусторонний охват на самом деле тоже имеет второй порядок точности для всех интервалов, хотя, похоже, это мало что меняет.

Иногда  $f$  бывают смещены, и иногда существенно. Начальная загрузка позволяет оценить смещение, используя  $\text{смещение}(f) = E[f] - \theta \approx E_B[f] - \hat{\theta}$ , что является последовательной оценкой. Может показаться, что лучше использовать  $E_B[f]$  вместо  $\hat{\theta}$  в качестве оценки  $\theta$ , но скорректированная оценка будет иметь большую дисперсию и может быть неустойчивой к предположениям, используемым в начальной загрузке. Иногда предлагается использовать для анализа значение отклонения. Эмпирическое правило гласит, что для достижения наилучших результатов погрешность должна быть равна

$\leq 0,25 \times$  стандартное отклонение (в работах [21.24 и 21.18] рекомендуется предел 0,1). Но если  $f$  выбрана разумно, проверять это не стоит, т. к. смещение оценить труднее, чем стандартное отклонение. Так что даже не пытайтесь рассчитывать его для нормальных интервалов. Кроме того,  $f(F) \approx \hat{\theta}$  теряет ценность, если оценивать смещение. Может потребоваться использовать функцию  $f(F)$  с данными, реплицированными 100 раз (хотя если  $f$  вычисляется за время  $O(n)$ , выполняется  $b$  репликаций). Но если  $f$  сильно смещена, нет смысла ее использовать, так что даже не утруждайте себя вычислением доверительных интервалов. Интервалы ВС и интервалы второго порядка автоматически корректируют смещение (иногда речь идет о медианном смещении), хотя у каждого из них свой механизм.

Точность второго порядка для некоторых интервалов предполагает, что начальная загрузка лучше, чем аналитические процедуры, которые обычно имеют точность первого порядка, для малых  $n$  (см. [21.84]). Но это дорого в вычислительном отношении и порождает дополнительные ошибки Монте-Карло. Используйте аналитические методы там, где они применимы, а начальную загрузку лучше приберечь на тот случай, когда хорошие методы не работают. В некоторых задачах начальная загрузка работает лучше, чем в других, и рекомендуется проверить, не изучалось ли ее применение для конкретной задачи ранее.

Вы можете расширить начальную загрузку, чтобы работать с многовыборочными функциями. Например, чтобы вычислить доверительный интервал для разности медиан двух выборок, нужно провести повторную выборку из обеих и вывести разницу медиан повторной выборки:

```
template<typename FUNCTION, typename DATA = double> struct MultisampleBooter
{
    Vector<Vector<DATA> const* > const& data;
    Vector<Vector<DATA> > resample;
    FUNCTION f;
    MultisampleBooter(FUNCTION const& theF = FUNCTION()): f(theF) {}
    void addSample(Vector<DATA> const& theData)
    {
        data.append(&theData);
        resample.append(theData);
    }
    void boot()
    {
        for(int i = 0; i < data.getSize(); ++i)
            for(int j = 0; j < data[i].getSize(); ++j)
                resample[i][j] = data[i][GlobalRNG().mod(data[i].getSize())];
    }
    double eval()const
    {
        assert(data.getSize() > 0);
        return f(resample);
    }
    MultisampleBooter cloneForNested()const
    { // для интервала начальной загрузки-t
        MultisampleBooter clone(f);
        for(int i = 0; i < data.getSize(); ++i)
            clone.addSample(*data[i]);
    }
};
```

```

        return clone;
    }
};

```

Может потребоваться корректировка нескольких тестов (обсуждается далее в этой главе), если они дают несколько доверительных интервалов. При наличии односторонних интервалов двусторонний интервал вычисляется в точке  $a/2$  и используется искомая половина. Также можно увеличить значение  $b$ , чтобы получить лучшее разрешение.

Как и в любом алгоритме Монте-Карло, при повторном выполнении результаты могут различаться. Значения  $b$  по умолчанию выбираются таким образом, чтобы ошибка из-за случайной выборки была намного меньше, чем изменчивость данных (см. [21.24]). Поэтому, если у вас не хватает вычислительных возможностей, выберите значение  $b$  в 10 или 100 раз больше, чтобы уменьшить изменчивость выборки до относительно незначительной. Но значение  $b > 10^6$  избыточно. В качестве проверочной эвристики для результатов анализа можно запустить его несколько раз и посмотреть, входит ли первоначальный результат в типичные случаи.

## 21.16. Когда использовать начальную загрузку?

Допустим, вы хотите знать, в каких случаях работают определенные доверительные интервалы. В этом случае более простой вопрос заключается в том, когда  $B \rightarrow S$ . Вопрос имеет смысл только асимптотически, что не должно исключать хорошего поведения конечной выборки. Даже тогда нет полезных в общем случае условий. Имеющиеся результаты (см. [21.71]) гарантируют, что начальная загрузка согласована, когда:

- ◆ у функции  $f$  есть аналогичная теорема Берри — Эссена (простое доказательство приведено в работе [21.36]). Если  $f$  — это среднее выборки, основная идея состоит в том, что среднее значение, стандартное отклонение и третий момент последовательно оцениваются по их аналогам в EDF. Таким образом, теорема Берри — Эссена применима и к EDF, и с ее помощью можно показать, что  $t$ -статистики реального распределения и EDF сходятся. Эта теорема известна только для среднего значения, на практике подойдет любая оценка  $f$  с асимптотически нормальным  $S$ ;
- ◆ функция  $f$  достаточно гладкая. Она должна быть дифференцируема по Фреше или Адамару, и влияние функции  $f$  должно быть незначительно ограничено. Но даже медиана выборки не обладает такой дифференцируемостью, поэтому  $f$ -«черный ящик» точно не будет дифференцируемой. А вот начальная загрузка в любом случае работает. Требования, что функция  $f$  — липшицева при  $\theta$  и близких значениях, достаточно, чтобы избежать проблем с гладкостью, когда достаточно быстрое непрерывное падение, по сути, близко к ступенчатой функции. Таким образом, хотя гладкость позволяет доказать согласованность для конкретного  $f$ , на практике она бесполезна.

Другой набор условий (см. [21.22], изначально эта идея была представлена в работе [21.8]) работает с набором распределений  $N$ , окружающих  $T$ . В частности, для некоторой функции  $G$ , связанной с  $f$ ,  $\forall A \in N$ :

- ◆  $G_{A,n}$  должна сходиться равномерно слабо;
- ◆ отображение из  $A$  в  $G_{A,n}$  должно быть непрерывным.

К сожалению, эти условия невозможно проверить на  $f$ -«черном ящике», даже имея хорошие знания предметной области. Таким образом, к работе с такими  $f$  вообще не при-

менить полезные теории. Зато при работе с обычными  $f$  проблемы вряд ли возникнут. Но примеры неудач также многочисленны (конкретные примеры приведены в работе [21.71] и дополнены в работе [21.67]):

- ◆ когда оцениваемый параметр находится на границе (см. [21.3]). Типичным примером является максимум — начальная загрузка не позволяет найти его стандартное отклонение;
- ◆ когда  $f$  недифференцируема в точке  $\theta$ .

Тот факт, что  $B \rightarrow S$ , не обязательно означает, что начальная загрузка будет работать. Например, для  $g(x) = x^2$  применение  $g$  к выборочному среднему гладкое, поэтому  $B \rightarrow S$ . Но при  $\mu = 0$  любой доверительный интервал будет пропускать границу. Необходимо также, чтобы работало моделирование. В целом построение доверительных интервалов предполагает немного больше, чем требуется для условий согласованности, — например, чтобы распределение  $S$  имело конечную дисперсию, хотя неудачи для последнего обычно также являются неудачами для первого.

## 21.17. Проверка гипотез

Иногда нужно посмотреть, исключается ли из данных определенное значимое значение  $\theta$  — например, 0. Проверим, что некоторое значение не попадает в доверительный интервал:

```
bool confIncludes(pair<double, double> const& interval, double value)
{return interval.first <= value && value <= interval.second;}
```

Так, имея два алгоритма, можно проверить доверительный интервал на разнице некоторых показателей производительности при нуле, что, если она присутствует, не исключает эквивалентной производительности в отношении данных.

Для построения теста можно использовать другой механизм. Пусть утверждение « $\theta$  содержит желаемое значение», является *null-гипотезой*. Обычно такая гипотеза принимается для одного конкретного значения  $\theta$  (что не обязательно). В этом случае *null* называется *простым*, иначе — *составным*. *Альтернативные гипотезы* являются ее дополнением, но объединение не обязательно должно охватывать всё пространство выборки. Обычно *null* означает, что рассматриваемое утверждение ложно, а альтернатива — что оно истинно.

Тесты обычно выполняются за время  $O(n)$ . Общий тест гипотезы (рис. 21.6):

1. Выбрать статистику — например, выборочное среднее.
2. Определить его распределение (называемое *null-распределением*) при расчете на выборке, предполагая, что *null*-значение истинно, — например, нормальное с неизвестной дисперсией, с центром в точке  $\theta$ .
3. Вычислить  $p$ -значение =  $\Pr(\text{может иметь в хвосте равное или большее значение, чем статистическая выборка})$  — например, положение хвоста  $z$ -показателя.
4. Если оно достаточно мало, отклонить *null*-значение и предположить, что альтернатива верна, — например, проверить утверждение « $p$ -значение  $\leq a = 0,05$ ».
5. В противном случае недостаточно доказательств, чтобы отклонить гипотезу, поэтому предположим, что альтернатива не доказана.

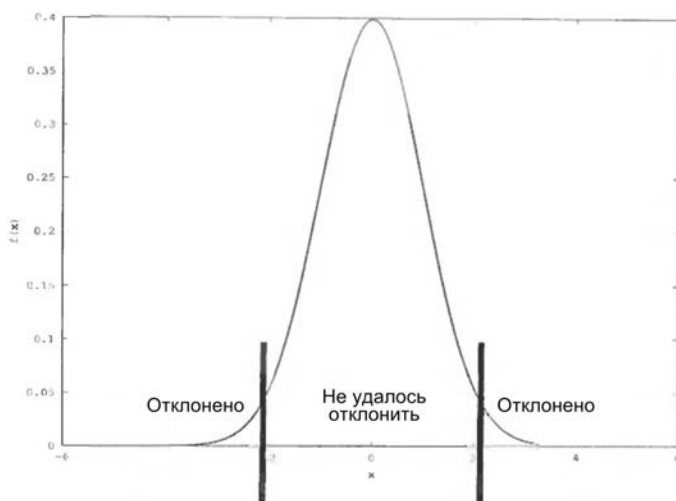


Рис. 21.6. Области решения проверки гипотезы для null

Еще один способ увидеть нулевое распределение основан на группировании данных. Например, если популяция делится пополам в результате подбрасывания монеты, в статистике подпопуляций возникают некоторые случайные различия. Эти различия можно считать шумом, если разделение не основано на какой-либо переменной. При null-распределении шума тест показывает, стала ли разница под влиянием шума больше, чем ожидалось. Это похоже на аргументацию от противного: если null плохо объясняет данные, он не может быть правдой, поэтому альтернатива должна быть. С точки зрения языка «принять требование» = «отклонить null» = «значение, исключенное данными».

До сих пор тестирование принадлежности к доверительному интервалу и проверка гипотез практически ничем не различались. Тест, который не получен из доверительного интервала, обычно можно преобразовать в доверительный интервал. Для этого нужно сдвинуть данные или их преобразование на значение, эквивалентное смещению параметра, — т. е. перевернуть логику построения тестов из доверительных интервалов. Например, доверительный интервал для оценок квантилей был получен путем инвертирования критерия знака. Но это не всегда работает — например, не удастся инвертировать критерий перестановки для коэффициента корреляции (обсуждается далее в этой главе).

Двусторонний тест, по сути, выполняет трехстороннее сравнение. Проверяется гипотеза о неравенстве, но, если статистика излишне экстремальна, нужно узнать, слишком ли она велика или слишком мала. Двусторонний тест эквивалентен двум односторонним тестам при  $\alpha/2$  (что имеет смысл с поправкой Бонферрони, обсуждаемой далее в этой главе). Таким образом, при 100-процентном интервале для  $p$ -значения и  $\alpha = 5\%$ , 95 % соответствует равенству, 2,5 % — одному хвосту и 2,5 % — другому. Также возможно асимметричное распределение хвостов. Если статистика попадает в любой из хвостов, null отклоняется с вероятностью 95% и делается вывод, что одна альтернатива имеет статистически более высокое значение на уровне 5%.

Подобно максимальной вероятности для построения оценок можно использовать *тесты отношения правдоподобия* (см. [21.79]):  $2(LL(MLE(\text{модель}, \text{данные})) -$

–  $LL(MLE(\text{модель, данные, null-область}))$ ) = хи-квадрат ( $d$ ), где  $d$  = измерение (полный набор параметров) – измерение (набор параметров при ограничении нулевой областью). Например,  $d = 1$  для проверки того, выполняется ли равенство  $\theta = 0$ . Нужны условия регулярности, чтобы логика работала во всех случаях (см. [21.20]), а в одностороннем случае полной ясности нет. В работе [21.69] приведены некоторые расширения.

Например, можно показать, что  $t$ -тест, инвертирование которого дает соответствующий доверительный интервал для среднего значения, использует статистику отношения правдоподобия, но с точным, а не с асимптотическим распределением (см. [21.82]). Имейте в виду, что, как и оценки максимального правдоподобия, тесты отношения правдоподобия могут работать плохо, несмотря на хорошие асимптотические свойства (см. [21.46]).

## 21.18. Сравнение тестов

Несмотря на математическую эквивалентность с доверительными интервалами, тесты несколько отличаются друг от друга, в том числе и по терминологии. Те же рассуждения, что и для доверительных интервалов, предполагают, что  $\alpha = 0,05$ . Меньшие значения в некоторых случаях имеют смысл, но важно помнить, что  $p$ -значения очень чувствительны к незначительным отклонениям допущений из-за обычно больших ошибок аппроксимации в хвостах.

Тест делает ошибки следующих видов:

- ♦ **I рода** — ложное отклонение нуля — например, заключение в тюрьму невиновного. То же самое для доверительных интервалов;
- ♦ **II рода** — ложная неспособность отвергнуть нуль — например, освобождение виновных. Связано это с тем, сколь коротким является инвертированный доверительный интервал.

Ошибки I рода более важны, потому что с научной точки зрения нам не хотелось бы принимать много ложных утверждений, даже если для этого приходится оценивать многие другие как несущественные. Таким образом, тесты пытаются контролировать этот процесс в явном виде — т. е. если отклонение основано на том, что  $p$ -значение  $\leq \alpha$ , ошибочные решения принимаются в  $\leq \alpha$  % тестов. Нам не хотелось бы использовать тест, который не удовлетворяет этому условию, по крайней мере, асимптотически или при некоторых условиях. Ошибки II рода не поддаются контролю и зависят от альтернативы и  $n$ . Null выбирается из-за того, что отказ от него обходится дороже, чем альтернатива.

Оба вида ошибок связаны через *мощность (power) теста* =  $\Pr(\text{отклонения заданного } \theta)$ :

- ♦ контроль ошибки I рода означает, что  $\max_{\theta \in \text{null}} \text{мощность}(\theta) \leq \alpha$ ;
- ♦ у ошибок II рода  $1 - \min_{\theta \in \text{альтернативная}} \text{мощность}(\theta)$ . В худшем случае это значение стремится к  $\alpha$ , так что в лучшем случае можно получить ошибку контроля II рода, когда есть некоторая зона безразличия, близкая к нулю. Выбор  $n$ , необходимого для конкретной границы ошибки, делается на основе анализа мощности (обсуждается далее в этой главе).

Если есть два теста, которые контролируют ошибки рода I, вам нужен более мощный тест по сравнению с альтернативным набором. Это можно рассматривать как функцию риска, использующую потери в диапазоне от 0 до 1 за ложное отклонение нуля, так что в итоге получается почти та же теория сравнения, что и для оценки. Соотношение допустимости образуется, когда один тест *равномерно мощнее*, а в частных случаях (в основном для односторонних тестов) получают *равномерно наиболее мощные* (Uniformly Most Powerful, UMP) тесты. Они определяются по отношению к конкретному нулю и альтернативе. Мощность сравнивается только с альтернативой, т. к. нулевая мощность должна оставаться  $\leq \alpha$ . Поэтому даже для одного значения  $\theta$  нельзя получить единое значение риска.

Достичь существования UMP можно за счет того, что тесты отношения правдоподобия являются равномерно наиболее мощными, когда отношение является монотонным. Это работает в основном для односторонних тестов (и тестов эквивалентности, обсуждаемых в комментариях, — см. [21.70]).

Критерии и интервалы, образованные друг из друга, наследуют те свойства оптимальности, которыми обладал исходный критерий, — например, наибольшую мощность (см. [21.46]). Например, тест, в котором null является простым, и UMP  $\forall \theta$  приводит к интервалу UMA (см. [21.70]). Это справедливо только для нерандомизированных тестов, хотя инверсия работает независимо ни от чего (но теряет приемственность при наличии рандомизаций, если только для инверсии не используются обычные случайные числа, как для тестов перестановки — см. далее в этой главе).

Для тестов, в отличие от оценок, концепцию минимакса не используют, потому что мощность ближе к нулю обычно очень мала (см. [21.52]). Таким образом, анализ мощности работает только с репрезентативной точкой, заданной областью безразличия. Байесовская априорная оценка, которая помещает тонкий хвост в область безразличия, также может использоваться, но она не особенно полезна. Однако при определенном безразличии альтернатива может рассматривать мощность на ней как риск и выполнять сравнение аналогично оценщикам.

Поскольку тесты UMP встречаются редко, рассмотрим их общие ограничения, как и для оценок, хотя они только расширяют набор частных случаев:

- ◆ *несмещенные тесты* — здесь (по крайней мере, для простого null) смещение равняется точке наименьшей мощности —  $\theta$ . Постарайтесь добиться того, чтобы истинные значения отклонялись реже, чем ложные, но смещенный тест может в обоих случаях иметь меньшую мощность. Понятия дисперсии, аналогичного используемому в доверительных интервалах, для тестов не существует, поэтому сформировать MSE нельзя;
- ◆ *инвариантные тесты* — в отношении однозначных преобразований — например, перевода и масштабирования. UMPU (несмещенный UMP) и UMPI (инвариантный UMP) обычно совпадают (см. [21.70]).

ARE теста можно вычислить несколькими способами, и наиболее распространенный из них, по существу, сводится к ARE оценщика. Для конкретной известной альтернативы отношение:

$$\frac{n(\text{тест1, альтернатива, целевая мощность})}{n(\text{тест2, альтернатива, целевая мощность})}$$

рассматривается как  $\theta \rightarrow \theta_{\text{null}}$  (см. [21.20]). Метод сводится к сравнению асимптотических дисперсий базовых оценок.

## 21.19. Эффективное использование тестов

Учитывая, что доверительные интервалы, если они известны, всегда несут строго больше информации по сравнению с результатами тестов, встает вопрос: полезны ли тесты для чего-либо, кроме построения доверительных интервалов путем инверсии? Когда единственной целью доверительного интервала является тестирование, чистые тесты в целом имеют некоторые преимущества:

- ◆ в некоторых случаях вычисление статистики на инвертированном доверительном интервале не имеет смысла, поэтому результат теста более интерпретируем. Например, если вы хотите сравнить два алгоритма на несвязанных входных данных, чтобы решить, какой из них лучше, показатели вроде процента качества не имеют смысла. Нужна общая оценка, которая, как и  $\theta$ , должна существовать, чтобы результат теста имел смысл. Но в общем случае для выставления оценок по каждому заданию не хватает информации;
- ◆ вычисления могут быть более эффективными — например, проверка знаков выполняется за время  $O(n)$ , занимает пространство  $O(1)$  и выполняется в реальном времени, в то время как выборочное вычисление медианы менее эффективно и не работает в реальном времени;
- ◆ в качестве результата можно указать  $p$ -значение и при необходимости сравнить его с  $\alpha$ . Например, для важных решений может потребоваться меньшее или большее значение  $\alpha$ , чем для менее важных, а в некоторых задачах оно может быть известно заранее. Это способно облегчить реализацию, хотя в некоторых случаях, например при использовании теста Немецкого (обсуждается далее в этой главе), с точки зрения вычислений удобнее и эффективнее передавать значение  $\alpha$  в качестве параметра. Так, вы можете вычислить критическое значение статистики — например, оценку  $z$ , равную 2, и передать ее, чтобы избежать оценки распределения;
- ◆ если известно много результатов, корректировка множественности (обсуждаемая далее в этой главе) работает с тестами несколько лучше, чем с доверительными интервалами, что приводит к более высокой мощности.

Тем не менее в большинстве случаев лучше указывать доверительные интервалы. Из отмеченного только изменение значения  $\alpha$  является потенциальной проблемой. Но для другого значения  $\alpha$  обычно можно пересчитать интервал. Учитывая, что значения  $\alpha$ , как правило, стандартны, а хорошие отчеты содержат информацию, необходимую для воспроизведения результатов, это редко вызывает проблему.

Доверительный интервал не дает  $p$ -значения. Но для целей интерпретации, если удастся вычислить достоверность при любом заданном  $\alpha$ , можно использовать бинарный поиск, чтобы поместить  $p$ -значение на границе, а  $p$ -значение будет равно значению  $\alpha$ . Однако такой результат кажется бессмысленным.

Типичный пример использования тестов — определение того, лучше ли вновь найденная альтернатива, чем существующая. Это позволяет избежать обращения к бесполезным альтернативам заинтересованными сторонами. Но и этот подход далеко не безупречен. Интуитивно кажется, что тестирование ничего толком не делает, потому что



оно решает, чем  $\theta$  не является, а цель противоположна. Фактически основной научный вопрос заключается в том, полезно ли тестирование вообще:

- ♦ отказ от null-значения, которое исключает нулевую разницу, не исключает разницы в  $\varepsilon$ . Например, изначально тест знаков использовался, чтобы показать, что пропорция рождений мужчин и женщин не равна строго 50/50 при любом разумном значении  $a$ , т. к. значение  $p$  равно  $2^{-82}$ , но теперь эта пропорция считается равной  $\approx 53/47$ ;
- ♦ неверный результат может не быть отклонен, если не было просмотрено достаточно данных, чтобы тест набрал мощность.

С доверительными интервалами этого не происходит:

- ♦ отсутствие членства в интервале по-прежнему связано с интервалом, который говорит, где должно быть значение  $\theta$  — далеко или близко. Доверительный интервал и оценка дают некоторое представление о том, насколько  $\theta$  близко к искомому значению, и часто считают достаточно близкое искомое значение достаточно точным, несмотря на то, что оно не содержится в интервале;
- ♦ членство связано с длиной интервала, поэтому легко увидеть, является ли интервал слишком длинным для получения достаточного количества данных. Интервал содержит всю нужную информацию и исключает необходимость проверки конкретного null-значения, а короткий интервал дает практически полную информацию о значении  $\theta$ .

Но исключать тесты как бесполезные во всех случаях, как это делается в некоторых источниках, неправильно. Значимость теста зависит от внешней информации. То есть тесты полезны при наличии дополнительных знаний о предметной области. В частности, для некоторых областей, таких как правительственные решения и судебные дела, любая разница имеет значение, а для некоторых других задач значение  $\theta$  может быть трудно оценить. Зададим, к примеру, вопрос: не ухудшает ли ваше самочувствие употребление кофе в течение всего дня, и не лучше ли чередовать кофе и воду? Данные об этом можно получить, объединив данные о днях с тем и другим вариантом употребления. Результат теста «в среднем стало лучше» достаточно полезен, но трудно установить, насколько именно стало лучше.

Приведем несколько примеров, когда тесты полезны (конкретные тесты обсуждаются далее в этой главе):

- ♦ в задачах выбора новых лекарств и экспериментально подтвержденных научных идей новый объект должен быть значительно лучше имеющегося — это служит базовой точкой для дальнейшего изучения, хотя на него часто можно ответить, используя доверительные интервалы. При этом суды выносят приговор виновной стороне в ходе состязательного судебного процесса, а фармацевтические компании проводят дополнительные исследования, чтобы решить, выгодно ли инвестировать в новое «значительно лучшее» лекарство;
- ♦ тесты совпадения распределений хороши для модульного тестирования генераторов случайных чисел. Нужно провести много тестов и посмотреть, близка ли доля отказов к  $a$  (это позволяет позже оценить долю приблизительных тестов и генераторов);
- ♦ проверка знаков используется при построении дерева регрессии;
- ♦ ранги Фридмана — это хороший способ сравнить алгоритмы на множестве входных данных, даже если соответствующие тесты значимости не учитываются.

Приблизительный доверительный интервал обычно лучше, чем менее приблизительный тест. Не следует использовать основанные на симметрии ранговые тесты, такие как *знаковый ранг Уилкоксона* и *Уилкоксон — Манн — Уитни* (подробности приведены в работе [21.28]), потому что усеченное среднее выборки, которое становится более качественным при наличии симметрии, связано и с оценкой, и с доверительным интервалом (можно вычислить доверительный интервал, инвертируя тест ранга, но соответствующая оценка *Ходжеса — Лемана* выполняется за время  $O(n^2)$  для очевидной реализации. В источнике [21.74] приведены сравнения с той же разбивкой 29%-ной выборки с усеченным средним значением). Ранговые тесты также чувствительны к асимметрии (см. [21.102]).

Часто  $p$ -значению пытаются придать смысл, обычно ошибочный.  $p$ -значение — это минимальное значение  $\alpha$ , при котором null отвергается. С точки зрения интерпретации  $p$ -значение — это вычислительный артефакт, который в конечном итоге можно сравнить только с некоторым значением  $\alpha$ . Свойство экстремального значения — это вычислительное определение: статистика случайна, а  $p$ -значение — это просто ранг ее экстремальности, но интерпретировать эту информацию и извлечь из этого пользу нельзя. Любые операции, кроме сравнения с  $\alpha$ , ошибочны.

Когда  $p$ -значение вычислено, в конечном итоге оно используется, чтобы принять или отклонить null-значение. Неважно, как принимается это решение. Во многих источниках говорится, что перед просмотром значения  $p$  необходимо выбрать  $\alpha$ , но это неверно — заинтересованные стороны не могут отклонить значение 0,06, когда в домене используется значение 0,05. Поэтому нет разницы, какое значение  $\alpha$  использовать. Таким образом, можно отклонить результат, если  $p < 0,001$ , не отклонить, несмотря ни на что, если  $p > 0,2$ , или принять решение на основании еще каких-то факторов. Также можно создать  $\alpha$  корзины. Например, для  $0,05 \leq p < 0,001$  можно организовать стандартное финансирование для некоторого научного исследования, а для  $0,001 \leq p$  — дать расширенное финансирование и т. п. Подобные многофакторные решения, возможно, являются наиболее правильным способом рассмотрения значения  $p$  как некоторой доказательной силы, хотя технически  $p$  таковой не является. С помощью доверительных интервалов также можно вычислить несколько значений  $p$  и использовать то из них, которое соответствует логике, как в приведенном примере с финансированием. Например, учитывая историю отказов, можно вычислить «личное эффективное значение  $\alpha$ », равное наибольшему отклоненному значению  $p$ . Но это уже работа с неправильной последовательностью. Если будет выполнено бесконечное количество тестов, значение  $p$  увеличится, чтобы оно соответствовало фактически используемому  $\alpha$ .

Нельзя думать, что значение  $p$  является вероятностью того, что null истинен. В частотной статистике это невозможно — ошибка оценки определяется только по отношению к конкретной выборке. Чтобы узнать эту вероятность, нужна точная априорная информация и байесовская статистика (далее в этой главе). Значение  $p$  является лишь частью апостериорной вероятности и в лучшем случае показывает относительное ранжирование — т. е. для двух случайно выбранных гипотез позволяет сказать, какая из них более вероятна. Небольшое  $p$ -значение дает большой набор  $\alpha$ , что приводит к отклонению null-значения, поэтому оно может быть полезным с точки зрения интерпретации, исходя из предположения, что используется много таких  $\alpha$ . То есть очень низкое  $p$ -значение удовлетворит большинство людей, принимающих решения. Возможно, высокая степень удовлетворенности от низких  $p$ -значений имеет смысл только с этой точки зрения. Но ни одна из этих идей не кажется полезной с практической точки зрения.

Еще одна ошибка — считать  $p$ -значение вероятностью совершить ошибку. Например, рассмотрим гипотетическую ситуацию, когда тест проверяет 20 истинных и 80 ложных утверждений (это на самом деле проценты) при  $\alpha = 0,05$ , и тест имеет мощность 80%. Это значит, тест примет  $\approx 16$  истинных утверждений и 4 ложных утверждения. В этом случае 25% принятых утверждений оказываются неверными, но их может быть меньше или больше в зависимости от распределения набора проверяемых null-значений. В психологии утверждается, что  $> 50\%$  опубликованных гипотез, принятых при 0,05, ложны. Таким образом, большинство проверенных гипотез ложны. Гораздо большее беспокойство вызывает вероятность того, что большинство преступников никогда не будут пойманы, а большая часть населения невиновна, если предположить, что «вне разумных сомнений» эквивалентно  $\alpha = 0,05$ .

Заманчиво думать, что значение вроде  $p = 10^{-6}$  означает, что соотношение мужчин и женщин определено верно. Но для точных вычислений нужен дополнительный контекст:

- ◆ «невозможные» события — такие как повторные выигрыши в лотерею, при многократном тестировании происходят чаще (обсуждается далее в этой главе, множество примеров также приведено в работе [21.34]);
- ◆ среди миллионов паттернов трудно выделить случайные события и причинно-следственные связи — даже алгоритмы машинного обучения не могут этого сделать, потому что им нужна другая информация, помимо данных;
- ◆ среди судебных исков, основанных на статистических закономерностях, «нарушение» типа дискриминации при приеме на работу, появится случайно, если решение будет принято случайным образом.

Многие тесты делают предположения, которые трудно проверить, поэтому на практике такие предположения игнорируются. В этом случае многие свойства теста, такие как контроль ошибок первого рода, становятся недействительными. Как и в случае с доверительными интервалами, обычно предполагается, что модель теста имеет некоторую погрешность аппроксимации, но получаемые выводы все еще достаточно близки к истине, чтобы быть полезными. Таким образом, строгий контроль над ошибками первого рода менее важен, чем кажется, но все равно желателен. В частности, разные методы расчета могут привести к разным  $p$ -значениям, поэтому следует осторожно выбирать предположения, которым можно доверять.

Имейте в виду, что определение значимости не является окончательным. Так, в суде вкупе с другими факторами рассматриваются допустимые доказательства. Автоматическая окончательная зависимость от уровня значимости имеет смысл только в том случае, если она согласована до проведения теста — например, для принятия новых лекарств агентством FDA. Для научных выводов низкое  $p$ -значение предполагает поднятие вопроса о продолжении исследования. Тест можно считать чем-то вроде экзамена на водительские права: те, кто его сдал, не обязательно хорошие водители, а те, кто не сдал, не обязательно плохие, но первые с определенной долей истинности считаются достойными получения прав, а последние — нет.

Существует потребность в точных процедурах — т. е. таких, которые гарантируют выполнение требований и в случае тестов имеют максимальную мощность. Но получить их можно лишь в некоторых случаях. Какая процедура является точной, интуитивно неясно. Это должна быть теоретическая конструкция доверительного интервала, кото-

рая имеет информацию  $\theta$  и  $S$ . Например, рассмотрим  $x$  переменных из нормального распределения. Критерий Стьюдента — точен, но основанный на нем доверительный интервал не таков, потому что он включает значение  $\sigma$ , которое оценивается как  $s$ , и  $s^2 \sim \chi^2$  с некоторыми степенями свободы. То есть для фиксированных значений  $x$  значения  $s$  могут меняться. Таким образом, доверительные интервалы, рассчитанные для разных выборок, имеют разную длину, хотя теоретическая конструкция имеет вполне конкретную длину. Но у теста  $t$ -статистика имеет распределение, не зависящее от  $\sigma$ . То же самое и с доверительным интервалом для медианы — как только эмпирическое распределение инвертируется путем поиска квантилей, найденные значения сводят на нет любую точность. Таким образом, в общем случае процедуры без предположений не являются точными, а инвертирование точных тестов не позволяет получить точный доверительный интервал.

Интересная дилемма возникает, когда несколько исследований дают разные  $p$ -значения. В этой ситуации возможны следующие решения:

- ◆ использовать наименьше из них, нужное для настройки многократного тестирования (обсуждается далее в этой главе);
- ◆ нескорректированное «первичное» значение  $p$  распределено равномерно, поэтому наименьшее значение из  $k$  подчиняется бета-распределению (см. главу 6. *Генерация случайных чисел*). Таким образом, вместо корректировки множественности можно рассчитать это распределение и инвертировать его, чтобы получить скорректированное значение  $p$ ;
- ◆ выбрать произвольное исследование либо случайным образом, либо используя некоторую информацию, если какое-то из исследований дает более надежную конструкцию или наибольший размер выборки;
- ◆ если возможно, вместо всего здесь отмеченного можно объединить все данные и провести один тест. Это позволяет избежать проблем с малой мощностью — таких как существование нескольких плохих исследований, которые не могут отклонить нулевое значение, вместо одного, чья статистика имела бы достаточно узкий доверительный интервал, чтобы его можно было отклонить. Комбинированные методы рассматриваются в рамках *метаанализа* (обсуждается в комментариях);
- ◆ исследования можно считать аналогичными собеседованиям — тот, кто прошел один раунд, переходит к следующему. Это позволяет новым исследованиям игнорировать предыдущие исследования, что, собственно, новые исследования и должны делать. Но неясно, как относиться к исследованиям, у которых одни  $p$ -значения велики, а другие — нет, — это зависит от других факторов. С доверительными интервалами и оценками такая проблема серьезно не стоит.

Интуитивное понимание тестов ведет ошибкам, потому что человеческий разум ищет более простые объяснения. Приведем некоторые ошибки, которые ранее не обсуждались:

- ◆ во многих учебниках сравнительные или оценочные тесты описываются как проверка того, выполняется ли равенство  $\theta = 0$ , что для естественных экспериментов почти наверняка окажется неверно из-за различий в каком-то десятичном разряде. Но можно проверить, исключается ли  $\theta = 0$  данными, как с доверительным интервалом. Смысл теста состоит в том, чтобы ответить на этот вопрос для имеющихся на текущий момент данных;

- ◆ любое распределение затрат/прибыли, основанное на  $p$ -значении или даже на  $\alpha$ , не имеет смысла. Вы получаете прибыль от принятых истинных утверждений, много теряете от принятых ложных утверждений, немного теряете от непринятия истинных утверждений и не имеете ни убытков, ни прибыли, если не отвергнете ложные утверждения. Предполагая, что все принятые утверждения ложны и все неудачно отвергнутые верны, вы можете получить допустимое, но бесполезное ограничение. Использование точных ошибок I и II рода ситуацию не спасает. Можно получить лучшую оценку с использованием байесовской логики только при наличии хорошего априорного значения. Иначе в лучшем случае можно изучить эти затраты и установить соответствующие цели по мощности и мощности для конкретной области;
- ◆ утверждение «тесты с малой мощностью распространены и приводят к тому, что слишком много ложных null-значений не отбрасывается» справедливо, но проблема не в тестах. Если тест основан на небольшой выборке и не отклоняет null, сам эксперимент был проведен плохо. Это равносильно получению широкого доверительного интервала. Распространенный совет сделать расчетную мощность  $\geq 0,8$  — это всего лишь рекомендация для лучшего проведения эксперимента. Хотя оппортунистическое тестирование, основанное на специальных доступных данных, тоже полезно, ему должен предшествовать некоторый анализ мощности.

О некоторых других распространенных ошибках можно почитать в работе [21.31].

## 21.20. Валидация исследований

Любой статистический результат верен только с некоторой вероятностью. В лучшем случае можно доверять процедуре, которая дала некоторый результат, осознавая при этом, что на этот раз она может оказаться ошибочной. Оценки, длина доверительного интервала и  $p$ -значения имеют распределения, дисперсия которых уменьшается с увеличением  $n$ , но редко становится пренебрежимо малой.

Например, для основанных на CLT доверительных интервалов среднего значения длина является функцией  $s^2$ , которая в нормальной модели равна хи-квадрат. Таким образом, на многих интервалах длина иногда оказывается очень большой. Для распределения с толстым хвостом это еще большая проблема. Асимптотическая скорость также имеет константу, которая подчиняется некоторому распределению, и возможны редкие колебания порядка величины. Даже для ограниченных распределений, где неравенство Хёффдинга дает интервалы без мешающих параметров, которые всегда имеют одинаковую длину, первичная статистика содержит некоторые вариации. Однако ничто из этого не мешает доверять результату разумной процедуры. Например, предполагается, что интегралы Монте-Карло точны до предельных значений CLT, хотя иногда это не так.

Байесовская логика (обсуждаемая позже в этой главе) иллюстрирует не правильность конкретного результата, а только веру в такой результат, основанную на некоторых исходных предположениях и данных.

Тем не менее результат считается статистическим доказательством и никогда полностью не подтверждается, если нет внешней информации. Так, Верховный суд США постановил, что статистические данные (такие как совпадение отпечатков пальцев) сами по себе не имеют значения (см. [21.31]), и нужно рассматривать другие факторы. На-

пример, если кто-то, бросая монету, получил «орла» 20 раз подряд, гораздо более вероятно, что этот человек имеет специальную подготовку или монета «неправильная», чем то, что у него есть способность какого-то сверхъестественного управления результатом.

У любого опубликованного результата есть смещение, связанное с самой публикацией — обычно принимаются только значимые значения  $p$  и короткие доверительные интервалы, поэтому значение  $a$  после публикации оказывается выше из-за предварительного отбора, как объяснялось ранее для значений  $p$ .

В конце концов сделанный вывод о том, подтверждено ли что-то, опровергнуто или остается под вопросом, оказывается несколько произвольным. При использовании голь статистики обычно требуется несколько убедительных исследований, чтобы что-то уверенно утверждать и исключить любые факторы эксперимента и экспериментатора. Обычно хочется, чтобы внешние данные подтверждали это. Утверждения, которые не могут быть подтверждены более надежными последующими исследованиями, обычно опровергаются. Таким образом, научные теории, основанные на экспериментах, считаются установленными фактами, если их не удастся опровергнуть в течение достаточно долгого времени, но случаются ошибки, и даже давние теории могут быть в конечном итоге опровергнуты или конкретизированы. Например, было обнаружено, что даже ньютоновская механика является лишь приближением, а не истиной, учитывая теорию относительности.

Кроме того, после проведения качественного исследования вы не сможете получить финансирование для дальнейших исследований по тому же вопросу. В терминах Байеса такое исследование действует как априорная информация.

Математика безоговорочно верна, но обычно неприменима к реальности, если предположения не выполняются точно, что технически никогда не случается. Поэтому необходимы эксперименты, чтобы убедиться, что принятые в предметной области предположения влияют на выводы лишь незначительно.

В юриспруденции есть критерий, определяющий, что исследование, направленное на поиск доказательств, должно быть выполнено разумным методом, чтобы его вообще можно было рассматривать (см. главу 4. *Основы компьютерного права*). Это, возможно, лучшее интуитивное обоснование приблизительных процедур. То есть если бы то же исследование провел бы другой эксперт, то он должен был бы использовать методы аналогичного качества. Нужно добиться достаточного доверия, чтобы представить результат лицу, принимающему решение. Более подробно эта тема рассмотрена в [21.7].

## 21.21. Сравнение совпадающих пар

*Эксперимент с совпадающими парами* — это парные наблюдения (например, измерение производительности двух алгоритмов на наборе контрольных задач). Требуется узнать, есть ли у одного из них значительное преимущество перед другим. В задачах компьютерного моделирования, когда алгоритм запускается с разными параметрами, используются тесты на сопоставление пар/кортежей.

*Знаковый тест* предполагает, что наблюдения происходят из непрерывного распределения. Но непрерывность — слабое допущение, потому что даже дискретное распределение можно сделать непрерывным, добавив немного PDF для создания связей. Таким образом, лучше сделать *общее предположение о положении*, гласящее, что наблюдения

не следуют какой-либо точной схеме. Концептуально, добавление небольшого количества шума к каждому наблюдению удовлетворит этому предположению во всех случаях, поэтому тест знаков вообще не имеет распределения.

Идея состоит в том, что при одинаковой производительности в  $n$  «играх» распределение «выигравших» является биномиальным  $(n/2, n)$ , и значительное отклонение отвергает это предположение. Ничьи так и считаются ничьими, а для дискретности бинома при нечетном числе ничьих необходимо отбросить одну ничейную пару. Но при аппроксимации биномиального  $(n/2, n) \approx$  нормального  $(n/2, n/4)$  распределения розыгрыша проблем не возникает. Если упростить алгебру, балл  $z$  становится равным (разнице выигрышей)/ $n$ .

```
bool signTestAreEqual(double winCount1, double winCount2, double z = 2)
{ return abs(winCount1 - winCount2)/sqrt(winCount1 + winCount2) <= z; }
```

Учитывайте, что при преобразовании к нормали может иметь значение дискретизация — например, вычисленное значение  $z$  может быть равно 2, что способно существенно повлиять на процент истинно нулевых отказов. При определенном значении  $a$  процедура будет иметь правильную гарантию ошибки I рода только для немного большего  $a$ , но этот приблизительный тест также полезен, потому что произвольность аппроксимации включается в произвольность выбора  $a$ . По заданным данным нужно подчитать выигрыши:

```
pair<double, double> countWins(Vector<pair<double, double> > const& data)
{
    int n1 = 0, n2 = 0;
    for(int i = 0; i < data.getSize(); ++i)
    {
        if(data[i].first < data[i].second) ++n1;
        else if(data[i].first > data[i].second) ++n2;
        else{n1 += 0.5; n2 += 0.5;}
    }
    return make_pair(n1, n2);
}

bool signTestPairs(Vector<pair<double, double> > const& data, double z = 2)
{
    pair<double, double> wins = countWins(data);
    return signTestAreEqual(wins.first, wins.second, z);
}
```

Нормальное приближение знакового теста является асимптотическим и неточным для очень малых  $n$ . Здесь следует использовать биномиальную инверсию с отбрасыванием нечетной связи.

Знаковый тест заключается в том, чтобы делать как можно меньше предположений. Как правило, нельзя делать статистические выводы, не делая предположений, и лучше использовать существующие методы, основанные на предположениях, признавая при этом их ограничения, чем вообще не иметь никакого метода для решения проблемы. ARE для  $t$ -критерия такие же, как и для медианы по отношению к среднему.

Если симметрии нет, знаковый тест учитывает медиану различий, которая не равна среднему значению. Таким образом, при наличии двух распределений с одинаковым средним значением, где одно смещено влево, а другое — вправо, при  $n \rightarrow \infty$  знаковый

тест покажет значительную разницу в медианах. То есть идея о том, что для проверки различия средств предполагается строго меньше нормальности, и поэтому она является наиболее обобщенной, неверна. Но это не имеет значения, потому что вместо нее следует найти доверительный интервал для такой разницы.

В общем, что касается оценок, не существует универсального хорошего теста для любой задачи, даже если это просто сравнение парных альтернатив. Необходимо учитывать свойства данных, связанные с предметной областью, и выбирать тесты соответственно. Можно сравнить метод с его копией, чтобы увидеть возможные отклонения в каких-либо метриках, предназначенных только для тестирования, — это помогает приучить себя не обращать слишком много внимания на шум, который человеческий разум слишком стремится интерпретировать как осмысленный узор.

## 21.22. Множественные сравнения

Работать с несколькими оценками сложно из-за *множественного тестирования/множественного сравнения*. Обычно принимается вывод, основанный на отклонении null, если значение  $p < 0,05$ . Но вывод, основанный на 100 исследованиях, где каждое значение  $p < 0,05$ , не может быть принят при  $\alpha = 0,05$ . Рассмотрим примеры:

- ◆ моделирование 100 нормальных выборок одинакового размера — выборка с наименьшим средним значением занижает истинное среднее значение, а выборка с наибольшим средним значением случайно завышает его. Доверительные интервалы в этом случае не будут иметь обещанного покрытия;
- ◆ отправка советов по инвестициям на покупку/продажу 1024 людям в течение 10 дней, где после каждого дня половина неверного прогноза отбрасывалась. В конце концов, одному человеку покажется, что предсказатель всегда прав. Но оценка вида 10/10 не будет правильно оценивать производительность.

Общая частота ошибок называется *частотой ошибок в семействе* (Family-Wise Error Rate, FWER). Это относится как к доверительным интервалам, так и к  $p$ -значениям. Простой метод внесения поправок для множественного тестирования обеспечивается *неравенством Бонферрони* (также называемым *коррекцией Бонферрони*, *границей объединения* или *неравенством Буля* в различных версиях):

- ◆  $\Pr(\text{event}_0 \cup \dots \cup \text{event}_{k-1}) \leq \sum \Pr(\text{event}_i);$
- ◆  $\Pr(\text{event}_0 \cap \dots \cap \text{event}_{k-1}) \geq 1 - \sum (1 - \Pr(\text{event}_i));$

В частности (см. [21.83]), для  $k$  событий:

- ◆ доверительные интервалы — чтобы получить общую достоверность  $1 - \alpha$ , каждый тест должен иметь  $1 - \alpha/k$  достоверность. Например, для нормального распределения нужно найти соответствующее  $z$ :

```
double find2SidedConfZBonf(int k, double conf = 0.95)
{
    assert(k > 0);
    return find2SidedConfZ(1 - (1 - conf)/k);
}
```

- ◆  $p$ -значения меняются с  $p_i$  на  $kp_i$ .



Имейте в виду, что нормальная модель является аппроксимацией, и ее качество обычно хуже в хвостах, поскольку они очень плоские. Поправка на множественность мало влияет на результаты, повышая чувствительность к выбору нормальной модели. Но метод все еще хорошо работает на практике, возможно потому, что поправка на множественность является консервативной.

У  $p$ -значений ситуация лучше. Если не удастся отклонить какой-либо null с ограниченной коррекцией, нужно меньше коррекции для других. В частности, принцип закрытого тестирования приводит к *процедуре Холма* ([21.83]):

1. Сортировка  $p$ -значений.
2. Скорректированное  $p$ -значение <sub>$i$</sub>  равно:

$$\min(1, \max(\text{скорректированное } p\text{-значение}_{i-1}, p\text{-значение}_i \times (k - i))).$$

```
void HolmAdjust(Vector<double>& pValues)
{
    int k = pValues.getSize();
    Vector<int> indices(k);
    for(int i = 0; i < k; ++i) indices[i] = i;
    IndexComparator<double> c(pValues.getArray());
    quickSort(indices.getArray(), 0, k - 1, c);
    for(int i = 0; i < k; ++i) pValues[indices[i]] = min(1.0, max(i > 0 ?
        pValues[indices[i - 1]] : 0, (k - i) * pValues[indices[i]]));
}
```

Метод строго менее консервативен, чем Бонферрони.

Но управление FWER слишком консервативно при больших  $k$ . В предварительных исследованиях можно значительно сократить количество ложных отклонений, вместо этого контролируя *частоту ложных открытий* (False Discovery Rate, FDR) — долю отклоненных истинных null. Затем для проверки можно использовать независимые данные. Можно даже разделить данные пополам и выполнять проверку в два раунда: первый — с помощью FDR, а второй — с помощью FWER.

*Процедура Бенджамини — Хохберга* контролирует FDR и похожа на процедуру Холма (см. [21.83]):

1. Сортировка  $p$ -значений.
2. Для  $i$  в убывающем порядке скорректированное  $p$ -значение <sub>$i$</sub>  равно:

$$\min(\text{скорректированное } p\text{-значение}_{i+1}, p\text{-значение}_i \times k / (i + 1)).$$

```
void FDRAdjust(Vector<double>& pValues)
{
    int k = pValues.getSize();
    Vector<int> indices(k);
    for(int i = 0; i < k; ++i) indices[i] = i;
    IndexComparator<double> c(pValues.getArray());
    quickSort(indices.getArray(), 0, k - 1, c);
    for(int i = k - 1; i >= 0; --i) pValues[indices[i]] = min(i < k - 1 ?
        pValues[indices[i + 1]] : 1, pValues[indices[i]] * k / (i + 1));
}
```

Алгоритм предполагает отсутствие отрицательной зависимости между гипотезами, которую невозможно проверить на практике, но это не проблема, потому что параметр FDR полезен только для предварительного анализа.

## 21.23. Сравнение совпадающих кортежей

При существовании  $k > 2$  альтернатив обычно требуется выяснить, какие пары альтернатив различны. Простое решение — использовать несколько знаковых тестов с поправкой на множественность. Но лучше применить *тест Неме́ни*. В нем используются *ранги Фридмана* — обобщение результатов знаковых тестов. Сначала преобразуйте наблюдения в ранги в каждом домене  $i$ . Ранги уникальны и распределены непрерывно, но на практике это не всегда так. Нарушения непрерывности обрабатываются путем усреднения равных рангов. Вычислите  $\forall 0 \leq j < k, r_j = \sum_i r_{ij}$  = сумма рангов для области  $j$ :

```
Vector<double> FriedmanRankSums(Vector<Vector<double> > const& a)
{ // a[i] — это вектор ответов домена i
  assert(a.getSize() > 0 && a[0].getSize() > 1);
  int n = a.getSize(), k = a[0].getSize();
  Vector<double> alternativeRankSums(k);
  for(int i = 0; i < n; ++i)
  {
    assert(a[i].getSize() == k);
    Vector<double> ri = convertToRanks(a[i]);
    for(int j = 0; j < k; ++j) alternativeRankSums[j] += ri[j];
  }
  return alternativeRankSums;
}
```

Асимптотически существуют альтернативы  $i$  и  $j$ ,  $\frac{|r_i - r_j|}{\sqrt{nk(k+1)/6}} \sim normal$  (см. [21.28]).

Важно понять, что это соответствует знаковому тесту для  $k = 2$ . В некоторых источниках ошибочно делят на 12 вместо 6). Обратите внимание, что мы используем аналитическую объединенную дисперсию вместо расчетной. Второстепенное предположение состоит в том, что наблюдения исходят из непрерывного распределения (см. [21.28]), но его нарушение приводит к созданию связей. Вычисление дисперсии предназначено для несвязанных наблюдений, но связи могут только уменьшить дисперсию и поэтому не увеличивают частоту ошибок I рода:

```
double NemenyiAllPairsPValueUnadjusted(double r1, double r2, int n, int k)
{ return 1 - approxNormalCDF(abs(r1 - r2)/sqrt(n * k * (k + 1)/6.0)); }
```

Еще один способ использования этой процедуры состоит в том, чтобы вычислить значимую разницу в среднем ранге, необходимую для достижения одобрения с вероятностью 95%. Хотя значения  $p$  здесь не вычисляются, способ работает для любой пары с попарной настройкой. Чтобы увеличить мощность, иногда можно уменьшить количество заранее запланированных сравнений. Типичным случаем является сравнение всех альтернатив с контрольной альтернативой. Затем выполняется коррекция для  $k$  гипотез, предполагая, что все альтернативы сравниваются с контрольной:

```
double findNemenyiSignificantAveRankDiff(int k, int n, bool forControl = false,
double conf = 0.95)
```

```
{ // сумма инвертированных рангов
    int nPairs = k * (k + 1)/2;
    double q = sqrt(nPairs/(3.0 * n)),
        z = find2SidedConfZBonf(forControl ? k : nPairs, conf);
    return q * z/n; // среднее рангов, а не сумма
}
```

Процедура Холма требует вычисления всех  $\frac{k(k-1)}{2}$   $p$ -значений. Результирующая матрица содержит фиктивные единицы на диагонали (для обозначения незначительности):

```
Matrix<double> RankTestAllPairs(Vector<Vector<double> > const& a,
    double NemenyiALevel = 0.05, bool useFDR = false)
{
    Vector<double> rankSums = FriedmanRankSums(a);
    int n = a.getSize(), k = rankSums.getSize();
    Vector<double> temp(k * (k - 1)/2);
    for(int i = 1, index = 0; i < k; ++i) for(int j = 0; j < i; ++j)
        temp[index++] = NemenyiAllPairsPValueUnadjusted(rankSums[i],
            rankSums[j], n, k);
    if(useFDR) FDRAdjust(temp);
    else HolmAdjust(temp);
    Matrix<double> result = Matrix<double>::identity(k);
    for(int i = 1, index = 0; i < k; ++i) for(int j = 0; j < i; ++j)
        result(i, j) = result(j, i) = temp[index++];
    return result;
}
```

Контрольное сравнение здесь не реализовано, и пользователь должен выполнить его вручную с помощью обоснованных  $p$ -значений.

Часто требуется найти только лучший вариант или лучшее подмножество. Для этого нужно просто сравнить вариант с самым низким рангом со всеми остальными, используя скорректированные попарные  $p$ -значения. Несколько более эффективными, но более сложными в реализации, являются множественные сравнения с лучшими процедурами (см. [21.83]).

В идеальном случае мы получим доверительные интервалы по отдельным средним рангам. К сожалению, оценки дисперсии доступны только для разностей. Вместо сравнения совпадающих наблюдений может иметь смысл игнорировать сопоставление и вычислять процент удовлетворительных результатов. Их можно анализировать как несколько независимых выборок для получения доверительных интервалов, поскольку поправка Бонферрони допускает наличие зависимости. Но то, что определяет удовлетворительную производительность, зависит от предметной области.

## 21.24. Сравнение независимых выборок

Типичной непараметрической целью является оценка доверительных интервалов для различий двух квантилей. Для  $k > 2$  простым правильным решением является вычисление доверительных интервалов с поправкой Бонферрони на  $a/k$  и формирование  $\text{diff}_{i,j} = (l_i - u_j, u_i - l_j)$ . Для этого не требуется дополнительных настроек кратности.

Даже для  $k = 2$  сравнение очень неэффективно, и разницу нужно определить напрямую. Но, несмотря на то, что нормальные аппроксимации являются аддитивными, у нас нет способа получить результирующие квантили разности. Эвристическое решение состоит в предположении о том, что доверительные интервалы получены из нормального распределения, вычисления подразумеваемой стандартной ошибки и формирования нормальной разности. Например, для медианы получается следующее:

```
pair<double, double> normal2SampleDiff(double mean1, double stel,
    double mean2, double ste2, double z)
{ // разность полученных из приблизительно нормального распределения
  // доверительных интервалов
  NormalSummary n1(mean1, stel * stel), n2(mean2, ste2 * ste2),
    diff = n1 - n2;
  double ste = diff.stddev() * z;
  return make_pair(diff.mean - ste, diff.mean + ste);
}

pair<double, double> normalConfDiff(double mean1, pair<double, double> const&
    conf1, double mean2, pair<double, double> const& conf2, double z)
{ // разность полученных из приблизительно нормального распределения
  // доверительных интервалов
  return normal2SampleDiff(mean1, (conf1.second - conf1.first)/2/z,
    mean2, (conf2.second - conf2.first)/2/z, z);
}

pair<double, double> median2SampleDiffConf(Vector<double> const& samples1,
    Vector<double> const& samples2, double z = 2)
{
  return normalConfDiff(median(samples1), quantileConf(samples1, 0.5, false,
    z), median(samples2), quantileConf(samples2, 0.5, false, z), z);
}
```

Аппроксимацию можно считать разумной, даже если оба распределения смещены в одном и том же направлении. Точность потеряется только при перекосе одного распределения влево, а другого — вправо. Но асимптотически это правильный тест. Более стандартной процедурой является оценка дисперсии медианы и ее использование для комбинированного доверительного интервала (см. [21.102]). Но это сложнее, чем непосредственная оценка доверительного интервала, т. к. здесь важно, какое именно было выбрано значение. Таким образом, описанная ранее процедура проста и эффективна.

Возможно, наиболее эффективной общей оценкой разницы является разница между усеченными средними. Хотя для асимметричных распределений получают псевдомедианы, для интерпретации разница между любыми мерами близлежащего местоположения псевдомедиана подходит:

```
pair<double, double> trimmedMean2SampleDiffConf(Vector<double> const&
    samples1, Vector<double> const& samples2, double z = 2)
{
  return normal2SampleDiff(trimmedMean(samples1),
    trimmedMeanStandardError(samples1), trimmedMean(samples2),
    trimmedMeanStandardError(samples2), z);
}
```

При использовании средних и больших  $n$  безопасно использовать разницу нормальных CLT. Для меньших  $n$   $t$ -распределение не применяется напрямую, потому что разница не

соответствует  $t$ -распределению. Что касается медиан, простое приближение состоит в том, что отдельные выборочные распределения являются нормальными со стандартными ошибками и поправкой  $\frac{t_{\text{значение}}}{z_{\text{значение}}}$ .

Для этих оценок также работает двухвыборочная начальная загрузка.

## 21.25. Перестановочные тесты

Перестановочные (или *рандомизационные*) тесты обычно не требуют предположений там, где они обычно применяются, но зато им нужны значительные вычислительные ресурсы. Предположим, мы сравниваем две группы на равенство и имеем iid непарных наблюдений. Null-значение состоит в том, что *сравниваемые распределения равны*, тогда как обычные тесты предполагают равенство конкретной статистики. В итоге получаем *взаимозаменяемость* — например, в двух экспериментах при null-значении наблюдения из одной группы могли бы с таким же успехом появиться и в другой. Поскольку конкретное наблюдение появилось там, где оно появилось, это могло произойти случайно, а могло из-за того, что значение null ложно. Чтобы проверить это, надо выбрать статистику и проверить ее значение при любом распределении наблюдений по группам (см. [21.38]). Все возможные присваивания образуют *группу перестановок*, определяющую null. Эта абстрактная формулировка полезна теоретически — в работе [21.52] приведена более подробная информация.

Статистика может быть любой, и обычно выбирают среднее значение или любое другое, подходящее для решаемой задачи значение. Важным вычислительным трюком, как и в случае начальной загрузки, является выборка некоторого числа  $b$  из всех возможных присвоений — например, 10 000, чтобы получить разумную оценочную точность  $p$ -значения (в работе [21.18] предлагается использовать большие числа):

1. Определить взаимозаменяемость и выберите статистику  $f$ .
2.  $b$  раз:
3. Сгенерировать случайное присвоение («перестановку») в пределах возможностей обмена.
4. Вычислить  $f_i = f(\text{присвоение})$ .
5. Подсчитать количество  $f_i \geq$  или  $\leq$  (или оба условия для двусторонних тестов), чем исходное.
6. Из этих подсчетов вычисляется нижняя граница  $p$ -значения.

Подсчеты показывают, насколько экстремальна  $f(\text{выборка})$  в качестве низкого или высокого значения. Поскольку двустороннее тестирование эквивалентно использованию одностороннего тестирования при  $\alpha/2$ , неизвестно, что искать — меньшее, большее, оба сразу или удвоить меньшее. В качестве небольшого трюка можно прибавить 1 к счетчику, чтобы избежать:

- ♦ обманчивых нулевых  $p$ -значений;
- ♦ численных проблем для вычисления доверительных интервалов с помощью числовой инверсии (обсуждается позже).

```

template<typename PERM_TESTER> double permutationTest(PERM_TESTER& p,
    int b = 10000, int side = 0, Random<>& rng = GlobalRNG())
{ // "более экстремальный" означает большее значение для одностороннего
  // теста, где сторона -1 соответствует меньшему, а 1 — большему
  assert(b > 1 && abs(side) <= 1);
  double f = p();
  int nLeft = 1, nRight = 1; // начало с единицы, а не нуля
  for(int i = 0; i < b; ++i)
  {
    p.exchange(rng);
    double fr = p();
    nLeft += (f <= fr);
    nRight += (f >= fr);
  }
  double leftP = nLeft * 1.0/(b + 1), rightP = nRight * 1.0/(b + 1);
  return side == 0 ? min(1.0, 2 * min(leftP, rightP)) :
    side == -1 ? leftP : rightP;
}

```

Определить взаимозаменяемость иногда сложно. Не все можно заменить — например, если есть данные пациентов и назначенное лечение, лечение взаимозаменяемо, а пациенты — нет. В случае парных наблюдений, ранее обработанных знаковым тестом, все присвоения генерируются с учетом замены лечения для пациента. Затем по всем возможным перестановкам рассчитывается распределение различий в средних и проверяется, не слишком ли велика разница в выборке. Это также работает для одного распределения, симметричного относительно 0. Нужно инвертировать знак данных, чтобы сформировать группу перестановок:

```

template<typename LOCATION_F> struct PairedTestLocationPermuter
{
  Vector<double> diffs;
  LOCATION_F f;
  PairedTestLocationPermuter(Vector<double> const& data): diffs(data){}
  PairedTestLocationPermuter(Vector<pair<double, double> > const& data)
  {
    for(int i = 0; i < data.getSize(); ++i)
      diffs.append(data[i].first - data[i].second);
  }
  double operator() ()const{return f(diffs);}
  void exchange(Random<>& r)
  {for(int i = 0; i < diffs.getSize(); ++i)if(r.mod(2))diffs[i] *= -1;}
  void setShift(double shift) // не должен вызываться после обмена
  {for(int i = 0; i < diffs.getSize(); ++i) diffs[i] += shift;}
};

```

Для медианы и усеченного среднего гипотеза перестановки также состоит в том, что используется симметричное распределение. Теоретически (см. [21.63]) тесты перестановок для парных данных, которые проверяют меру местоположения, таковы:

- ◆ точны, когда разностное распределение симметрично;
- ◆ асимптотически точны, когда распределение асимметрично, но с конечной дисперсией.



```
{
  double stat = p();
  permConfHelper<PERM_TESTER> f(p, a, time(0), b);
  double left = exponentialSearch1Sided(f, -stat, -0.001).first,
    right = exponentialSearch1Sided(f, -stat).first;
  left = isinfinite(left) ? left : -stat;
  right = isinfinite(right) ? right : -stat;
  return make_pair(2 * stat + left, 2 * stat + right);
}
```

Например, для показателей местоположения получаем (рис. 21.7–21.9).

Все это свидетельствует о том, что методы перестановки никогда не превосходят индивидуально настроенное решение, несмотря на меньшее количество предположений. В некоторых случаях — таких как при вычислении медианы выборки Леви и средних значений всех усеченных выборок — значение *a* резко превышает.

Распределение и статистика	T				Нормальное			
	aL	aR	%L	ACL	aL	aR	%L	ACL
Нормальное среднее	0.025	0.025	0.034	0.231	0.029	0.030	-0.009	0.243
Лог–нормальное среднее	0.002	0.113	-0.065	0.261	0.004	0.123	-0.104	0.268
2-нормальное среднее значение смеси	0.023	0.024	0.031	0.224	0.028	0.028	-0.012	0.236
Средние ранги	1.667		3.000	1.667	4.000		1.333	3.333
Начальная загрузка –t с ограничением								
aL	aR	%L	ACL	aL	aR	%L	ACL	
0.021	0.022	0.071	0.223	0.025	0.025	0.034	0.232	
0.011	0.057	0.200	0.337	0.003	0.121	-0.117	0.250	
0.027	0.025	0.072	0.244	0.024	0.024	0.028	0.225	
1.667		4.000	3.000	2.667		1.667	2.000	

Рис. 21.7. Характеристики нескольких доверительных интервалов для среднего значения при n = 30. Те же настройки, что и для моделирования начальной загрузки, за исключением повторения 10 000 раз

Распределение и статистика	Нормальное				Перестановка				Начальная загрузка –t с ограничением			
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL
Нормальное среднее	0.020	0.018	0.082	0.211	0.066	0.065	-0.015	0.362	0.038	0.036	0.210	0.330
Медиана Леви	0.023	0.020	0.411	0.285	0.001	0.233	1.835	0.281	0.024	0.026	0.633	0.371
2-нормальное среднее значение смеси	0.022	0.020	0.274	0.261	0.016	0.006	0.470	0.215	0.017	0.016	0.184	0.214
Средние ранги	1.667		1.667	2.000	2.333		2.333	2.000	2.000		2.000	2.000

Рис. 21.8. Характеристики нескольких доверительных интервалов для медианы

Распределение и статистика	Нормальное				Перестановка				Начальная загрузка –t с ограничением			
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL
Нормальное обрезанное среднее	0.037	0.034	-0.006	0.267	0.039	0.037	0.015	0.281	0.025	0.022	0.097	0.237
Лог–нормальное обрезанное среднее	0.015	0.062	-0.025	0.265	0.006	0.127	-0.099	0.249	0.017	0.028	0.149	0.248
Обрезанная медиана Леви	0.003	0.126	-0.136	0.286	0.000	0.258	-0.455	0.190	0.002	0.023	12.119	2.321
2-нормальное обрезанное среднее значение смеси	0.024	0.038	-0.028	0.242	0.013	0.024	0.025	0.193	0.018	0.032	0.058	0.237
Средние ранги	2.000		1.000	3.000	3.000		1.000	1.000	1.000		3.000	1.000

Рис. 21.9. Характеристики некоторых доверительных интервалов для выборочного усеченного среднего. Начальная загрузка ненадежна, в то время как перестановочное тестирование, по-видимому, наследует надежность любой используемой статистики



При вычислении корреляции группа перестановок формируется всеми перестановками значений  $x$  и  $y$ , К сожалению, для инвертирования неясно, как сдвигать данные в ответ на сдвиг, особенно потому, что это может нарушить ограничения диапазона (рис. 21.10).

Распределение и статистика	Нормальное-Z				Начальная загрузка -t с ограничением			
	aL	aR	%L	ACL	aL	aR	%L	ACL
Нормальная ошибка (почти линейная)	0.017	0.006	0.267	0.203	0.027	0.016	0.250	0.273
Нормальная ошибка кубическая	0.002	0.000	1.083	0.087	0.019	0.046	0.189	0.281
Средние ранги	1.000		2.000	1.000	2.000		1.000	2.000

Рис. 21.10. Характеристики нескольких доверительных интервалов для корреляции Пирсона

Заменяемость не так полезна, как независимость и случайность данных, но и ее хватает. Помните, что при null-значении одинаковы все свойства, а не только среднее значение. В правильном эксперименте со случайными назначениями никакое различие не может быть правильно принято за null-значение, но вы не сможете обнаружить различие в чем-либо, кроме проверенной статистики. Надлежащая рандомизация означает, например, отсутствие предвзятости данных в отношении теста — т. е. когда испытуемая группа уже может быть предрасположена к определенному результату.

В независимых выборках ситуация иная — например, если сравнить два распределения с разными дисперсиями, но с одинаковыми средними значениями, они не будут обладать полной взаимозаменяемостью. Ситуация аналогична тесту Уилкоксона — Манна — Уитни (дополнительная информация имеется в работе [21.28]), который работает с рангами, но требует симметрии, чтобы все возможные присвоения рангов были равновероятными. У двух независимых выборок средняя разница асимптотически точна тогда и только тогда, когда отношение размера выборки  $\rightarrow 1$  или конечные дисперсии равны (см. [21.63]). Для медианы или усеченного среднего различия становятся еще меньше. Таким образом, инверсия критерия перестановки — плохой вариант для доверительных интервалов на различиях независимых выборок. Тем не менее по выводам, приведенным в работе [21.18], перестановки дают хорошие результаты для малых  $n$ , где отклонения от предположений обнаружить нельзя.

Перестановки, безусловно, полезны как способ расчета точных распределений для непараметрических статистик — таких как распределение для теста Неменьи, особенно со связями, а также для соответствующих доверительных интервалов. Но для стандартных случаев — таких как проверки совпадения (или несовпадения, хотя в этом случае доверительные интервалы получают напрямую), лучше всего применять непараметрические тесты, которые являются более надежными. Впрочем, говорят, что в программном обеспечении StatXact используются перестановочные тесты. В других случаях, как и в случае с начальной загрузкой, сначала нужно провести исследование, посвященное конкретной проблеме.

## 21.26. Сравнение нескольких альтернатив в нескольких предметных областях

Поскольку несколько предметных областей можно моделировать с помощью согласованных выборок, вы можете выполнить преобразование оценок в ранги и использовать

тест Неменьи. Тест предполагает, что все предметные области одинаково ценны, и объединяет показатели с использованием усредненных рангов. Но существуют и другие способы сделать это. Основная проблема заключается в том, что любой тест может иметь статистическую значимость без практического значения. То есть вам нужен оценщик производительности для каждой альтернативы. В целях статистической значимости необходимо предположить, что домены для набора предметных областей выбираются случайным образом.

Обычно нет смысла выполнять агрегацию по всем доменам, используя такие функции, как среднее. Например, даже если для любой предметной области и альтернативы показатели производительности распределены нормально, средний вес областей большего масштаба тоже больше. Нужно преобразовывать оценки, специфичные для предметной области, чтобы их усреднение было более значимым. Рассмотрим часто используемые преобразования:

- ◆ преобразование в ранги — то, что делает тест Неменьи, не получая при этом полезных оценок. Например, если есть 10 альтернатив с похожим качеством, их очевидные «ранговые оценки»  $1 - \frac{\text{ранг} - 1}{10}$  равны 1, 0,9, 0,8, 0,7 и т. д.;
- ◆ отображение с диапазона [мин, макс] на [0, 1]. Это позволяет сохранить информацию лучше, чем ранги, но влечет проблемы, когда показатели производительности очень похожи. Например, учащиеся с оценками 99 и 98 получают скорректированные оценки 100 и 0;
- ◆ для процентных оценок и случаев, когда большие значения лучше, выполняется отображение [0, max] на [0, 1]. Это позволяет избежать увеличения разницы между близкими оценками;
- ◆ когда значение min лучше, нужно отобразить абсолютные оценки на относительные, разделив их на эффективность базовой альтернативы, которая обычно является наименее эффективной. Например, это полезно для сравнения времени выполнения.

Чтобы изучить значимость других параметров, помимо рангов, можно выполнить начальную загрузку в месте, где выполняется ресемплинг. Но для начальной загрузки у вас должно быть  $\geq 10$  доменов, чтобы получить точные результаты.

Когда сами оценки представляют собой агрегаты с известными распределениями, возникают потери мощности. Но использовать эту информацию сложно, потому что преобразования вроде минимакса оцениваются на основе данных. Требуется сделать предположение о том, что баллы могут быть агрегированы по предметным областям. Таким образом, это имеет смысл только в том случае, если для средневзвешенного значения используются независимое от данных масштабирование и веса.

Составление оценок с учетом предметной области заключается в преобразовании баллов в двоичную форму для определения того, является ли производительность приемлемой. Затем можно напрямую взять средние значения и перейти к решению задачи независимых сравнений. На практике именно так проводят сравнения и выбирают методы с низкой вероятностью катастрофической производительности, даже если они в среднем немного хуже, чем остальные. К сожалению, анализ рангов Фридмана для этой цели бесполезен и может служить только в роли дополнительной информации.

Среднее значение — не единственный способ агрегирования, но все агрегаты связаны с *теоремой Эрроу о невозможности* (см. [21.85]): при наличии трех или более альтернатив и назначенных для них некоторыми избирателями рангов приведенные далее условия не могут существовать одновременно:

- ◆ отсутствие диктатуры — ни один избиратель не контролирует результат;
- ◆ единодушие — если все избиратели предпочитают вариант А, а не Б, то же самое делает и группа;
- ◆ *независимость от нерелевантных альтернатив* (Independence of Irrelevant Alternatives, ИА) — предпочтение группы А по сравнению с Б должно зависеть только от предпочтений избирателей из двух альтернатив, а не от других.

Большинство систем комбинаций рангов жертвуют свойством ИА, но для статистики это не обязательно плохо. Стоит предпочесть альтернативу, которая лучше работает на большем количестве предметных областей (т. е. с большим количеством избирателей), и альтернативу, которая на некоторых доменах работает неплохо. Таким образом, добавление неконкурентной альтернативы может дать другой лучший результат, потому что существующий лучший вариант хуже, чем этот вариант для некоторой предметной области. Например, при наличии двух хороших студентов, один из которых получил пятерку на одном экзамене, добавление большего числа троечников понизит рейтинг неудачливого студента на этом тесте, а другой за счет этого сможет выиграть по среднему баллу.

Особым случаем ИА является *критерий Кондорсе*: большинство не может выбрать другого кандидата, кроме победителя. Например, на выборах в США в 2000 году Гор победил бы без Надера. Победителя, по Кондорсе, при транзитивных предпочтениях не существует, но на практике такие ситуации редки. Аналогичные проблемы относятся и к методам криволинейной оценки, потому что ИА может дать преимущество альтернативам, которые превосходно справляются с задачами, с которыми большинство справляется лишь хорошо.

Существует интересный метод под названием *голосование за одобрение*: человек отмечает каждого кандидата как одобренного или нет, и побеждает тот, у кого больше всего одобрений (см. [21.100]). На выборах эта стратегия используется довольно часто:

- ◆ избиратели не штрафуются за выражение своих предпочтений — т. е. можно одобрить и Надера, и Гора;
- ◆ теорема Эрроу неприменима к схеме оценок, но применима к схеме ранжирования, поскольку технически избиратель имеет более одного голоса, несмотря на очевидную справедливость;
- ◆ избиратели автоматически отдают предпочтение более центристским альтернативам.

Получается что-то вроде голосования за то, что более модно, а не за медиану или среднее значение. Каждый «избиратель» фактически создает  $\epsilon$ -диапазон вокруг желаемого выбора, и выбирается область максимального охвата. Это очень похоже на null-значение с наибольшей вероятностью в статистическом тестировании. Можно рассмотреть тот же вопрос со стороны голосования за неодобрение, т. е. наименее одобренные альтернативы будут исключены. Метод может показаться полезным для выбора альтернатив, которые никогда не бывают слишком плохими, но на самом деле он, скорее,

направлен на то, чтобы избежать альтернатив, которые никогда не бывают непопулярны у большинства. Ситуация аналогична наказанию за сколь угодно плохую производительность при небольшой группе избирателей.

Вы можете сделать турнирный отбор, если альтернативы образуют некоторые логические подгруппы. Разделите для этого предметные области пополам или около того. Затем в первой половине рассматриваются подгруппы альтернатив, а во второй половине — победители и те, кто не значительно хуже. Одним из преимуществ метода является то, что можно выполнить корректировку множественности FDR, и если плохих альтернатив много, немалая часть из них выпадет в первом раунде.

Иногда требуется только знать, дает ли выбор другого решения (если текущее решение реализовано и развернуто) достаточно превосходящий результат для переключения на него. Стоит помнить, что плохие альтернативы могут улучшить результаты, если проблема слишком проста. Чтобы сравнение было значимым, необходимо, чтобы базовый экспертный метод превосходил базовый плохой метод.

Еще одной возможной задачей является поиск альтернативы с определенным свойством (например, простой), производительность которой ненамного хуже, чем у лучшей альтернативы или набора альтернатив. В этом случае применяется *правило одной стандартной ошибки* — выбрать альтернативу с желаемым свойством, производительность которой  $\geq$  производительности лучшей модели за вычетом стандартной ошибки оценки ее производительности. Также можно выбрать альтернативу, производительность которой отстает от лучшей, скажем, на 5%, но обладающую желаемым свойством, таким как простота. Можно добавить дополнительные штрафы, чтобы сделать выбор такой альтернативы оправданным.

## 21.27. Работа с данными расчета

Часто модель рассчитывается с использованием биномиального  $(p, n)$  распределения ( $p$  здесь не следует путать с  $p$ -значением), которое является нормальным  $\left(p, \frac{p(1-p)}{n}\right)$

с небольшой потерей точности, если  $p$  не близко к 0 или 1. В противном случае  $p$  оценивается очень неточно. Например, чтобы проверить отличие  $p$  от 0 при определенном

$z$ -показателе, равенство null отклоняется, если  $\bar{x} \geq \frac{1}{1 + n/z^2}$ , а это бывает довольно час-

то. Например, после просмотра  $n$  образцов «да»  $\Pr(\text{«нет»})$  равна нулю по нормальному приближению.

При оценке  $p$  интервал Вальда работает плохо. Лучший интервал получается путем инвертирования нормального теста, который использует нулевую дисперсию  $p(1-p)$  вместо  $s^2 = \bar{x}(1 - \bar{x})$ . Знаковый тест делает то же самое, но в другом контексте. Логика заключается в том, что при точной проверке используется биномиальное правдоподобие со значением  $p$ , которое имеет нулевую дисперсию. Это соответствует принятой в математической статистике конструкции *балльного теста*. Решение уравне-

ния приводит к *интервалу баллов Уилсона*:  $\left(\frac{p + t/2 - \Delta}{1 + t}, \frac{p + t/2 + \Delta}{1 + t}\right)$ , где  $t = \frac{z^2}{n}$  и

$\Delta = z \sqrt{\frac{\bar{x}(1-\bar{x}) + t/4}{n}}$  (см. [21.99]). Интервал асимметричен, когда  $m \neq 0,5$ , но симметричен относительно байесовской оценки, что для  $z = 2$  эквивалентно добавлению двух истинных и двух ложных наблюдений:

```
pair<double, double> wilsonScoreInterval(double p, int n, double z = 2)
{
    assert(p >= 0 && p <= 1 && n > 0 && z >= 0);
    double temp = z * z/n, delta = z * sqrt((p * (1 - p) + temp/4)/n);
    return make_pair((p + temp/2 - delta)/(1 + temp),
        (p + temp/2 + delta)/(1 + temp));
}
```

Этот интервал не дает строгого контроля над значением  $a$ . Но на практике его эффективность настолько хороша, что считается, будто она даже лучше, чем точные интервалы, основанные на инвертировании бинома, потому что в нем не выбирается наибольшее доступное значение  $a \geq$  запрошенного (см. [21.12]). Для  $p$ -значений этот аргумент не работает, но тест на баллы по-прежнему полезен при наличии связей и вычислений. Интервал оценки Уилсона также позволяет обрабатывать ничьи из парных сравнений, допуская половинный подсчет (рис. 21.11). Этот метод используют, например, для оценки эффективности классификации (см. главу 26. *Машинное обучение: классификация*).

Распределение и статистика	Уилсон				Нормальное			
	aL	aR	%L	ACL	aL	aR	%L	ACL
Бернулли05, среднее	0.021	0.021	0.014	0.211	0.021	0.021	0.073	0.224
Бернулли01, среднее	0.026	0.000	-0.059	0.062	0.008	0.184	-0.122	0.332
Бернулли001, среднее	0.036	0.000	0.969	0.034	0.0000	0.740	-0.461	0.464
Средние ранги	3.000		1.667	3.000	3.667		1.333	4.000
Хэф_Ори	Хёффдинг							
	aL	aR	%L	ACL	aL	aR	%L	ACL
	0.003	0.003	0.378	0.100	0.003	0.003	0.487	0.108
	0.002	0.000	0.213	0.023	0.0000	0.000	0.490	0.006
	0.003	0.000	1.093	0.012	0.000	0.000	2.869	0.000
	1.667	3.000	1.667	1.000	4.000	1.333		

**Рис. 21.11.** Сравнение производительности выборок Бернулли с различными параметрами. Оценка Уилсона не теряет точности даже при 0,01, в отличие от нормального распределения. Методы конечной выборки дают гораздо более длинные интервалы

Обратите внимание, что нельзя комбинировать интервалы исходя из того, какой из них короче, и вообще использовать любые другие критерии, основанные на данных, поскольку это приводит к многократному тестированию.

Проблема с нулевым значением, которая является частным случаем, решается с помощью *правила трех* (см. [21.87]). Если бы наблюдались только события типа «да»,  $p$  должно быть небольшим.  $\Pr(n \text{ событий «да»}) = (1 - p)^n$ . Для достоверности 95% требуется  $(1 - p)^n \geq 0,05$ . Учитывая, что  $\lg(0,05) \approx 3$  и что  $\lg(1 - p) \approx p$  для очень малого  $p$ , получаем  $p \leq 3/n$ . То есть при доверительной вероятности 95%  $p \in [0, 3/n]$ . Хотя доля событий «нет» уменьшается с ростом  $n$  при фиксированной достоверности, исключить

их существование нельзя. С точки зрения разработки программного обеспечения тестирование не доказывает отсутствие вариантов использования их с ошибками, но снижает вероятность их обнаружения.

Существует похожая *задача о черном лебедь*: найти  $\Pr(\exists \text{ черного лебедя, увидев } n \text{ белых})$ . По сути, событие «черный лебедь» невозможно предсказать, потому что данные не точны (черные лебеди в конце концов были обнаружены в Австралии, а на других континентах они не живут). Суть логики «черного лебедя» состоит в том, что модель может иметь неизвестные переменные. Нам следует принимать явления такими, какие они есть, и несколько экстремальных непредсказуемых событий, как правило, позволяют объяснить, почему они именно такие. Например, рассмотрим сумму выборок Коши — большая часть ее значения обычно приписывается нескольким выборкам. То же самое и с фондовым рынком — большая часть изменений стоимости происходит в несколько очень хороших и очень плохих дней и из-за небольшого количества великих компаний среди множества средних. Здесь имеет место и толстохвостое распределение, и отсутствие независимости и случайности в данных. Общая стратегия противодействия этому состоит в том, чтобы избегать оценки вероятностей — т. е. агностически не класть все яйца в одну корзину. Тогда вы получите базовую безопасность, работая непосредственно в пространстве стратегии.

## 21.28. Тестирование различий в распределении

Чтобы проверить соответствие выборки дискретному распределению, используйте *критерий хи-квадрат*. Для ограниченного распределения с  $n$  независимыми выборками

статистика  $\sum_{mean_i > 0} \frac{(count_i - mean_i)^2}{mean_i}$  приблизительно равна критерию хи-квадрат

с  $n$  степеней свободы (см. [21.96]). Аппроксимация считается точной, если все средние  $> 5$ . Если существуют зависимости между перекрывающимися областями выборок, степени свободы изменяются. Например, степень свободы равна  $n - 1$ , если сумма средних равна константе.

Тест должен оценивать CDF хи-квадрат, но точная оценка выполняется медленно и приходится обрабатывать много случаев (см. [21.76]), поэтому оценивайте приблизительно. Хи-квадрат распределен нормально, а преобразование улучшает аппроксимацию (см. [21.13]) — если:

$$X \sim \text{ChiCDF}(n), x = \left(\frac{X}{n}\right)^{1/6} - \frac{1}{2}\left(\frac{X}{n}\right)^{1/3} + \frac{1}{3}\left(\frac{X}{n}\right)^{1/2} \sim \text{normal}\left(\frac{x - \mu}{\sigma}\right),$$

где  $\mu$  и  $\sigma^2$  равны  $\sum \frac{a_i}{x^i}$  с соответствующими:

$$a_i = \left\{ \frac{5}{6}, \frac{1}{9}, \frac{7}{648}, \frac{25}{2187} \right\} \text{ и } \left\{ 0, \frac{1}{18}, \frac{1}{162}, \frac{37}{11664} \right\}.$$

Ошибка в худшем случае равна  $O(10^{-2})$ , а типичное значение  $O(10^{-5})$ . Точность растет с увеличением  $n$ :

```

double evaluateChiSquaredCdf(double chi, int n)
{
    assert(chi >= 0 && n > 0);
    double m = 5.0/6 - 1.0/9/n - 7.0/648/n/n + 25.0/2187/n/n/n,
           q2 = 1.0/18/n + 1.0/162/n/n - 37.0/11664/n/n/n, temp = chi/n,
           x = pow(temp, 1.0/6) - pow(temp, 1.0/3)/2 + pow(temp, 1.0/2)/3;
    return approxNormalCDF((x - m)/sqrt(q2));
}

double chiSquaredP(Vector<int> const& counts,
    Vector<double> const& means, int degreesOfFreedomRemoved = 0)
{
    double chiStat = 0;
    for(int i = 0; i < counts.getSize(); ++i)
    { // для лучшей аппроксимации сравниваем каждую выборку с 5
        assert(means[i] >= 5);
        chiStat += (counts[i] - means[i]) * (counts[i] - means[i])/means[i];
    }
    return evaluateChiSquaredCdf(chiStat,
        counts.getSize() - degreesOfFreedomRemoved);
}

```

Тест также применим к  $D > 1$ . Нужно только определить ячейки и рассчитать ожидаемое количество.

## 21.29. Сравнение данных с распределением

Используя DKW, можно сравнить выборку с полностью заданным непрерывным распределением  $T$ , используя эмпирическое распределение  $F$ . Сначала нужно вычислить статистику  $\max_x(|T(x) - F(x)|)$ . Максимум возникает либо в точке данных, либо перед следующей точкой в порядке сортировки. Ввиду непрерывности  $\max_x(|T(x) - F(x)|) = \max_x(|T(x_i) - F(x_i)|, |T(x_i) - F(x_{i+1})|)$ . Итак, отсортируйте каждый образец и выполните сканирование, подобное сортировке слиянием, для точек событий:

```

template<typename CDF> double findMaxKDiff(Vector<double> x, CDF const& cdf)
{ // хелпер для вычисления максимальной разности
    quickSort(x.getArray(), x.getSize());
    double level = 0, maxDiff = 0, del = 1.0/x.getSize();
    for(int i = 0; i < x.getSize(); ++i)
    {
        double cdfValue = cdf(x[i]);
        maxDiff = max(maxDiff, abs(cdfValue - level));
        level += del;
        while(i + 1 < x.getSize() && x[i] == x[i + 1])
        {
            level += del;
            ++i;
        }
        maxDiff = max(maxDiff, abs(cdfValue - level));
    }
    return maxDiff;
}

```

```
template<typename CDF> double DKWPValue(Vector<double> const& x,
    CDF const& cdf)
{ // DKW не используется для p < 0.5
    double delta = findMaxKDiff(x, cdf);
    return min(0.5, 2 * exp(-2 * x.getSize() * delta * delta));
}
```

Асимптотически ([21.28]) для скорректированной статистики  $d = \frac{\max_x (|T(x) - F(x)|)}{1/\sqrt{n}}$ ,

$p$ -значение  $= 2 \sum_{1 \leq i \leq \infty} (-1)^{i-1} e^{-2i^2 d^2}$  (см. [21.28]) в итоге получает *критерий Колмогорова* для одной выборки. Приведенный ранее *критерий DKW* идентичен при использовании только первого члена ряда, но имеет конечную выборку. Непрерывность для достоверности не нужна, но тест может утратить часть мощности, если непрерывности нет.

Полная информация об интегральной функции распределения имеется редко, за исключением особых случаев — таких как сравнение случайно сгенерированных чисел с равномерным распределением (0, 1). Оценить ее по данным и выполнить сравнение — это не то же самое, что заранее знать распределение. Но метод годится для больших выборок (более 200 значений). Таким образом, хорошего общего расширения критерия DKW для сопоставления выборки с семейством распределения не существует, за исключением специального сопоставления формы с диапазоном DKW. В некоторых ситуациях — например, в случае с нормальным распределением, можно центрировать и масштабировать данные, что ограничивает ошибку оценивания.

Приблизительное знание распределения тоже часто полезно. Но тест — не лучший способ показать это, потому что он ищет точное совпадение. Распределение может быть хорошей моделью для данных из определенной области, но маловероятно, что оно будет совпадать в каком-то десятичном разряде, который в конечном итоге обнаружит проверка значимости. Возможно, лучший способ проверить это — выбрать допуск DKW и проверить, не превышает ли он. Видимо, лучшим способом является параметрическая начальная загрузка — выборка как из данных, так и из предполагаемого распределения. В этом случае можно посмотреть, достаточно ли отличаются результаты для желаемого приложения. Например, для загрязненного нормального распределения расстояние Колмогорова может быть небольшим по сравнению со стандартной нормой, но разница в дисперсии может быть очень большой (см. [21.102]).

Еще одно применение критерия DKW — проверка распределения на нормальность. Здесь наиболее действенным с практической точки зрения является *критерий Шапиро — Уилкса*, основанный на корреляции между данными и квантилями нормального распределения. Критические значения получают путем моделирования. Используйте критерий Шапиро-Уилкса, пока  $n$  не станет достаточно большим и табличные значения не превысятся, а затем переключитесь на *критерий DKW*. Подробности, если вам интересно, можно найти в работе [21.28].

Обычно предлагаемая стратегия проверки на нормальность и использование параметрических или непараметрических тестов в зависимости от результата ошибочна, поскольку комбинированная процедура имеет не такую мощность и ошибку первого рода, как отдельные процедуры. Для малых  $n$  обычно отвергнуть нормальность не удастся, а для больших  $n$  непараметрические тесты имеют достаточную мощность и являются



более надежными. Поэтому никогда не следует использовать тестирование распределения для выбора процедуры.

В более общем случае методов для проверки произвольных предположений не существует. В лучшем случае можно провести некоторый прогнозный анализ, основанный на экспериментах с выбранной моделью, и сделать вывод о том, что нарушения незначительны или модель имеет ограниченную чувствительность к ним. Исследовательский анализ зарезервированной части данных может помочь определить степень отклонения, но количественную меру получить не удастся (например, погуглите `qq plot for normality`). В байесовской логике любое конкретное предположение фактически почти наверняка ложно.

## 21.30. Сравнение распределений двух выборок

Для двух выборок размера  $m$  и  $n$  простое концептуальное решение состоит в том, чтобы оценить EDF  $F_1$  и  $F_2$  для обеих выборок, вычислить их доверительные интервалы для истинных распределений с помощью DKW и проверить, не пересекаются ли они между собой.

Но прямое сравнение обладает большей мощностью. Вычислите *статистику Колмогорова — Смирнова* (KS statistic)  $\max_x (|F_1(x) - F_2(x)|)$ , отсортировав каждую выборку и выполнив сканирование, подобное сортировке слиянием, для точек событий. Чисто случайно каждая EDF может оказаться выше другой. Если распределение биномиальное,

ожидаемое отклонение равно  $O(1/\sqrt{n})$ . Для статистики KS получаем  $\sqrt{\frac{m+n}{mn}}$ . Это приводит к скорректированной статистике:

$$d = \max_x \left( \frac{|F_1(x) - F_2(x)|}{\sqrt{(m+n)/(mn)}} \right).$$

Асимптотически (см. [21.28]), ее  $p$ -значение  $= 2 \sum_{1 \leq i \leq \infty} (-1)^{i-1} e^{-2i^2 d^2}$ . Консервативности ради можно использовать только первый член последовательности ( $\approx 95\%$  относительного значения), но даже тогда критерий будет точен, кроме случая  $n = m \geq 458$  (см. [21.80]):

```
double findMaxKSDiff(Vector<double> a, Vector<double> b)
{ // хелпер для вычисления максимальной разницы
  quickSort(a.getArray(), a.getSize());
  quickSort(b.getArray(), b.getSize());
  double aLevel = 0, bLevel = 0, maxDiff = 0, delA = 1.0/a.getSize(),
    delB = 1.0/b.getSize();
  for(int i = 0, j = 0; i < a.getSize() || j < b.getSize(); )
  {
    double x, nextX = numeric_limits<double>::infinity();
    bool useB = i >= a.getSize() || (j < b.getSize() && b[j] < a[i]);
    if(useB)
    {
      x = b[j++];
      bLevel += delB;
    }
```

```

    else
    {
        aLevel += delA;
        x = a[i++];
    } // обработка равных значений — они обрабатываются до обновления разницы
    if(i < a.getSize() || j < b.getSize())
    {
        useB = i >= a.getSize() || (j < b.getSize() && b[j] < a[i]);
        nextX = useB ? b[j] : a[i];
    }
    if(x != nextX) maxDiff = max(maxDiff, abs(aLevel - bLevel));
}
return maxDiff;
}

double KS2SamplePValue(Vector<double> const& a, Vector<double> const& b)
{ // вычисление коррекции, затем поиск p-значения для d
    double stddev = sqrt(1.0 * (a.getSize() + b.getSize()) /
        (a.getSize() * b.getSize())),
        delta = findMaxKSDiff(a, b) / stddev;
    return 2 * exp(-2 * delta * delta);
}

```

Чтобы алгоритм работал корректно с конечной выборкой, можно рассчитать связанное расстояние DKW для каждой полосы, используя одну и ту же статистику, но она имеет гораздо меньшую мощность из-за меньшего ожидаемого отклонения  $d$ .

## 21.31. Анализ чувствительности

Пусть есть функция  $y = f(x)$ , и нам нужно узнать влияние  $x$  на  $y$ . Обычно для вычисления производных используются конечные разности (см. главу 23. *Численные алгоритмы: работа с функциями*), но это проблематично, если функция  $f$  зашумлена. И интерпретировать производные, которые различаются в разных областях функции, сложно. Существует проблема, связанная с *парадоксом Симпсона* (см. главу 27. *Машинное обучение: регрессия*) — переменная, пропущенная при выборе модели, может быть сильно коррелирована с другой переменной, которая есть в модели, и производная от модели не будет отражать влияние первой переменной. Кроме того, в отличие от большинства статистических данных, анализ чувствительности изучает оценщиков, а не оценки.

В качестве альтернативы можно использовать глобальный анализ чувствительности (см. [21.65]). Особенно полезен подход, основанный на вычислении позднего *индекса*

*чувствительности Соболя*. Для любой входной переменной  $x_i$ ,  $S_i = \frac{\text{Var}(E[y|x_i])}{\text{Var}(y)}$  (не

путайте  $S_i$  со стандартным отклонением). Чтобы вычислить числитель, для любого значения  $x_i$  нужно рассчитать его ожидание по отношению ко всем другим переменным, а затем дисперсию ожиданий. Значение  $S_i$  имеет смысл, только если  $\text{Var}(y)$  ограничено — нужен ограниченный диапазон для  $x$  или распределение с плоскими хвостами. Тогда имеем разложение  $\text{Var}(y) = \sum \text{Var}(E[y|x_i]) + \sum \text{Var}(E[y|x_i, x_j]) + \text{взаимодействия более высокого порядка}$ . Первый член соответствует *основным эффектам*.

Простой способ вычислить  $S_i$  из  $n$  оценок  $f$  состоит в том, чтобы использовать срезы, подобные гистограмме: для любого  $x_i$  сортировать значения, разбивать их, возможно, на  $\sqrt{n}$  интервалов, вычислять математическое ожидание по каждому интервалу, а затем вычислять дисперсию. Она согласована (из аргументов  $k$ -NN — см. главу 26. *Машинное обучение: классификация*), но вносит небольшую погрешность. Алгоритм Салтелли (см. [21.65]) более эффективен и дает лучшую точность:

1. Имея  $2n$   $D$ -мерных выборок  $x$ , разбить их на «матрицы»  $XA$  и  $XB$  по  $n$  примеров в каждой.
2.  $\forall 0 \leq j \leq D$  создать  $XC_j = XB$ , но с  $j$ -й переменной во всех примерах, взятых из  $XA$ .
3. Вычислить  $f$  в строках  $XA$  и  $XC$ , чтобы сформировать векторы размера  $n$   $YA$  и  $YC$ .
4. 
$$S_j = \frac{YAYC_j / n - Ave(YA)^2}{YAYA / n - Ave(YA)^2}.$$

Используйте начальную загрузку на  $YA$  и  $YC$ , чтобы вычислить смешанные доверительные интервалы с поправкой на множественность. Они полезны для проверки стабильности вычислений (например, имеют нестабильность, если  $Var(y)$  неограничена и т. д.):

```
Vector<double> findSobolIndicesHelper(Vector<double> const& ya,
    Vector<Vector<double> > const& yc)
{ // вычисление S из YA и YC
    int n = ya.getSize(), D = yc.getSize();
    Vector<double> result(D);
    IncrementalStatistics s;
    for(int i = 0; i < n; ++i) s.addValue(ya[i]);
    double f02 = s.getMean() * s.getMean(), tempa = dotProduct(ya, ya)/n;
    for(int j = 0; j < D; ++j)
        result[j] = max(0.0, (dotProduct(ya, yc[j])/n - f02)/(tempa - f02));
    return result;
}

template<typename FUNCTOR> pair<Vector<double>, Vector<pair<double, double>>>
    findSobolIndicesSaltelli(Vector<pair<Vector<double>, double> > const&
    data, FUNCTOR const& f, int nBoots = 200, double a = 0.05)
{ // вычисление ya и yb
    int D = data[0].first.getSize(), n = data.getSize()/2;
    Vector<double> ya(n), yb(n), yaR(n);
    for(int i = 0; i < 2 * n; ++i)
        if(i < n) ya[i] = data[i].second;
        else yb[i - n] = data[i].second;
    // вычисление yc
    Vector<Vector<double> > yc(D, Vector<double>(n)), ycR = yc;
    for(int j = 0; j < D; ++j)
        for(int i = 0; i < n; ++i)
        {
            Vector<double> x = data[n + i].first;
            x[j] = data[i].first[j];
            yc[j][i] = f(x);
        }
}
```

```

// загрузка для поиска стандартных отклонений
Vector<IncrementalStatistics> s(D);
for(int k = 0; k < nBoots; ++k)
{ // ресемплинг строк данных
    for(int i = 0; i < n; ++i)
    {
        int index = GlobalRNG().mod(n);
        yaR[i] = ya[index];
        for(int j = 0; j < D; ++j) ycR[j][i] = yc[j][index];
    }
    // оценка
    Vector<double> indicesR = findSobolIndicesHelper(yaR, ycR);
    for(int j = 0; j < D; ++j) s[j].addValue(indicesR[j]);
}
Vector<double> indices = findSobolIndicesHelper(ya, yc);
Vector<pair<double, double> > confs;
double z = getMixedZ(a/D);
for(int j = 0; j < D; ++j)
{
    double delta = s[j].stdev() * z;
    confs.append(make_pair(indices[j] - delta, indices[j] + delta));
}
return make_pair(indices, confs);
}

```

## 21.32. Последовательность Соболя

Во многих случаях — например, в задачах вычисления многомерных интегралов, моделирование обычно происходит на единичном гиперкубе, возможно расширенном обратным методом на более общие области. Здесь не исключено дальнейшее снижение дисперсии. Хитрость заключается в использовании специальных неслучайных выборок. Однородно случайные единичные выборки гиперкуба имеют тенденцию образовывать небольшие промежутки и кластеры и не покрывают гиперкуб равномерно, как нам бы хотелось.

Покрывание должно обладать малым *звездным несоответствием*:

$$D^*(n) = \max_x \left( \frac{\text{количество выборок в } X}{n} - \text{Vol}(X) \right),$$

где  $X$  —  $D$ -мерный гиперпрямоугольник с  $0 \leq x_j \leq u_j \leq 1$  среди всех возможных вариантов  $u_j$  (см. [21.90]). Многие *квазислучайные последовательности* обладают асимптоти-

чески оптимальным  $D^*(n) = O\left(\frac{\lg(n)^D}{n}\right)$ .

Их полезность показана *неравенством Коксм — Хлавки*: задана функция  $f$  с ограниченной *суммарной дисперсией* (см. [21.91])  $V$  в гиперкубе, для  $n$  выборок  $x_i$ ,

$|\text{Ave}(f(x_i)) - f_{\text{гиперкуб}}| \leq V(f) D^*(n)$ . Получается:  $O\left(\frac{\lg(n)^D}{n}\right)$  — это ошибка в худшем случае, которая, несмотря на превышение доверительной вероятности 95% по Монте-

Карло для типичного  $n$  и не слишком маленького  $D$ , лучше, по крайней мере, при  $D \leq 3$ , а на практике при большем  $D$ .

Для дифференцируемого  $f$   $V(f) = \int \|\nabla f\|$ , а для недифференцируемого  $f$  часто принимается предельное значение. Например, для  $f(x) = 1$ , если  $x$  находится в единичной окружности, и 0 в противном случае, с использованием предельного тонкого диска шириной  $h$ , где имеет место линейное увеличение от 0 до 1,  $V(f) = 2\pi$ .

Последовательность Соболя имеет теоретическую эффективность и быстро оценивается (см. [21.10]). Для любой размерности предвычислительных данных инициализации:

1. Выберите еще не выбранный примитивный двоичный многочлен  $x^D + a_1 x^{D-1} + \dots + a_{D-1} x + 1$  наименьшей степени.
2. Пусть точность типа `double` равна  $B$  битам.  $\forall d \in [0, D-1]$  и  $j \in [0, B-1]$  вычисляет  $v_j = \frac{2j+1}{2^{B-1-j}}$  для  $j < D$  и  $v_{j-D} \wedge v_{j-D} / 2^D \wedge a_{D-1} v_{j-1} \wedge \dots \wedge a_1 v_{j-D}$  в противном случае.
3. Установите счетчик  $k = 1$  и начальную переменную  $x = 0$ .
4. Чтобы сгенерировать:  $x_i \wedge = v_{i,c}$ , где  $c$  — позиция самого правого единичного бита  $k$ .

В реализации используется фиксированное количество полиномов, которое можно найти в работе [21.60] или в Интернете. Требование неравенства  $k < 2^B$  не является проблемой даже для очень длинных симуляций. Значения  $v$  представляют собой двоичные дроби, представленные целыми числами с нормирующим коэффициентом  $2^{-B}$ :

```
class Sobol
{ // Полиномы Соболя не содержат старших и младших единиц
    enum{B = numeric_limits<double>::digits};
    unsigned long long k; // номер текущего моделирования
    Vector<unsigned long long> x, v; // текущая выборка и предварительно
                                   // вычисленные данные

    double factor;
    int vIndex(int d, int b){return d * B + b;} // отображение на индекс массива
public:
    static int maxD(){return 52;}
    Sobol(int d): factor(1.0/twoPower(B)), v(d * B, 0), x(d, 0), k(1)
    {
        assert(d <= maxD());
        unsigned char const SobolPolys[] = {0,1,1,2,1,4,2,4,7,11,13,14,1,13,16,
            19,22,25,1,4,7,8,14,19,21,28,31,32,37,41,42,50,55,56,59,62,14,21,
            22,38,47,49,50,52,56,67,70,84,97,103,115,122},
            SobolDeps[] = {1,2,3,3,4,4,5,5,5,5,5,5,6,6,6,6,6,6,7,7,7,7,7,7,7,
            7,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8};
        for(int i = 0; i < d; ++i)
            for(int j = 0; j < B; ++j)
            {
                unsigned long long value;
                int l = j - SobolDeps[i];
                // ситуация j < D
                if(l < 0) value = (2 * j + 1) * twoPower(B - j - 1);
```

```

else
{ // ситуация j >= D
    value = v[vIndex(i, 1)];
    value ^= value/twoPower(SobolDeps[i]);
    for(int k = 1; k < SobolDeps[i]; ++k)
        if(Bits::get(SobolPolys[i], k - 1))
            value ^= v[vIndex(i, 1 + k)];
}
v[vIndex(i, j)] = value;
}
next();
}
void next()
{
    for(int i = 0, c = rightmost0Count(k++); i < x.getSize(); ++i)
        x[i] ^= v[vIndex(i, c)];
}
double getU01Value(int i) const { return x[i] * factor; }
double getUValue(int i, double a, double b) const
{ return a + (b - a) * getU01Value(i); }
};

```

Например, для оценки числа  $\pi$  конкретная симуляция с последовательностью Соболя дает значение 3,14158992 с ошибкой в наихудшем случае  $\leq 8,5e^{-5}$ , предполагая, что

$$D^*(n) = \frac{\lg(n)^D}{n}. \text{ Точность получается выше, чем у оценки Монте-Карло (см. главу 6.}$$

*Генерация случайных чисел*) примерно в 5 раз.

Последовательность Соболя (как и другие квазислучайные последовательности) имеет ограничения:

- ◆ функция  $f$  должна быть детерминированной, чтобы можно было применить неравенство Коксмы — Хлавки;
- ◆  $V(f)$  редко известна;
- ◆ неравенство Коксмы — Хлавки бесполезно за пределами очень маленького  $D$ ;
- ◆ не существует случайных границ — последовательность полностью детерминирована и может создать функцию, свойства которой не обнаруживаются при разумном  $n$ .

Несмотря на отсутствие разумных гарантий, в некоторых задачах, таких как вычисление интегралов, последовательность Соболя дает очень хорошую точность (см. [21.72]). Кажется, что она обладает многими хорошими свойствами случайных чисел, но не является при этом случайной, поскольку избегает пробелов и кластеров случайной выборки, из-за чего возникают смещения. Возможно, причина того, чтобы избежать наихудшего поведения, та же, что и у псевдослучайных чисел, — информация из последовательности и предметной области имеет тенденцию быть независимой.

Поскольку последовательность Соболя полностью детерминирована, единственным источником неопределенности являются значения  $f$  при неоцененном  $x$ . Простой способ получить доверительную границу — *рандомизированный квази-Монте-Карло* (Randomized Quasi Monte Carlo, RQMC) — сгенерировать равномерно случайное  $u$  в гиперкубе и объединить его с каждой выборкой Соболя  $x_i$ , используя  $x_{new,i} = (x_i + u) \% 1$ . Это

не отменяет несоответствия результирующей последовательности (см. [21.53]). Используя множество таких конфигураций, разные последовательности будут выбирать  $f$  в разных случайных независимых точках. Средние значения отдельных партий Соболя должны иметь низкую дисперсию, поэтому для точных границ CLT достаточно всегда использовать примерно 30 различных значений  $u$ .

Еще одно простое расширение — использовать случайную выборку, если размер превышает максимально поддерживаемый:

```
class ScrambledSobolHybrid
{
    int D;
    Vector<double> shifts;
    mutable Sobol s;
    Vector<pair<double, double> > box;
public:
    ScrambledSobolHybrid(Vector<pair<double, double> > const& theBox):
        D(theBox.getSize()), shifts(D), s(min(D, Sobol::maxD())), box(theBox)
    {
        for(int i = 0; i < D; ++i)
            shifts[i] = GlobalRNG().uniform(box[i].first, box[i].second);
    }
    Vector<double> operator()() const
    { // сперва берем отклонения Соболя для поддерживаемых измерений
        Vector<double> next(D);
        for(int i = 0; i < min(D, Sobol::maxD()); ++i)
            next[i] = s.getUValue(i, box[i].first, box[i].second);
        s.next();
        // случайное значение для случайных измерений
        for(int i = min(D, Sobol::maxD()); i < D; ++i)
            next[i] = GlobalRNG().uniform(box[i].first, box[i].second);
        // скремблирование
        for(int i = 0; i < D; ++i)
        {
            next[i] += shifts[i] - box[i].first;
            if(next[i] > box[i].second)
                next[i] -= box[i].second - box[i].first;
        }
        return next; // генерация первого значения
    }
};
```

Обычно неизвестно, является ли последовательность Соболя более точной, чем случайная выборка, но на практике чаще всего  $D = 1000$  или около того, что и было проверено в литературе. Предпочтительным практическим методом обычно является RQMC. Но чистая последовательность Соболя без доверительного интервала, как правило, дает лучшую оценку, для которой можно эвристически использовать математически неверный интервал из независимых случайных данных.

Генерация работает с прямоугольными областями, но с помощью преобразований можно семплировать другие области. Так, в машинном обучении часто полезно выбирать геометрические сетки — например, от  $10^{-3}$  до  $10^3$  или около того. Логарифмическое преобразование превращает эту область в гиперпрямоугольник:

```

template<typename SAMPLER = ScrambledSobolHybrid> class GeometricBoxWrapper
{
    static Vector<pair<double, double> > transformGeometricBox
        (Vector<pair<double, double> > box)
    {
        for(int i = 0; i < box.getSize(); ++i)
        {
            assert(box[i].first > 0 && box[i].first < box[i].second);
            box[i].first = log(box[i].first);
            box[i].second = log(box[i].second);
        }
        return box;
    }
    SAMPLER s;
public:
    GeometricBoxWrapper(Vector<pair<double, double> > const& box):
        s(transformGeometricBox(box)) {}
    Vector<double> operator() () const
    {
        Vector<double> result = s();
        for(int i = 0; i < result.getSize(); ++i) result[i] = exp(result[i]);
        return result;
    }
};

```

## 21.33. Планирование экспериментов: основные идеи

Невозможно сделать полезные выводы из данных, полученных в ходе плохо спланированного исследования. Эксперимент нужно разрабатывать так, чтобы получить максимальную достоверность и максимально уменьшить дисперсию. Иногда эти параметры конфликтуют. Например, для сравнения нескольких алгоритмов можно использовать:

- ◆ независимые входные данные — вы получите доверительный интервал для каждого алгоритма, предполагая, что данные независимы и случайны;
- ◆ общие входные данные — уменьшают дисперсию из-за выбора входных данных, но не могут давать общедопустимый доверительный интервал для каждого алгоритма. Парные интервалы для вычисления разницы в производительности оказываются короче, чем у независимых данных.

Требования перестановочных тестов удовлетворяются с помощью планирования (дизайна) эксперимента. Обычно *факторы* (*переменные* на языке Design of Experiments, DOE) непрерывны, возможно, в ограниченных диапазонах в соответствии со знанием предметной области. Общие принципы планирования эксперимента:

- ◆ *рандомизация* — выборка данных случайным образом или хотя бы произвольное распределение субъектов по сравниваемым группам. Это позволяет использовать перестановочное тестирование, потому что все возможные назначения становятся равновероятными. Рандомизация — единственный способ убедиться, что случайное



поведение в окружающей среде искажает результаты не более чем случайно. Поэтому установите переменные, которыми можно управлять, на случайные значения;

- ◆ *блокировка* — для переменных, над которыми имеется прямой контроль, но которые в эксперименте не изучаются, используется рандомизация, но она может уменьшить дисперсию путем сравнения ответов на фиксированных *уровнях (значениях переменных на языке DOE)*. Процесс аналогичен спариванию в тестах, подобных тесту Неменьи, и похож на общую стратегию случайных чисел. Нам необходимо предотвратить расхождение в ответах из-за различий в уровнях. Поэтому блокируйте то, что можно заблокировать, и рандомизируйте то, что нельзя. Уровни блокировки должны иметь научное значение — например, для сравнения алгоритмов используйте максимально возможные размеры экземпляров и меньшие, которые имеют значение для более редких вариантов использования. Хотя можно сказать, что для сравнения алгоритмов при выборе входных данных используется блокировка, а не рандомизация (но для анализа предполагается рандомизация);
- ◆ *репликация* — для уменьшения систематической ошибки, связанной с выбором определенного уровня, используется множество разных уровней. Например, сравните алгоритмы на множестве разных задач. Это относится как к заблокированным, так и к рандомизированным переменным. Репликация  $\neq$  повторению, потому что используются разные значения уровня. Репликация, по существу, эквивалентна увеличению размера выборки.

Классический эксперимент: *женщина, пробующая чай*, — у известного статистика Фишера была коллега, которая утверждала, что может определить определенное качество чая по вкусу. Он дал ей четыре чашки одного чая и четыре чашки другого в произвольном порядке. Шанс случайно найти правильную перестановку невелик. В итоге получился перестановочный тест, в котором каждая последовательность решений столь же вероятна, как и любая другая. Но обратите внимание, что здесь не проводился анализ мощности (Фишер был против идеи альтернативных гипотез). Использование четырех чашек дает достаточно хорошее разрешение только для  $p$ -значения, но анализ мощности требует указания области безразличия и альтернативы, которая его удовлетворяет. Также тест предполагает случайную предсказательную способность, хотя логически можно было бы ожидать чуть лучшую, чем случайная.

Планирование экспериментов регрессии (см. главу 27. *Машинное обучение: регрессия*) принципиально отличается от планирования эксперимента сравнения. Линейная регрессия часто упоминается в литературе из-за ее полезности в предварительном анализе. Ее низкая ошибка оценки позволяет создать полезную модель из нескольких выборок, что очень важно, если проведение эксперимента дорого. В регрессии используются предположения, которые изучают  $y$  при заданном  $x$ , и связь является линейной. Это означает, что не нужно исследовать предметную область путем выборки в разных областях. Требуется достаточно хорошая эффективность оценки.

Чтобы найти оптимальное значение чего-либо, следует пропустить регрессию и планирование эксперимента и использовать метод числовой минимизации — например, метод Нелдера — Мида или SPSA (см. главу 24. *Численная оптимизация*). Но часто требуется получить некоторое представление о поведении изучаемой системы — например, для процедуры выпечки хлеба возможными факторами являются температура, степень вентиляции, тип муки и т. д.

В частности, для линейной регрессии с  $k$  факторами можно рассмотреть *полный факторный дизайн*  $2^k$  — используйте любые 2 уровня для любой переменной и попробуйте все возможности. Это также позволяет легко найти решение: любая переменная имеет  $2^{k-1}$  оценок соответствующего коэффициента линейной регрессии, которые усредняются. Интересная особенность заключается в том, что это позволяет наиболее эффективно использовать  $2^k$  прогонов для оценки линейной модели при нормальном предположении о независимости и случайности оценки. Это следует из свойств линейной модели — оптимальный план для минимизации дисперсии следует из *D-оптимальности*: максимизировать  $\det(X^T X)$ , где  $X$  — матрица значений уровней факторов для прогонов, масштабированная так, что значение равно  $-1$ , второе —  $1$ . Полный факториал гораздо более эффективен, чем очевидный анализ по одному фактору за раз.

При большом  $k$  полный факториал нецелесообразен, и для оценки требуется только  $k + 1$  пример. Простым обходным решением является вычисление оптимального плана для заданного количества запусков с использованием D-оптимальности (см. [21.29]). Это работает для оценки линейных моделей в системах с низкой дисперсией и позволяет учитывать такие ограничения, как предварительное знание невозможности некоторых обработок в факторном плане. Вы можете использовать результаты, чтобы решить, какие факторы могут быть нерелевантными (на основе проверки гипотезы о нормальных ошибках с поправкой на множественность — примеры можно найти в работе [21.29], хотя для этого лучше использовать регрессию лассо — см. главу 27. *Машинное обучение: регрессия*) или предназначенными для оптимизации. Литературы по DOE много. Практическая проблема заключается в том, что было предложено и проанализировано очень много конструкций, но лишь недавно появился тренд на автоматическую генерацию.

Для последующей оптимизации нужно выбрать лучшее значение из эксперимента с линейной моделью в области удвоения доверия (см. главу 24. *Численная оптимизация*). Такие суррогатные модели (см. главу 27. *Машинное обучение: регрессия*), как правило, работают лучше, позволяя использовать более мощные модели. Это может быть более эффективным, чем прямая числовая оптимизация.

Во многих задачах регрессии ресурсы оценки не сильно ограничены, и мы будем пробоовать несколько моделей с процессом выбора. Здесь рандомизация кажется лучшим выбором, чем блокирование факторов, поскольку определяющие критерии неприменимы к другим моделям, а для таких подзадач, как оценка производительности, исследование важно. Хочется иметь дизайн, который хорошо работает для любой модели. Случайный выбор для этого подходит. Когда оценки стоят дорого, нужно избегать плохих проектов, но лучший требуется редко.

Планы экспериментов, основанные на последовательностях с низким расхождением, тоже хороши, потому что они пытаются хорошо работать для всех возможных функций. Еще одна возможность — *выборка латинского гиперкуба* (Latin Hypercube Sampling, LHS) — разбить пространство на области и взять выборку из каждой области. Это нечто среднее между случайной и квазислучайной выборкой. LHS, вероятно, является самой неслучайной выборкой, которая тем не менее имеет достаточную случайность для поддержки большого числа вариантов математического анализа. Но она работает хуже, чем последовательность Соболя, в задачах интеграции (см. [21.49]) и не масштабируется за пределы малых  $D$  (хотя ее можно комбинировать с дробным факториалом, используя последний для выбора областей). Она также не вычисляется в реальном времени.

*Наблюдательные исследования* сильно отличаются от заранее спроектированных из-за отсутствия явной рандомизации. Например, при сравнении двух популяций случайное распределение означает, что любое значимое различие можно отнести к членству в популяции, потому что рандомизация уравнивает все остальное. Но если данные получены из наблюдений, возможность замены данных теряется, поэтому атрибуция недействительна. Например, если группе пациентов, принимающих одно лекарство, становится лучше, чем группе, принимающей другое лекарство, это может быть связано с тем, что последняя группа изначально находится в худшем состоянии. Общее заблуждение заключается в том, что у субпопуляции есть свойство, которого нет у генеральной совокупности, т. е. определенная субпопуляция может предрасполагать ее к такому свойству.

Поскольку большая часть данных является наблюдательной, а спроектированные исследования часто использовать просто непрактично, наблюдательные исследования полезны с научной точки зрения, если позволяют доказать, что распределение выборок, будучи случайным, имело бы такой же вид. Этот вывод был сделан, например, для классического исследования связи между курением и раком. Статистика — это уровень заболеваемости раком среди курильщиков и некурящих. В отличие от спланированного исследования, неизвестно, развился бы рак у курильщиков, больных раком, если бы они не курили. Это можно проверить только на некурящих. Аргумент состоял в том, что решение человека курить не связано с какими-либо другими потенциально вызывающими рак факторами — такими как генетическая предрасположенность или культурные предпочтения. Кроме того, размеры выборки были большими, а на выводы главного хирурга влияли внешние факторы, такие как возникновение рака полости рта.

Общий метод улучшения заключается в том, чтобы сбалансировать распределение с помощью некоторых логических факторов, где были бы представлены все соответствующие категории выборок. Например, в медицинских исследованиях объединяются люди одинакового возраста, общего состояния здоровья и т. п. Пока удастся избежать очевидных предубеждений, такие исследования обычно считаются достаточно достоверными. Тем не менее известно много случаев, когда последующий эксперимент противоречит результатам наблюдательного исследования, и в этих случаях требуется научная помощь. *Противоречащие фактам* данные не рассматриваются: из многих возможных исходов наблюдаются только конкретные, а данные типа «как изменилась бы ваша жизнь, если бы вы не поступили в колледж» в расчет не принимаются. Исследование связи между переменными всегда оправдано. Идея состоит в том, что можно наблюдать только то, что произошло, а не то, что не произошло, и ассоциации используются только в первом случае (см. подробности в работе [21.49]). В некоторых случаях проекты являются *квазиэкспериментальными*, и тогда нельзя использовать случайные выборки или распределения, но нужно по возможности рандомизировать и сбалансировать все, что можно.

## 21.34. Марковская цепь Монте-Карло

Для распределений без известных генераторов можно использовать методы генерации, основанные на *марковской цепи Монте-Карло* (Markov Chain Monte Carlo, MCMC), причем иногда только их и можно использовать. Они производят слегка зависимые и одинаково распределенные выборки, а при отсутствии случайных независимых данных — это лучший вариант.

Алгоритму случайного блуждания по Метрополису (Random Walk Metropolis, RWM) нужно распределение PDF, заданное с точностью до нормировочной константы:

1. Начать с исходной выборки  $x_0$ , такой что  $f(x_0) \neq 0$ .
2. Использовать предоставленный пользователем семплер  $p$  для создания выборки.
3.  $x_{\text{new}} = x + \text{выборка}$ .
4. Установить  $x \leftarrow x_{\text{new}}$ , если  $\text{uniform}(0, 1) \leq f(x_{\text{new}}) / f(x)$ .

Теоретически, семплер  $p$  должен быть таким, чтобы PDF везде была положительна и симметрична. Например,  $p$  может быть нормальным со средним значением 0 или симметризованным Леви с толстым хвостом. Тогда  $x_i$  образуют цепь Маркова, конечное распределение которой равно  $f$ . Это связано с тем, что цепь эргодична и по построению удовлетворяет условию *детального баланса*. Скорость сходимости имеет значение и зависит от неизвестных констант (см. [21,23]). Вы можете проверить сходимость по Колмогорову — Смирнову, и она вполне возможна, потому что начальное значение  $x_0$  может быть очень маловероятным и в среднем приводит к большому количеству отклонений теста, чем ожидалось. Таким образом, эмпирическое правило состоит в том, чтобы отбросить некоторое количество первых  $x_i$ , обычно 1000.

Почти никогда это не проблема, но технически цепочка никогда не бывает эргодичной, потому что числовая ошибка приводит к тому, что значения  $p$  приближаются к нулю в хвостах, особенно если у распределения  $p$  тонкие хвосты. Таким образом, на практике можно воспользоваться равномерным распределением с большим диапазоном (который не превышает 0 за пределами границ). Теоретические рекомендации по выбору  $p$  и его дисперсии (или эквивалентного параметра) ограничены, но они должны быть такими, чтобы коэффициент приемлемости был далек как от 0, так и от 1. Для устранения корреляции между последовательными выборками можно отбросить некоторые после последней использованной выборки, но неясно, сколько именно отбрасывать.

Простая эвристика для решения этих проблем состоит в том, чтобы позволить цепочке найти правильное распределение с помощью поиска по сетке (см. главу 24. *Численная оптимизация*). Чтобы создать новую выборку, нужно взять имеющуюся выборку из равномерного распределения с очень малой дисперсией и удвоить ее. Предложения с небольшой дисперсией принимаются, но не имеют значения, а предложения с большой дисперсией в основном отклоняются. Поэтому используются только дисперсии правильной величины, и нужно принять меры, чтобы уменьшить корреляцию результирующих выборок. Количество вызовов  $f$  по-прежнему составляет  $O(1)$  из-за равной  $O(1)$  разницы в логарифмах границ дисперсии:

```
template<typename PDF> class GridRWM
{
    PDF f;
    double x, fx, aFrom, aTo;
    int from, to;
    double sampleHelper(double a)
    {
        double xNew = x + GlobalRNG().uniform(-a, a), fxNew = f(xNew);
        if(fx * GlobalRNG().uniform01() <= fxNew)
        {
            x = xNew;
```

```

        fx = fxNew;
    }
    return x;
}

public:
    GridRWM(double x0 = 0, PDF const& theF = PDF(), int from = -10,
        int to = 20): x(x0), f(theF), fx(f(x)), aFrom(pow(2, from)),
        aTo(pow(2, to)) {}
    double sample()
    {
        for(double a = aFrom; a < aTo; a *= 2) sampleHelper(a);
        return x;
    }
};

```

Для  $D > 1$  нужно сгенерировать многомерную равномерную выборку, но, поскольку вероятность отбраковки увеличивается с ростом  $D$ , дополнительная эвристика заключается в использовании для сетки мультипликативного коэффициента  $2^{1/D}$ . Из-за того, что увеличение времени выполнения зависит только от  $D$ , алгоритм это масштабируется до больших значений  $D$ :

```

template<typename PDF> class MultidimGridRWM
{
    PDF f;
    Vector<double> x;
    double fx, aFrom, aTo, factor;
    Vector<double> sampleHelper(double a)
    {
        Vector<double> xNew = x;
        for(int i = 0; i < xNew.getSize(); ++i)
            xNew[i] += GlobalRNG().uniform(-a, a);
        double fxNew = f(xNew);
        if(fx * GlobalRNG().uniform01() <= fxNew)
        {
            x = xNew;
            fx = fxNew;
        }
        return x;
    }
public:
    MultidimGridRWM(Vector<double> const& x0, PDF const& theF = PDF(),
        int from = -10, int to = 20): x(x0), f(theF), fx(f(x)), aFrom(pow(2,
        from)), aTo(pow(2, to)), factor(pow(2, 1.0/x.getSize())) {}
    Vector<double> sample()
    {
        for(double a = aFrom; a < aTo; a *= factor) sampleHelper(a);
        return x;
    }
};

```

Для очень большой  $D$ -сетки поиск может не дать достаточно независимых выборок, поэтому лучше проверить его результаты на любом другом распределении и понять,

достаточно ли часто сходятся основанные на выборке оценки некоторых свойств. Уменьшение коэффициента умножения — один из способов уменьшить корреляцию, но из-за проклятия размерности этот способ вряд ли поможет.

## 21.35. Байесовские методы

Многие проблемы интуитивно воспринимаются как четко определенные, но при математическом описании не имеют хорошего разумного решения. Такие задачи *некорректны*, поскольку не содержат необходимой информации. В статистике некорректные задачи обычно возникают тогда, когда не хватает данных. Вернемся, например, к оценке  $s^2$  из рассмотренного ранее примера.

Основной подход к решению предполагает наличие условий, которые должны выполняться естественным образом, но не являются частью спецификации задачи. Такие допущения могут сделать задачу разрешимой, но создают проблемы. Во многих задачах невозможно сделать вывод без некоторых допущений, и лучше сделать допущение и создать полезную модель, чем вообще ничего не делать.

Частотная статистика означает смотреть на мир на основе каких-то данных *iid* и не знать реальности. Байесовская логика позволяет моделировать представления о реальности с точки зрения лица, принимающего решения. Предположение о том, что вера приблизительно равна реальности, является предположением моделирования, как и то, что распределение является нормальным. Так, например, в частотном доверительном интервале границы случайны, а  $\theta$  фиксировано, но в *байесовском доверительном интервале*  $\theta$  является случайным, а границы фиксированы. Первое говорит о том, что с высокой вероятностью границы близки к  $\theta$ , а последнее о том, что  $\theta$  близко к границам.

Основной алгоритм байесовского вывода:

1. Выбрать неинформативное априорное распределение для данных и вероятностное распределение для параметров.
2. Попробовать сформировать апостериорное распределение аналитически, используя теорему Байеса.
3. Если распределение сформировано, сделать из него аналитический вывод.
4. В противном случае вывод делается из выборок, смоделированных с помощью MCMC.

Описанное относится только к вероятностным моделям и мало чем отличается от частотного вывода. Несколько основных соображений:

- ◆ априорные значения обычно вносят небольшое смещение, но снижают дисперсию. В байесовских методах рекомендуется использовать неинформативные, *объективные априорные значения*, т. к. с научной точки зрения они полезнее (для получения воспроизводимых результатов). Использование в статистических выводах *субъективных априорных данных*, основанных на существующих знаниях в предметной области, осуждается. Это связано с тем, что даже при должной осторожности и добрых намерениях невозможно избавиться от некоторых произвольных входных данных и потери воспроизводимости, поскольку при достаточно сильном априорном

знании вам не нужно проводить эксперимент вовсе — вы просто используете априорное значение в качестве апостериорного;

- ◆ априорные значения иногда делают некорректно поставленные задачи корректными и позволяют решать неуклюжие или не поддающиеся решению задачи аналитически с помощью частотных методов. Примером тому являются *иерархические модели* (см. [21.27]), где некоторые входные данные для модели сами по себе являются результатом подмодели. Хотя байесовская логика здесь естественна, иногда такие модели решаются с привлечением частотной логики с помощью алгоритма ЕМ для смеси распределений (см. главу 28. *Машинное обучение: кластеризация*, где описан алгоритм ЕМ с примесью гауссиана);
- ◆ априорное значение позволяет использовать теорему Байеса, которая в некоторых случаях может существенно упростить вывод, хотя обычно априори в лучшем случае оценивается на основе другой информации — например, о проведенных ранее исследованиях. Так, для вывода ОСВА (обсуждается далее в этой главе) работать с теоремой Байеса намного проще;
- ◆ когда апостериорное значение известно аналитически, частотное максимальное правдоподобие и апостериорный вывод, как правило, оказываются непротиворечивы (см. [21.79]). Если вы не знаете апостериорного анализа аналитически, вы теряете эффективность вычислений;
- ◆ байесовские методы нуждаются в вероятностной модели и не имеют эквивалентов различных непараметрических тестов, хотя существуют и непараметрические байесовские методы.

Для получения дополнительной информации о байесовских методах почитайте работы [21.23 и 21.27]. В работе [21.66] приведено сравнение частотных и байесовских оценок. Одной из особых проблем для последнего являются многомерные пространства, где априорное значение, по сути, подавляет реальные данные.

В частности, байесовские методы часто дают хорошие оценки с частотной точки зрения. На байесовский взгляд они автоматически верны, поскольку выводятся из определенных логических аксиом, но это правдиво только при условии правильного априорного значения и его веса. Как отмечено в работе [21.58], частотность была открыта первым мошенником, который понял, что можно зарабатывать деньги на азартных играх. В подобной ситуации только повторные эксперименты могут помочь выявить какие-то свойства, и никакая вера не поможет. Поэтому такие частотные свойства, как постоянство, весьма важны. По сути, байесовская статистика похожа на максимальную вероятность с точки зрения принципа получения хороших оценок. Достоверные интервалы также можно использовать в качестве алгоритма для вычисления доверительных интервалов с последующей частотной проверкой, но это используется редко. Таким образом, частотные свойства, видимо, полезны только при принятии решений отдельно от статистики. Любая попытка оценить производительность для конкретной ситуации должна быть частотной, хотя байесовская оценка может дать лучшие результаты.

Принятие решений для отдельного индивидуума выполняется не так, как оценка параметров популяции, потому что апостериорное распределение обычно достаточно хорошо известно. Здесь лучше всего работает байесовская логика. Например, если медицинское обследование человека указывает на наличие какого-то заболевания, по теореме Байеса очень маловероятно, что у человека есть это заболевание, если оно встре-

чается очень редко. Байесовская логика также ставит вопрос: а почему бы не попробовать всё в жизни, включая некоторые потенциально вредные или опасные вещи, и понять, нравятся ли они вам? Большое количество априорных знаний мешает новым данным повлиять на мнение. Кроме того, определенные события, такие как победа на определенных президентских выборах, не связаны с распределением вероятностей с точки зрения частотного анализа, поскольку они происходят только один раз, но байесовские методы моделируют убеждения, поэтому здесь они тоже применимы. Однако конкретный метод оценки (который может включать стратегии опроса и т. д.) может быть оценен в частотной логике на множестве всех президентских выборов.

В статистике оценка обычно является лишь частью проблемы. Другая ее часть — это предположения и способы использования оценки. Разница с точки зрения оценки в том, что «частотники» рассматривают другие данные после анализа, а «байесовцы» — до. Например, на фондовом рынке неизвестные параметры оцениваются, используя частотные методы, а байесовская логика, позволяет решить, на что делать ставку. Многие из этих ставок проходят регрессионное тестирование по известной инвестиционной истории, и это позволяет выявить частотные свойства.

## 21.36. Поиск лучшей альтернативы с помощью моделирования

При  $k = 2$ , чтобы решить, является ли одна альтернатива статистически лучше другой, нужно смоделировать каждую из них много раз (выполняя моделирование до тех пор, пока предположения о нормальности не станут разумными). Если ограничить бюджет и проверить результаты в конце, нужно будет вычислить  $k$  доверительных интервалов с кратностью  $k$ .

Возможно, эффективнее было бы выполнять моделирование до тех пор, пока с уверенностью не обнаружится победитель, в надежде, что бюджет не исчерпается раньше. Это добавляет дополнительную множественность из-за потенциального пропуска будущих сравнений. Эффективный способ сравнения — выполнить его после удвоения количества симуляций. Это увеличивает множественность только в  $O(\lg(\text{бюджет}))$  раз. Реализация предполагает, что вся множественность помещается в параметр  $a$ :

```
bool isNormal0BestBonf(Vector<NormalSummary> const& data, double aLevel)
{ // наименьшее значение является лучшим с точностью meanPrecision
    int k = data.getSize();
    assert(k > 1);
    double z = find2SidedConfZBonf(k, 1 - aLevel),
        upper = data[0].mean + z * data[0].stddev();
    for(int i = 1; i < k; ++i)
    {
        double lower = data[i].mean - z * data[i].stddev();
        if(lower <= upper) return false;
    }
    return true;
}
```

Этот наивный подход неэффективно выполняет моделирование, рассматривая альтернативы с небольшими отклонениями и неравномерными соревновательными средства-



ми. В таком случае требуется в  $k$  раз меньше симуляций, если используются те же критерии завершения, что дает достаточно возможностей для улучшения.

Байесовская логика бывает полезна для выбора следующей альтернативы по той причине, что при наличии предварительных убеждений множественные правила проверки и остановки игнорируются. В частности, для той или иной наилучшей альтернативы со средним значением  $m_0$  считается наилучшим случай, когда  $\Pr(m_0 < m_1 \cap m_0 < m_{k-1})$  достаточно велико. Применяя неравенство Бонферрони, получаем  $\geq 1 - \sum_{i>0} (1 - \Pr(m_0 < m_i))$ . Когда  $m_i$  распределено нормально, например в случае моделирования с достаточным количеством выборок, можно вывести  $\Pr(m_0 < m_i)$  из нормальной разницы между альтернативами 0 и  $i$ , как при  $k = 2$ .

Основанная на описанном представлении эвристика ОСВА (Optimal Computing Budget Allocation, оптимальное распределение вычислительного бюджета) проста и является одной из самых эффективных. Теорема ОСВА (см. [21.17]): Если в результате начального моделирования  $m_i$  распределено нормально с дисперсиями  $s > 2$ , количество дополнительных моделирований  $T$ , необходимых для достижения достоверности Бонферрони  $p$ , минимизируется при  $p \rightarrow 1$ , когда относительные коэффициенты распределения

$$\text{заданы как } R_i = \begin{cases} \sum_{i>0} \frac{R_i^2}{s_i^2}, i = 0 \\ \frac{s_i^2}{(m_i - m_0)^2} \end{cases}.$$

Таким образом, эвристика говорит, что следующее моделирование надо выполнять на альтернативе с самым высоким  $R_i$ . Например, если есть распределения (1, 0,1), (2, 0,2), (3, 0,3), то  $R_1 = 0,2$ ,  $R_2 = 0,075$  и  $R_0 \approx 0,15$ , поэтому выбирается вариант 1:

1. Промоделируйте каждую альтернативу  $n_0$  раз, чтобы получить исходные  $m_i$  и  $s_i$ .
2. До тех пор, пока не будет превышен бюджет или пока не будет определен победитель со скорректированной уверенностью:
3. Смоделируйте альтернативу с самым высоким  $R_i$ .

Значение  $n_0$  должно быть  $\geq 30$ , чтобы использовать нормальное распределение, и чтобы от CLT был разумный эффект. В случаях, когда выполнять моделирование очень дорого, используйте  $n_0 \geq 5$  (см. [21.17]). Время выполнения на итерацию составляет  $O(k)$ :

```
template<typename MULTI_FUNCTION> struct OCBA
{
    MULTI_FUNCTION const& f;
    Vector<IncrementalStatistics> data;
    int nDone;
    OCBA(MULTI_FUNCTION const& theF = MULTI_FUNCTION(), int initialSims = 30):
        f(theF), data(theF.GetSize())
    {
        int k = f.GetSize();
        for(int i = 0; i < k; ++i)
            for(int j = 0; j < initialSims; ++j) data[i].addValue(f(i));
        nDone = k * initialSims;
    }
}
```

```

pair<Vector<NormalSummary>, int> findBest()
{
    int k = f.getSize();
    Vector<NormalSummary> s;
    for(int i = 0; i < k; ++i) s.append(data[i].getStandardErrorSummary());
    int bestI = 0, bestRatioI = -1;
    double bestMean = s[0].mean, ratioSum = 0, bestRatio;
    for(int i = 1; i < k; ++i)
        if(s[i].mean < bestMean) bestMean = s[bestI = i].mean;
    swap(s[0], s[bestI]);
    return make_pair(s, bestI);
}

void simulateNext()
{
    pair<Vector<NormalSummary>, int> best = findBest();
    int k = f.getSize(), bestI = best.second, bestRatioI = -1;;
    Vector<NormalSummary> s = best.first;
    // вычисление наибольшего соотношения ОСВА
    double bestMean = s[0].mean, ratioSum = 0, bestRatio;
    for(int i = 1; i < k; ++i)
    {
        double meanDiff = s[i].mean - bestMean, ratio =
            s[i].variance/(meanDiff * meanDiff);
        ratioSum += ratio * ratio/s[i].variance;
        if(bestRatioI == -1 || ratio > bestRatio)
        {
            bestRatio = ratio;
            bestRatioI = i;
        }
    }
    double ratioBest = sqrt(ratioSum * s[0].variance);
    if(ratioBest > bestRatio) bestRatioI = bestI;
    else if(bestRatioI == bestI) bestRatioI = 0;
    // моделирование альтернативы с наибольшим соотношением
    data[bestRatioI].addValue(f(bestRatioI));
    ++nDone;
}

int simulateTillBest(int simBudget = 100000, double aLevel = 0.05)
{
    assert(nDone < simBudget);
    int k = f.getSize(), nTests = lgCeiling(simBudget) - lgFloor(nDone);
    while(nDone < simBudget)
    {
        simulateNext();
        if(isPowerOfTwo(nDone) || nDone == simBudget - 1)
        {
            Vector<NormalSummary> s = findBest().first;
            if(isNormal0BestBonf(s, aLevel/nTests)) break;
        }
    }
    return nTests;
}
};

```

Например, при наличии шести альтернатив положим, что истинная производительность альтернативы  $i$  (начиная с 1) определяется как  $\text{normal}(i, 10 - i)$ . ОСВА и наивный подход соответственно требует в среднем около  $11\,000 \pm 1000$  и  $29\,000 \pm 2400$  симуляций (при 10 000 повторений), чтобы найти лучшее с достоверностью 0,95. От прогона к прогону альтернатива 0 всегда выбирается правильно, но общее количество симуляций для обоих подходов существенно различается.

Если несколько лучших альтернатив имеют одинаковое значение  $E[x]$ , ОСВА не сможет различить их. Интересное расширение (не реализованное здесь) состоит в том, чтобы ввести средний параметр безразличия  $\epsilon$ , чтобы пользователю было все равно, хуже ли выбранная альтернатива наилучшей. ОСВА будет использовать значение  $m_0 - \epsilon$  вместо  $m_0$  при вычислении  $R_i$  и соответственно сравнивать наилучшие. Количество необходимых симуляций неизвестно, но в худшем случае, когда все альтернативы имеют одинаковое значение  $E[x]$  и большое  $s_i^2$ , число симуляций задается как  $O(\sum s_i^2 \epsilon^{-2})$  с учетом того, как вычисляется доверительный интервал Монте-Карло.

Наиболее полезным аспектом ОСВА, по-видимому, является распределение следующей симуляции с фиксированным бюджетом. Для значений  $\epsilon \in [0, 1]$  критерий UCS (см. главу 29. *Машинное обучение: другие задачи*) более популярен, но неясно, как он соотносится с ОСВА. Сравнение и наивный алгоритм распространяются на поиск нескольких лучших альтернатив, а ОСВА — нет.

## 21.37. Расчет размера выборки

Перед проведением статистического эксперимента обычно необходимо указать его цели, чтобы обосновать стоимость. В частности, нужно знать, сколько образцов требуется для получения желаемых выводов. Это не имеет решающего значения для компьютерных экспериментов, где дополнительные образцы практически бесплатны, но важно, например, для медицинских исследований с участием людей.

Эмпирическое правило гласит, что мощность проверки гипотезы должна составлять около 80% (см. [21.64]). Проблема в том, что даже для параметрических методов — таких как доверительные интервалы для среднего значения при предположениях о нормальности, необходимо иметь предварительные оценки дисперсии и определять размер значимой разницы. Такой байесовский подход привел к разработке множества формул для различных процедур (см. [21.64]).

В простых, но ресурсоемких процессах вместо формул применяется моделирование. Требуется настроить несколько сценариев вида «что, если», в которых ожидается обнаружение определенной разницы, а затем использовать экспоненциальный поиск, чтобы найти  $n$ , моделирование с которым примерно приводит к 80%-ной мощности или желаемой длины доверительного интервала. Это также относится даже к непараметрическим процедурам. Требуется также решить, какие распределения взять для моделирования, и обычно оценки отличаются сильно, порой даже в 5 раз (см. [21.64]). Обычно выбираются нормальные распределения, но иногда нужны и распределения с толстыми хвостами вроде Коши, а также асимметричные, такие как логнормальные.

Более эффективно производить расчет доверительного интервала, если вы работаете с тестом. Доверительный интервал используется как искусственная статистика (как

средний ранг в случае теста Неменьи). Таким образом, вы не зависите от null-значения, удаляя из моделирования все параметры.

## 21.38. Анализ временных рядов

Статистика обычно изучает независимые случайные данные, но большая часть общих данных связана зависимостями, которые нельзя игнорировать. Особый вид зависимости — временной, т. е. когда данные упорядочены во времени. Например, рассмотрим цену акций компании  $x_i$  на конец дня и то, как она изменяется с момента  $x_0$  в начале дня до последней цены  $x_n$  в конце дня.

Существует несколько простых стратегий, позволяющих свести анализ к привычным методам:

1. Вычислить различия в данных  $\Delta x_i = x_i - x_{i-1}$  и предположить, что они одинаковы, т. к. это позволяет делать регулярные статистические выводы. Такая модель часто полезна, например, для фондового рынка, который, как известно, реагирует в основном на новую информацию. Но в этом случае предполагается, что  $x_{i-1}$  не содержит шума, а это не так. То есть потенциально лучшая модель состоит в том, чтобы взять вместо  $x_{i-1}$ , возможно, среднее значение нескольких последних  $x_i$ , как в регрессии ближайшей окрестности.
2. Предположить, что  $x_i$  является функцией времени  $i$ , и использовать регрессию (см. главу 27. *Машинное обучение: регрессия*). Это позволит понять общую тенденцию.
3. Игнорировать структуру временного ряда и модель  $x_i$  как функцию другой информации. Это хороший подход, но он менее эффективен, если часть поведения  $x_i$  в некоторой степени зависит от предыдущих значений.

Литература по временным рядам почти исключительно сосредоточена на линейных моделях вида  $x_{i+1} = \sum_{0 < i < p} \phi_i x_{i-i} + w_{i+1}$ , где все члены шума  $w \sim \text{normal}(0, \sigma)$ . Такие модели обобщают случай (1), где  $\Delta x_i = w_{i+1}$  и является функцией прошлых  $p$  точек данных. Это называется *авторегрессией* (Autoregression, AR), потому что значения  $x_{i-i}$  регрессируют сами по себе. В качестве введения в эту тему можно почитать книгу [21.43], но там используются несколько иные обозначения.

Нелинейные модели этих значений также возможны, но они не сильно лучше. Это может стать неожиданностью, потому что, например, регрессия случайного леса обычно намного лучше, чем линейная регрессия, но у авторегрессии есть рекурсивная структура со следующими своими свойствами:

- ♦ у рекурсии проблемы со стабильностью. Например, ряд Фибоначчи имеет форму AR, и значения неограниченно возрастают. С другой стороны, функции вроде  $x_{i+1} = 0,9x_i$  затухают до стабильного значения. В целом рост или затухание экспоненциальны и зависят от наибольшего корня в абсолютном значении полинома, представленного моделью AR (или эквивалентного собственного значения соответствующей матрицы — см. главу 23. *Численные алгоритмы: работа с функциями*). У нелинейных моделей нет явно описывающего их полинома, но обычно можно использовать локальные линейные приближения — такие как ряды Тейлора, и определить его численно;

- ◆ рекурсивная модель не должна иметь предсказательную силу — она больше предназначена для устранения структуры зависимостей. *Модель динамической регрессии* затем использует другую, вероятно, нелинейную модель для моделирования шума из модели AR.

Оценка модели AR и ее обобщений является основной задачей в литературе. Очевидным решением кажется минимизирование суммы квадратов невязок для выборки, где по заданным текущим значениям  $\phi_i$  невязка  $i$  вычисляется как прогнозируемое  $x_{t-i}$  — реальное  $x_{t-i}$ . Здесь можно пропустить первые значения  $p$  (*условная сумма квадратов*) или считать их предыдущими  $x_i = 0$  (*безусловная сумма квадратов*). Но у этого подхода есть пара проблем:

- ◆ процесс временных рядов бесконечен в прошлом — это единственный метод, для которого такие понятия, как согласованность оценок, могут иметь смысл. То есть предыдущее  $x_{t-i} \neq 0$ . Это не является прямой проблемой для оценки AR, но для более общей оценки ARMA — да (обсуждается позже);
- ◆ минимум цели метода наименьших квадратов не обязательно является значением, дающим наилучшую оценку. Минимизация методом наименьших квадратов происходит от максимизации гауссовского правдоподобия, и здесь правдоподобие отличается из-за зависимости от  $x_i$  (обсуждается позже). Следует предпочитать прямую максимизацию правдоподобия, даже с учетом того, что метод наименьших квадратов обычно непротиворечив.

Распространенным обобщением AR является также включение компонента *скользящего среднего* (Moving Average, MA). Получается:

$$x_{t+1} = \sum_{0 < i < p} \phi_i x_{t-i} + \sum_{0 < i < q} \theta_i w_{t-i} + w_{t+1},$$

что приводит к модели ARMA( $p, q$ ). Условия шума относятся к модели AR (см. главу 27. *Машинное обучение: регрессия*). Можно создать чистый процесс MA, и в этом случае шумовые члены являются частью данных, не предсказанных предыдущими шумовыми членами. У процессов MA есть плюсы и минусы:

1. Теоретически стабильный процесс MA может быть преобразован в стабильный процесс MA и наоборот. Кроме того, использование обоих вариантов дает некоторые преимущества.
2. Условия шума наблюдаются только в смоделированных данных. Они не могут наблюдаться в реальных данных и требуют оценки. Для оценки необходимо не  $p$  или  $q$  членов, а вся выборка, что делает как оценку, так и прогнозирование с помощью подходящей модели, вычислительно неэффективными и громоздкими.
3. Некоторые источники выступают за использование чистой модели AR вместо модели ARMA или MA. Обычно необходимо иметь значение  $p$  больше  $p + q$ , но это противоречит условию (2).
4. Используемый здесь компромисс состоит в том, чтобы использовать ARMA, но ограничить историю в прогнозировании или расчете остатка до 100 или около того для больших  $n$ . Обоснование состоит в том, что для стабильного процесса влияние любого конкретного члена шума на будущее  $x_{t-i}$  экспоненциально затухает, и 100 шагов обычно достаточно, чтобы сделать это влияние незначительным. Более сложный и затратный в вычислительном отношении метод, такой как удвоение, теоретически может быть более гибким, но далее не рассматривается.

Из-за предполагаемой устойчивости процесса примем также без ограничения общности, что его среднее значение равно нулю. Для достижения соответствия можно предварительно обработать данные, вычтя из них среднее значение выборки и получив тем самым большую точность.

Получение функции правдоподобия ARMA нетривиально и включает в себя множество математических и в целом новых концепций. Литература по временным рядам непоследовательна, т. к. существует несколько способов получить эту функцию: некоторые из них основаны на старых, а некоторые — на более новых концепциях. Чистая, современная презентация на уровне выпускников вузов представлена в работе [21.68], на которой основаны все расчеты. Вот некоторые идеи из этой работы:

1. Точную вероятность рассчитать проблематично, поскольку данные зависят друг от друга. Поэтому ее можно вычислить как условное правдоподобие, логарифм которого  $\sum \ln(\text{прогнозируемое } x_{t-i} | \{x_{t-i}\})$  учитывает предыдущие точки данных. Это проще и в вычислительном отношении эффективнее.
2. Интуитивно понятно, что невязка распределена как  $\text{normal}(0, \text{std})$  по модели Гаусса (в принципе, можно использовать и другие распределения, но обычно это только усложняет задачу). В итоге получаем обычную условную регрессию методом наименьших квадратов. Для модели AR условная вероятность приводит непосредственно к условному методу наименьших квадратов.
3. В моделях MA или ARMA процесс (2) не совсем работает из-за зависимости от более чем  $p$  или  $q$  прошлых наблюдений, которые влияют на шумовые члены. Самая первая (новейшая) точка данных в выборке имеет большое стандартное отклонение, потому что все ненаблюдаемые предыдущие точки данных и шумовые составляющие влияют на нее, к тому же имеется ее собственный шумовой член. Таким образом, квадрат остатка корректируется с коэффициентом  $1/r_t - 1$ , вычисление которого и результирующая вероятность обсуждаются позже. Эти значения велики для первых нескольких последних точек данных и сходятся к 1 для самых последних. Таким образом, качество как условного, так и безусловного метода наименьших квадратов зависит от того, насколько быстро происходит эта сходимость. Естественно, что для больших выборок разница, скорее всего, незначительна, а для малых — нет.
4. Используйте информационные критерии, основанные на правдоподобию, такие как BIC (мы выбираем его, хотя в других источниках рекомендуются AIC и другие). В этом случае вы ничего не потеряете, работая с минимизацией правдоподобия.

Весь код модели ARMA содержится в классе:

```
struct ARMA
{
    Vector<double> phi, theta;
    double std;
public:
    ARMA(): std(1){} // пустой конструктор, для будущего заполнения
    ARMA(Vector<double> const& thePhi, Vector<double> theTheta, double theStd):
        phi(thePhi), theta(theTheta), std(theStd){} // прямая копия
};
```

Полезной операцией является преобразование устойчивого ARMA-процесса в представление МА  $x_{t+1} = \sum_{0 < i < \infty} \psi_i w_{t+1-i}$ , которое обязательно обрезается в некоторой логически заданной точке. Это делается рекурсивно путем решения  $\psi_i = \theta_i - \sum_{0 < k \leq i} \phi_k \psi_{i-k}$ . Здесь  $\phi_0 = 0$ , как и  $\phi$  и  $\theta$  за вычетом  $p$  и  $q$  соответственно:

```
Vector<double> calculatePsi(int maxSize) const
{ // начальное значение веса задается равным 1,
  // а последующие вычисляются
  Vector<double> psi(max<int>(maxSize,
    max<int>(phi.getSize(), theta.getSize() + 1)), 1);
  for(int i = 0; i < psi.getSize() - 1; ++i)
  {
    double sum = 0;
    for(int j = 0; j < min(i + 1, phi.getSize()); ++j)
      sum += phi[j] * psi[i - j];
    double thetai = i < theta.getSize() ? theta[i] : 0;
    psi[i + 1] = thetai + sum;
  }
  return psi;
}
```

Представление МА позволяет вычислить теоретические автоковариации. Вместо решения системы уравнений, предложенной в работе [21.68], используется простая, достаточно хорошая числовая аппроксимация, основанная на уравнении  $\gamma(h) = \sigma^2 \sum_{0 \leq i \leq \infty} \psi_i \psi_{i+h}$ :

```
Vector<double> gammaAll(int h) const
{ // вычисление с использованием численной аппроксимации из psi
  Vector<double> gamma(h, 0), psi = calculatePsi(1000); // достаточно
  for(int i = 0; i < h; ++i) // сортировка от меньших чисел к большим
    for(int j = psi.getSize() - 1 - i; j >= 0; --j)
      gamma[i] += psi[j] * psi[j + i];
  return gamma * (std * std);
}
```

Теперь получим автокорреляцию  $\rho(h) = \gamma(h)/\gamma(0)$ :

```
double rho(int h, Vector<double> const& gamma) const
{
  assert(h >= 0 && h < gamma.getSize());
  return gamma[h]/gamma[0];
}
```

Выполнение прогнозирования с помощью подходящей модели — это простая, но программно сложная операция. В ней неясно, сколько именно исторических данных нужно использовать с моделью МА. Простой ответ состоит в том, чтобы просмотреть любые доступные конечные данные и получить хорошую оценку прошлых значений шума  $q$ . Подробности приведены в работе [21.68].

Помимо естественного прогноза следующего значения, вам нужен доверительный интервал для него и обобщение на  $m$  будущих значений. Приблизительная дисперсия прогноза равна  $\sigma^2 \sum_{0 \leq i \leq m} \psi_i \psi_{i+h}$ . Это связано с тем, что для первого прогноза имеется

только дисперсия шума, но для предсказания будущего появляется дополнительная дисперсия из-за некоторого неизвестного прошлого:

```
Vector<pair<double, double> > operator()( // прогнозы и их отклонения
    Vector<double> const& xPast, int m) const
{ // на основе усеченной оценки
    assert(m > 0);
    int p = phi.getSize(), q = theta.getSize();
    // последнее значение находится по индексу "размер выборки минус один"
    // недоступные прошлые значения задаем равными нулю
    Queue<double> xPastRolling(p), wPastRolling(q);
    for(int i = 0; i < p; ++i) xPastRolling.push(0);
    for(int i = 0; i < q; ++i) wPastRolling.push(0);
    // для предсказания с последними значениями
    double psiSum = 0;
    Vector<double> psi = calculatePsi(m);
    Vector<pair<double, double> > predictionsAndIntervals(m);
    // обрабатываем соответствующие прошлые данные,
    // затем все будущие данные
    for(int i = q > 0 ? 0 : max(0, xPast.getSize() - 1 - p);
        i < xPast.getSize() + m; ++i)
    { // как сборка, так и предсказание
        double xPredicted = 0; // сначала оцениваем предсказание
        for(int j = 0; j < p; ++j)
            xPredicted += phi[p - 1 - j] * xPastRolling[j];
        for(int j = 0; j < q; ++j)
            xPredicted += theta[q - 1 - j] * wPastRolling[j];
        double xi = xPredicted; // случай предсказания
        if(i < xPast.getSize()) xi = xPast[i]; // случай прошлых данных
        else
        { // случай предсказания
            int predI = i - xPast.getSize();
            psiSum += psi[predI] * psi[predI];
            predictionsAndIntervals[predI] =
                make_pair(xPredicted, std * std * psiSum);
        }
        double wPredicted = xi - xPredicted; // равняется нулю в случае прогноза
        // обновляем очереди
        if(p > 0)
        {
            xPastRolling.pop();
            xPastRolling.push(xi);
        }
        if(q > 0)
        {
            wPastRolling.pop();
            wPastRolling.push(wPredicted);
        }
    }
    return predictionsAndIntervals;
}
```



```
// xPast сортируется в обычном порядке, от самого давнего к новому
double operator() (Vector<double> const& xPast) const
{ return operator() (xPast, 1) [0].first; } // одно усеченное предсказание
```

В задаче подгонки модели невязки в литературе называются *инновациями*. Они выводятся из предсказания текущего наблюдения на основе любых предыдущих. Для экономии вычислительных ресурсов ограничьте размер истории:

```
enum{HISTORY_LIMIT = 100}; // эвристика для экономии вычислительных ресурсов
// можно использовать удвоение, но это сложнее
int desiredHistorySize() const
{ // p для моделей AR и как минимум History_LIMIT
  // для моделей MA и ARMA
  return max(phi.getSize() + theta.getSize(),
    theta.getSize() == 0 ? 0 : HISTORY_LIMIT);
}
Vector<double> calculateTruncatedInnovations(Vector<double> const& x) const
{
  Vector<double> innovations = x;
  for(int i = 1; i < x.getSize(); ++i)
  {
    int nPast = min(i, desiredHistorySize());
    Vector<double> xPast(nPast); // nPast значений i - 1
    for(int j = 0; j < nPast; ++j) xPast[j] = x[i - nPast + j];
    innovations[i] -= operator() (xPast);
  }
  return innovations;
}
```

*Функция частичной автокорреляции* (Partial Autocorrelation Function, PACF)  $\phi_{hh}$  представляет собой корреляцию остаточных  $h$  шагов друг от друга. Обозначение перегружено до уровня, аналогичного AR, но различия обычно ясны. Расчет определяется системой линейных уравнений со структурированной матрицей и решается с помощью специального алгоритма эффективности (подробнее об этом в работе [21.68]):

```
Matrix<double> calculatePACFHelper(int n, Vector<double> const& gamma) const
{ // уравнение Дурбина – Левинсона для решения матричного уравнения
  assert(n >= 0 && gamma.getSize() == n + 1);
  Matrix<double> PACF(n + 1, n + 1);
  PACF(0, 0) = 0;
  for(int i = 1; i < n + 1; ++i)
  {
    double sum1 = 0, sum2 = 0;
    for(int j = 1; j < i; ++j)
    {
      sum1 += PACF(i - 1, j) * rho(i - j, gamma);
      sum2 += PACF(i - 1, j) * rho(j, gamma);
    }
    PACF(i, i) = (rho(i, gamma) - sum1) / (1 - sum2);
    for(int j = 1; j < i; ++j) PACF(i, j) =
      PACF(i - 1, j) - PACF(i, i) * PACF(i - 1, i - j);
  }
  return PACF;
}
```

```
Matrix<double> calculatePACF(int n) const
{ return calculatePACFHelper(n, gammaAll(n + 1)); }
```

Значения  $r$  для корректировки правдоподобия вычисляются рекурсивно как  $r_0 = \sum_{0 \leq i \leq \infty} \Psi_i \Psi_i$  и  $r_i = r_{i-1} * (1 - \phi_{ii}^2)$ :

```
double calculateR0() const
{ // бесконечная сумма квадратов psi, сходится или расходится
  // значения 1000 обычно более чем достаточно, поэтому можно
  // использовать удвоение и мониторинг суммы
  Vector<double> weights = calculatePsi(1000);
  double sum = 0; // перебор для численной стабильности
  for(int i = weights.getSize() - 1; i >= 0; --i)
    sum += weights[i] * weights[i];
  return sum; // для расходящегося случая сумма велика или бесконечна
}

Vector<double> rValues(int n) const
{
  double r = calculateR0();
  Matrix<double> PACF = calculatePACF(min<int>(n, HISTORY_LIMIT));
  Vector<double> result;
  for(int i = 0; i < n; ++i)
  {
    if(i < HISTORY_LIMIT)
    { // проверка численных ошибок – процесс
      // может быть недетерминирован
      if(r < 0) return Vector<double>(n,
        numeric_limits<double>::infinity());
      if(i > 0) r *= 1 - PACF(i, i) * PACF(i, i);
    }
    else r = 1; // ограничение времени r
    result.append(r);
  }
  return result;
}
```

Логарифмическая вероятность равна  $\frac{-n}{2} * \log(2\pi\sigma^2) - \sum \log(r_i) - \frac{S}{2\sigma^2}$ , где

$S = \sum \frac{\text{невязка}_i^2}{r_i}$ . Обратите внимание, что  $r$  и  $S$  являются функциями текущих значений

$\phi$  и  $\theta$ , но не  $\sigma$ , если  $\sigma \neq 0$ :

```
pair<double, double> LLHelper(Vector<double> const& x) const
{
  int n = x.getSize();
  double rSum = 0, SSum = 0;
  Vector<double> innovations = calculateTruncatedInnovations(x),
    r = rValues(n);
  if(!isfinite(r[0]) || log10(r[0]) > 3) // эвристическое правило
    // для бесконечных значений
  { // процесс недетерминированный, если r0 расходится
    double inf = numeric_limits<double>::infinity();
```

```

    return make_pair(inf, inf);
}
for(int i = 0; i < n; ++i)
{
    // r[i] = 1; // раскомментируйте, чтобы получить безусловный
    // метод наименьших
    rSum += log(r[i]);
    SSum += innovations[i] * innovations[i]/r[i];
}
return make_pair(rSum, SSum);
}
double LL(Vector<double> const& x) const
{ // для BIC
    pair<double, double> sums = LLHelper(x);
    // не причинный процесс, если r0 расходится
    if(!isfinite(sums.first) || !isfinite(sums.second))
        return -numeric_limits<double>::infinity();
    int n = x.getSize();
    double temp = 2 * std * std;
    return -(log(temp * PI()) * n/2 + sums.first/2 + sums.second/temp);
}

```

Алгоритм можно оптимизировать напрямую, но дальнейшее улучшение за счет некоторой математики (см. [21.68]) состоит в том, чтобы уменьшить *концентрированную вероятность*  $\frac{-n}{2} * \log(2\pi\sigma^2) - \sum \log(r_i) - \frac{S}{2\sigma^2}$ . Тогда  $\sigma = S$  (оптимальные параметры)/ $n$ .

Используйте локальную минимизацию (см. главу 24. Численная оптимизация):

```

double concentratedLL(Vector<double> const& x) const
{ // для МИНИМИЗАЦИИ
    int n = x.getSize();
    pair<double, double> sums = LLHelper(x);
    if(!isfinite(sums.first) || !isfinite(sums.second))
        return numeric_limits<double>::infinity();
    return log(sums.second/n) + sums.first/n;
}
static pair<Vector<double>, Vector<double> > unpackParams(
    Vector<double> const& params, int p)
{ // параметры phi|theta|std
    assert(p >= 0 && p <= params.getSize());
    Vector<double> thePhi(p), theTheta(params.getSize() - p);
    for(int i = 0; i < p; ++i) thePhi[i] = params[i];
    for(int i = p; i < params.getSize(); ++i)
        theTheta[i - p] = params[i];
    return make_pair(thePhi, theTheta);
}
struct LLFuncionr
{
    Vector<double> const& x;
    int phiSize;
    double operator() (Vector<double> const& params) const

```

```

{ // значение std не имеет значения, если не равно 0
  pair<Vector<double>, Vector<double> > result =
    unpackParams(params, phiSize);
  ARMA a(result.first, result.second, 1);
  return a.concentratedLL(x);
}
};

```

Поскольку для численной минимизации требуются начальные значения, простой подход состоит в том, чтобы установить  $\phi$  и  $\theta$  равными 0, что соответствует модели чистого шума. Хорошая и дешевая начальная оценка коэффициентов AR основана на использовании эмпирической автоковариантности  $\hat{y}(h) = \frac{1}{n} \sum x_{i+h} x_i$ . Из нее можно рас-

считать эмпирическую PACF:

```

Vector<double> gammaAllEmpirical(int n, Vector<double> const& x) const
{ // первые n эмпирических автоковариаций в предположении,
  // что среднее значение равно 0
  assert(n > 0 && x.getSize() > 0);
  Vector<double> result(n);
  for(int i = 0; i < min(n, x.getSize()); ++i)
  {
    double sum = 0;
    for(int j = 0; j + i < x.getSize(); ++j) sum += x[j + i] * x[j];
    result[i] = sum/x.getSize();
  }
  return result;
}

Matrix<double> calculatePACFEmpirical(int n, Vector<double> const& x) const
{ return calculatePACFHelper(n, gammaAllEmpirical(n + 1, x)); }

```

В процессе AR из PACF используется линейное уравнение  $\phi_i = \phi_{p,i+1}$ :

```

ARMA(int p, int q, Vector<double> const& x): phi(p, 0), theta(q, 0), std(1)
{ // подгонка модели; нужно больше данных, чем параметров
  assert(p >= 0 && q >= 0 && x.getSize() > p + q + 1);
  if(p + q > 0)
  {
    // неравномерное начальное решение — все равны нулю
    Vector<double> x0(p + q, 0);
    LLFunctor ll = {x, p};
    // для параметра тета используйте оценщик Юле — Уолкера
    if(p > 0)
    {
      Vector<double> x1(p + q, 0);
      Matrix<double> pacfe = calculatePACFEmpirical(p, x);
      for(int i = 0; i < p; ++i) x1[i] = pacfe(p, i + 1);
      if(ll(x1) < ll(x0)) x0 = x1;
    }
    // максимальная вероятность
    pair<Vector<double>, double> result =
      hybridLocalMinimize(x0, ll, 200); // больше не требуется
  }
}

```

```

pair<Vector<double>, Vector<double> > result2 =
    unpackParams(result.first, p);
phi = result2.first;
theta = result2.second;
std = sqrt(LLHelper(x).second/x.getSize());
}
else
{ // модель шума
    IncrementalStatistics s;
    for(int i = 0; i < x.getSize(); ++i) s.addValue(x[i]);
    std = s.stdev();
}
}

```

Для автоматического выбора  $p$  и  $q$  используйте критерии BIC для поиска по всем возможным значениям до некоторых пределов:

```

double BIC(Vector<double> const& x, int extraParams = 0) const
{
    int n = x.getSize(),
        k = phi.getSize() + theta.getSize() + 1 + extraParams; // 1 для std
    assert(n >= 0 && extraParams >= 0);
    return k * log(n) - 2 * LL(x);
}

ARMA(Vector<double> const& x, int maxP = 5, int maxQ = 5): std(1)
{ // Выберите наименьшее значение BIC от 0<=p, q<=maxOrder
    assert(maxOrder >= 0);
    double bestBIC = numeric_limits<double>::infinity();
    for(int p = 0; p <= maxP; ++p)
        for(int q = 0; q <= maxQ; ++q)
        { // реализовать использование предыдущих коэффициентов модели
            // и лучшей для инициализации следующей модели
            ARMA b(p, q, x);
            double bBIC = b.BIC(x);
            if(bBIC < bestBIC)
            { // выполняется хотя бы раз, если данные существуют
                *this = b;
                bestBIC = bBIC;
            }
        }
}
}

```

Требование стабильности очень ограничивает выбор возможных моделей. Например, из-за этого нельзя моделировать тренд или даже просто случайное блуждание. Таким образом, обобщением ARMA является *ARIMA*, где «I» означает «интегрированный», — т. е. прогноз должен отменить любые операции с разностями, примененные к данным, перед подгонкой ARMA. Некоторые вычисления должны быть скорректированы для использования существующего кода ARMA:

1. К данным рекурсивно применяется заданное количество различий  $d$ , что дает модель  $ARIMA(p, d, q)$ . Обычно для практических данных достаточно  $d = 1$ , хотя  $d = 2$  иногда дает лучшие результаты. Большие значения, как правило, даже не рассматриваются. Например, при  $d = 1$  среднее значение результата представляет собой линейный тренд.

2. ARIMA также является хорошим местом для вычитания среднего из разностных данных.
3. BIC с числом параметров в ARMA, увеличенным на  $d$ , по-прежнему позволяет выбирать ту же модель, поскольку в соответствии с лежащим в основе принципом MDL  $d$  — это просто еще один параметр, который необходимо закодировать.
4. Корректировка прогнозов и их ошибок следует из линейности нормальной модели. То есть для  $d = 1$  просто добавьте прогноз ARMA и их дисперсии к последнему известному значению предразличия.

Модель ARI, где  $q = 0$ , скорее всего, в конечном итоге окажется нестабильным процессом AR, поэтому возникает естественный вопрос: почему бы не подогнать процесс AR, чтобы начать с использования условного метода наименьших квадратов и избежать всей сложности. Ответ заключается в том, что оценка для стабильной модели намного точнее из-за рекурсивного характера процесса AR. И этот процесс работает и для более общего ARIMA. Кроме того, стабильный процесс не меняется в пределе, что важно для таких свойств, как непротиворечивость оценок. Прямое соответствие функции данным просто не будет иметь таких свойств, хотя оно, вероятно, тоже будет полезным:

```
struct ARIMA
{
    ARMA a;
    mutable double mean;
    int d;
    Vector<double> transformData(Vector<double>& x,
        bool computeMean) const
    { // разница
        Vector<double> xRemoved;
        for(int j = 0; j < d; ++j)
        {
            xRemoved.append(x.lastItem());
            for(int i = 0; i + 1 < x.getSize(); ++i) x[i] = x[i + 1] - x[i];
            x.removeLast();
        }
        if(computeMean)
        { // оценка среднего
            IncrementalStatistics s;
            for(int i = 0; i < x.getSize(); ++i) s.addValue(x[i]);
            mean = s.getMean();
        }
        // удаление среднего значения из данных
        for(int i = 0; i < x.getSize(); ++i) x[i] -= mean;
        return xRemoved;
    }
public:
    ARIMA(int p, int theD, int q, Vector<double> x): d(theD)
    { // нужно больше данных, чем параметров
        assert(d >= 0 && p >= 0 && q >= 0 && x.getSize() > p + q + d + 3);
        transformData(x, true);
        // подходящий предиктор
        a = ARMA(p, q, x);
    }
```

```

ARIMA(Vector<double> const& x, int maxD = 1, int maxP = 5, int maxQ = 5)
{ // нужно больше данных, чем параметров
    assert(maxD >= 0 && maxP >= 0 && maxQ >= 0 &&
        x.getSize() > maxP + maxQ + maxD + 2);
    double bestBIC = numeric_limits<double>::infinity(), bestMean = 0;
    int bestD = 0;
    for(d = 0; d <= maxD; ++d)
    { // BIC сравнивается с d, потому что сохраняется сжатие MDL
        Vector<double> xCopy = x;
        transformData(xCopy, true);
        // подходящий предиктор
        ARMA aNew = ARMA(xCopy, maxP, maxQ);
        double bic = aNew.BIC(xCopy, d + 1);
        if(bic < bestBIC)
        {
            a = aNew;
            bestBIC = bic;
            bestMean = mean;
            bestD = d;
        }
    }
    mean = bestMean;
    d = bestD;
}

Vector<pair<double, double> > operator()( // прогнозы и вариации
    Vector<double> x, int m) const
{
    assert(x.getSize() >= d + 1 && m > 0); // нужно оставить хотя бы одну точку данных
    Vector<double> xRemoved = transformData(x, false);
    Vector<pair<double, double> > predictions = a(x, m);
    // добавление среднего обратно
    for(int i = 0; i < m; ++i) predictions[i].first += mean;
    // интегрирование
    for(int i = 0; i < d; ++i)
        for(int j = 0; j < m; ++j)
        {
            predictions[j].first +=
                j == 0 ? xRemoved[d - 1 - i] : predictions[j - 1].first;
            predictions[j].second += // сложение отклонений
                j == 0 ? 0 : predictions[j - 1].second;
        }
    return predictions;
}

double operator()(Vector<double> x) const
{ return operator()(x, 1)[0].first; }

double BIC(Vector<double> x) const
{
    transformData(x, false);
    return a.BIC(x, d + 1);
}
};

```

## 21.39. Использование статистики на практике

Для человека с серьезной подготовкой вдумчивое использование статистики требует большего, чем просто критическое мышление и здравый смысл, т. к. многие статистические понятия интуитивно непонятны. Психологически легче рассуждать с точки зрения количества, чем вероятности (см. [21.30]). Нужно быть осторожным, чтобы случайно не солгать статистике, а это случается часто — множество примеров приведено в работе [21.42]. Представьте, что вы получаете набор данных, вычисляете среднее значение и его доверительный интервал на основе  $t$ . Но затем выясняется, что выборки взяты из распределения Коши. То есть сами данные дополнительного контекста не дают (см. [21.40 и 21.15]).

Необходимо понимать цели анализа:

- ◆ могут уже существовать исследования, где были найдены ответы на все вопросы;
- ◆ цели помогают выбрать процедуры и статистики — например, определиться между средним значением и медианой;
- ◆ цели дают некоторое представление о том, сколько данных нужно собрать, и какие переменные надо измерить.

Необходимо знать, что у вас за данные:

- ◆ данные, если они уже собраны, могут содержать неверную информацию или быть получены некорректно. Примеры ситуаций:
  - стандартный набор данных является наблюдательным, а не экспериментальным;
  - измерения могут быть недостаточно точными — например, использование времени в секундах, когда требуется более высокое разрешение;
  - при сборе данных бывают предубеждения — например, со стороны чиновников, которые хотят упростить процесс, или со стороны респондентов, которые не хотят быть честными;
  - данные поступают из сочетания нескольких механизмов генерации и нуждаются в специальной агрегации или вообще не допускают агрегацию. В работе [21.40] утверждается, что редко удастся получить большую однородную выборку;
  - предположение о случайности и независимости может быть неразумным — здесь независимость важнее идентичного распределения, потому что многие последовательности данных из реального мира не обладают независимостью. Кроме того, в таких задачах, как регрессия, в предположение о случайности и независимости заложена часть ошибки аппроксимации. Например, если использовать линейную модель для нелинейного явления с iid шумом, в некоторых областях шума с точки зрения линейной модели будет неестественно больше, чем в других.
- ◆ вы можете получить полезную информацию о вероятном или маловероятном пространстве, но следует остерегаться перехвата данных;
- ◆ ограничения предметной области позволяют обнаруживать невозможные измерения и вероятные выбросы.

Обычно рекомендуется зарезервировать одну часть данных для исследования, а другую — для последующей проверки итоговой модели. Например, когда вы пробуете со-



мнительную процедуру на одной части данных, не загоняйте себя в угол, а постройте другую процедуру из другой их части. Для очень больших наборов данных возможны только автоматический анализ и предварительные проверки, поэтому следует делать все по возможности автоматически.

Наконец, после завершения анализа проверьте результаты на соответствие знанию предметной области:

- ♦ любая статистика — например, конечные точки доверительного интервала, не может выходить за пределы допустимого диапазона значений;
- ♦ вычислительная модель, такая как линейная регрессия, часто не имеет правильного поведения в пределе из-за несоответствия стабильному состоянию реальности. Так, если имеется, может стать проблемой.

Компьютерные специалисты, занимающиеся статистикой, должны помнить, что:

- ♦ статистики заботятся о значении цифр, а не о самих цифрах, поэтому нет смысла выводить значения с огромной точностью;
- ♦ результаты работы алгоритмов важнее самих алгоритмов.
- ♦ данные интересны тем, что они говорят что-то полезное о параметрах модели, а не сами по себе. Например, если одно число выглядит слишком необычно, оно потенциально является выпадающим, но его никогда не заметят, если вы не удосужитесь на него посмотреть.

Статистика уязвима для лжи, потому что заинтересованная сторона будет пробовать множество методов и использовать наиболее благоприятный как единственно верный, проводя многократное тестирование и отслеживание данных. В частности, для публикации исследований общее требование  $p \leq 0,05$  привело к случайной корректировке данных, а воспроизводимость коррелирует с тем, отвечают ли авторы на запросы данных. Такие агентства, как FDA в США, строго контролируют свои процессы, чтобы избежать предвзятости.

Те, кто хочет заниматься статистическим консультированием, должны иметь в виду следующее (см. [21.1]):

- ♦ предметная область клиента, вероятно, будет вам незнакома, и вам придется изучить ее основы хотя бы для того, чтобы обсуждать результаты на одном языке с клиентом;
- ♦ клиенты хотят получить работающее решение, а не лекцию о статистике;
- ♦ часто требуется достаточно быстрое решение, потому что на надлежащее исследование нет времени;
- ♦ коммуникативный аспект, вероятно, самый важный — в частности, вы должны без колебаний критиковать идеи клиента или говорить ему, что существующие данные недостаточно достоверны и потребуют дорогостоящего повторного сбора, чтобы довести их до приемлемых стандартов;
- ♦ во многих случаях требуются сложные или несовершенные экспериментальные схемы для сбора данных (даже нерассматриваемые в книге), особенно если данные поступают от людей.

Во многих приложениях в сфере статистики хорошо себя зарекомендовали общие модели. Но для достижения наилучших результатов необходимо исследовать специализи-

рованную литературу, в которой используются предметно-ориентированные знания. Например, значимость совпадения отпечатков пальцев сильно зависит от предварительных знаний о таких отпечатках, и «разумные» предположения, не зависящие от предметной области, бесполезны.

## 21.40. Анализ решений

В задаче принятия решения важно различать *решение* и *результат* (см. [21.37]). Каждое из них может быть хорошим или плохим независимо от друг от друга. Например, поставить все свои сбережения на рулетку с шансом 50/50 и выиграть — это плохое решение (при условии, что вы не склонны к риску, как и большинство людей) с хорошим исходом.

Математически наилучшие решения максимизируют ожидаемую ценность, предполагая тождественные функции полезности (см. главу 24. *Численная оптимизация*). Но ожидаемые значения могут быть обманчивы — например, очень редкое событие с огромным ожидаемым выигрышем кажется привлекательным, но на практике оно никогда не произойдет. Определить полезность сложно, поскольку разные вещи для разных людей в разное время могут иметь разную ценность. Например, купленная вами книга стоит для вас меньше, когда вы ее уже прочитали. Также ценность может уступить место риску — например, при инвестировании нужно отказаться от некоторой ожидаемой прибыли, чтобы уменьшить ожидаемый риск.

Почти все рассуждения происходят на байесовском уровне убеждений, который определяет обычно надежные решения, за исключением случаев, когда расчет вероятностей не выполняется. Несмотря на обширные разработки в логике, статистике, теории игр, экономике и т. п., к рассуждениям, основанным на инстинктивных убеждениях, все еще прибегают весьма часто. Тем не менее известно, что интуитивные человеческие суждения иррациональны, если в них преобладают психологические эффекты, поэтому, например, для принятия важных решений требуется нечто большее, чем просто хорошее чутье. Большинство решений, как правило, существенно зависит только от нескольких переменных. Но обычно лучше инвестировать в получение большего количества информации, чем больше думать с существующей информацией.

Еще одно инстинктивное понятие — *сожаление*. Люди ненавидят ощущение, что принятое решение привело к упущению лучшей возможности. В большинстве случаев решения не несут заметных издержек, поэтому аргументация на основе убеждений работает хорошо. Но для принятия серьезных решений — таких как выбор карьерного пути, приходится подумать получше. Рациональное решение в возможно иррациональном мире убеждений обычно не приводит к сожалениям, но только в том случае, если решение и результат должным образом разделены.

Рассмотрим задачу распределения инвестиций у физического лица. Эмпирическое правило состоит в том, чтобы повторять стратегию пенсионных фондов. Следует инвестировать диверсифицированно, выбирая не одну акцию или облигацию, а многие, используя так называемые *фонды, торгующиеся на бирже* (ETF, Exchange traded fund — публичный инвестиционный фонд, который выставляет для продажи на бирже универсальные акции, состоящие из долей разных компаний). Это снижает риск от единичной инвестиции до рыночного. Даже интуитивно вы не пожалеете о совершении каких-либо одиночных инвестиций. Еще одно распространенное решение — когда

адаптироваться к конкретной новой технологии. Лучше всего проводить осторожную политику — пусть сначала новинку оценят другие. Они могут выиграть в лотерею, но ожидаемый выигрыш обычно лучше, если подождать.

Решения не должны быть оправданными. Соблазнительно иметь возможность защищать многие решения, особенно с научной точки зрения, но во многих случаях приходится принимать решения быстро. В результате длинной череды суждений все стало так, как оно есть, потому что так оно и было. Например, большинство аспектов нынешней судебной системы США были определены сотни лет назад без каких-либо формальных инструментов принятия решений или с использованием лишь минимальных средств — таких как простая логика. Но мысль о том, чтобы присяжные рассматривали доказательства и принимали окончательное решение на основе своей веры, — это и есть эвристика.

В некоторых случаях переменные, которые необходимо измерить, не могут быть измерены напрямую. Хитрость заключается в использовании косвенных измерений — т. е. в измерении связанных величин и использовании взаимосвязей для вывода оценки. В качестве примера можно привести вопрос с собеседования о том, сколько стоит сдвинуть гору Фудзи. Можно использовать ее высоту и радиус окружности, чтобы оценить ее объем, плотность камня, чтобы оценить вес, и тарифы грузоперевозок, чтобы оценить стоимость. Учитывая байесовский априор для каждого из них, можно даже получить достоверный интервал результата. Более реальной задачей является измерение ценности хорошего обслуживания клиентов для компании. Необходимо изучить известную информацию — такую как уровень продаж довольным клиентам и пр. Основная трудность заключается в поиске достаточно полезной косвенной информации (подробнее об этом см. в работе [21.39]).

Важной моделью для последовательных решений является *задача секретаря*. Предположим, вы проводите по одному собеседованию со 100 кандидатами на работу, должны дать каждому ответ «да/нет» и не можете изменить решение. Математически (см. [21.92]), чтобы максимизировать ожидаемое качество нанятого человека, оптимальная стратегия состоит в том, чтобы отклонить первые  $100/e$  процентов кандидатов и нанять первого, который лучше любого из них, или последнего, если до него дойдет. Таким образом, при  $n = 100$  первые 37 отбрасываются. Эта логика распространяется на многие другие ситуации — в частности, на временной диапазон за счет предельной дискретизации. Например, если вы хотите купить дом в течение следующих трех лет, в течение последних двух лет покупайте первый дом, который окажется лучше, чем любой виденный в первый год. Но можно выбрать более надежную стратегию и *не искать лучший вариант*, если очень хороший найдется быстро.

Многие организации подают иски в корыстных целях. Ключевым аспектом является *фальсифицируемость* — большинство таких утверждений нельзя опровергнуть, потому что это невозможно или требует слишком большого числа ресурсов.

Некоторые решения имеют временной компонент — например, выбор может зависеть от того, над какой задачей работать дальше. Когда вы завалены задачами, полезной стратегией является *диаграмма Эйзенхауэра* — классифицируйте задачи на важные/неважные и срочные/несрочные (см. [21.94]). Важные и срочные получают приоритет, затем выполняются важные и несрочные, затем остальные. Как бы ни было заманчиво выполнить сначала неважные и несрочные, этого лучше не делать и расставить приоритеты правильно.

Большая часть логики принятия решений исходит из экономики, особенно в отношении выбора продуктов среди множества вариантов. Модель *монополистической конкуренции* предполагает, что все товары немного отличаются друг от друга, поскольку разные производители почти никогда не делают абсолютно одинаковый выбор. Это, как правило, на практике верно. Даже обычное мыло отличается кусок от куска: некоторые предназначены для детей, некоторые различаются по запаху, некоторые содержат определенные минералы и другие натуральные ингредиенты. Поэтому наряду с ценой учитывайте *особенности* различных предметов. Некоторые особенности имеют значение, а другие — нет, и переплачивать за последние не хочется. Технически правильный подход состоит в том, чтобы сформировать *Парето-оптимальную границу* (см. главу 16. *Комбинаторная оптимизация*) товаров, не уступающих другим по желаемым свойствам и цене, и выбрать любой из них.

Предположение о случайности и независимости во многих практических случаях неоправданно. Например, иногда вы перед покупкой пытаетесь определить качество продукта по средним отзывам. Здесь присутствует состязательный компонент, который фактически представляет собой смесь распределения справедливых оценок, оценок заинтересованных сторон и оценок конкурентов. Нетрудно создать 50 или около того отличных оценок для заинтересованной стороны.

Другие стратегии принятия решений в экономической сфере задействуют модели из теории игр. Типичным случаем, который не моделируется математически, является игра с нулевой суммой между очень опытными профессионалами — такими как вымышленные Дамблдор и Гриндевальд или реальными шахматистами, боксерами, теннисистами и профессиональными переговорщиками. В любой из этих ситуаций любой сильный игрок может вести разумную игру против любого гораздо более сильного игрока, хотя, скорее всего, в конечном итоге проиграет. Но потеря, наверняка, произойдет из-за множества мелких ошибок, а не из-за одной крупной. Со статистической точки зрения, чтобы более слабый игрок не позволил «намного более искусному», более сильному игроку материализоваться в выигрыше, нужно добавить больше дисперсии, чтобы создать неопределенность в отношении того, какое ожидаемое значение больше. Точно так же более сильный игрок делает более безопасные ходы с низкой дисперсией (я убедился в этом на многих шахматных партиях с гроссмейстерами, но тут есть и существенный психологический эффект — такой как страх за более слабого игрока и стремление к определенности за более сильного). Но это, пожалуй, все, что можно сказать о применении статистического и байесовского мышления к такого рода моделям. В лучшем случае вы можете создать некий вероятностный профиль поведения человека применительно к конкретной ситуации.

## 21.41. Примечания по реализации

Говоря об оценщиках, сложнее всего реализовать выборочный квантиль из-за выбора, который необходимо сделать при решении вопросов о равенстве. Моя реализация самая простая и не хуже любой другой.

В задачах определения доверительного интервала процедуры начальной загрузки, безусловно, являются наиболее сложными из-за выбора процедуры и требуемой логики надежности. Я даже не смог выбрать единственную наилучшую процедуру среди интер-

валов 2-го порядка, хотя было достаточно ясно, что смешанный интервал по умолчанию лучше других.

Реализации инвертирующих перестановочных тестов при формировании доверительных интервалов с использованием обычных случайных чисел для получения непрерывности несколько эвристичны, но, по-видимому, оригинальны.

Другие процедуры точно следуют описаниям в учебниках или в научной литературе. Основная трудность заключалась в том, что почти по каждой отдельной подтеме требовалось исследовать очень много источников.

## 21.42. Комментарии

Наиболее формальный подход к вероятности — использование *теории меры*. Не всем подмножествам универсального множества можно присвоить вероятность, поэтому ограничьте разрешенные события большой коллекцией — например, всеми интервалами. На практике это незаметная формальность, которая в большинстве задач не нужна. По сути, все результаты остаются теми же, просто формулируются в более общем виде. Одно из основных различий состоит в том, что каждое распределение представляет собой смесь непрерывного, дискретного и дельта-распределения, причем некоторые смеси сложнее других. Но важно знать теорию, особенно, если вы хотите изучать исследовательские работы или математические доказательства. Большинство основных идей статистики появились до того, как была формализована теоретико-мерная вероятность, и были формально переформулированы, хотя иногда это было трудно. Лучше всего освоить следующие курсы по порядку:

1. Статистика бакалавриата: [21.82, 21.79, 21.46 и, возможно, 21.59].
2. Теория меры: [21.32, 21.56 и 21.45].
3. Теоретико-мерная вероятность: [21.61, 21.25 и 21.75].
4. Теоретико-мерная статистика:  
[21.70 (см. также лекции автора: <http://pages.cs.wisc.edu/~shao/>), 21.51 и 21.20].

Среди множества других вариантов перечисленные здесь книги я нашел наиболее читабельными, но простых книг по этой теме нет, особенно в пункте 4. Основная идея состоит в том, что нужен критерий эффективности, не взломанный *сверхэффективными оценщиками*, которые имеют хорошее поведение в нужной точке за счет соседних точек. Попробуйте маленькую асимптотическую окрестность размером, может быть,  $1/\sqrt{n}$ , с поправкой на конкретную ситуацию. Как подсказывает здравый смысл, такая стратегия дает лучших оценщиков. Хорошо написанное введение приведено в первых главах книги [21.47]. К сожалению, эта теория обычно представлена на гораздо более абстрактном уровне, возможно, полезном только для тех, кто обожает теоретизировать. На этом мои собственные знания, по сути, заканчиваются, поэтому какие-либо другие книги рекомендовать я не стану.

Тезис о том, что Крамер-Рао ограничивает снизу другие функции потерь при некоторых условиях, сформулирован мной, хотя и сам собой очевиден.

Еще один способ агрегирования риска, который больше похож на численный анализ с полиномами, представляет собой что-то вроде байесовского риска, но с использова-

нием квадрата потерь. Это позволяет придать большим значениям больший вес. Возможны и многие другие расширения, но они в литературе не исследованы.

(Двустороннее) *эмпирическое неравенство Бернштейна*, которое применяется в тех же случаях, что и неравенство Хёффдинга, дает доверительный интервал

$$\mu \in \bar{x} \pm \frac{\sqrt{2(n-1)s \ln(3/p) + 3 \ln(3/p)}}{n} \quad (\text{см. [21.5]}). \text{ Разница заключается в члене } O(n^{-1}),$$

и она довольно велика. Например, при выполнении моделирования с низкой дисперсией Бернулли (0,9) нужно значение  $n \approx 600$ , чтобы эффективность оказалась лучше, чем у Хёффдинга. Здесь даже при  $n = 10^6$  у последнего интервал оказывается длиннее всего на  $\approx 55\%$ , а при  $n = 30$  — на  $56\%$  короче. Для переменной с ограниченным диапазоном и таким большим  $n$  даже диапазон CLT оказывается очень точен. Таким образом, эмпирическое неравенство Бернштейна практически бесполезно (теоретически из-за того, что с ним труднее работать в доказательствах).

В работе [21.21] идея оценки на основе расстояния выводится на новый уровень и предлагаются *интервалы аппроксимации* такие, что только значения, которые проходят DKW или аналогичные тесты гипотез на основе расстояния, должны рассматриваться как формирующие действительный интервал. В отличие от доверительного интервала, здесь учитывается ошибка аппроксимации, поэтому у моделей с большей ошибкой аппроксимации вычисленный интервал может не содержать значений. Хотя это концептуально полезно и позволяет получить ответы на многие вопросы о достоверности регулярных доверительных интервалов в не совсем правильных моделях, существуют трудности с вычислениями:

- ◆ интервалы аппроксимации рассчитываются численно;
- ◆ хороших критериев расстояния для  $D > 1$  не существует;
- ◆ применение бинарного поиска с генерацией случайных данных приводит к образованию в интервалах дыр, если не использовать обычные случайные числа.

Основная польза от этого предложения состоит в том, чтобы рассматривать прогностические свойства моделей, а не доверительные интервалы, как это делается в машинном обучении.

В определенных приложениях полезно использовать *интервалы прогноза и допуска*. Например, в задачах поиска направления движения обычно получают диапазон того, сколько времени может занять поездка. Он не равен доверительному интервалу. При изучении производительности рандомизированных алгоритмов — таких как хеш-таблицы, разница между возможными значениями и средними значениями очевидна, а неопределенность среднего значения намного меньше. Интервал допуска проще для понимания. Например, в случае построения маршрута можно использовать 10-й и 90-й квантили, чтобы узнать продолжительность поездки в 80%-ном интервале прогноза/допуска. Если интервалы известны точно, то они окажутся одинаковыми. Но если оценивать квантили по недавней активности вождения, можно использовать соответственно нижнюю и верхнюю односторонние доверительные границы, добавив поправку на множественность. Это даст 80-процентный интервал допуска с 95-процентной достоверностью. Интервал прогнозирования будет другим, потому что он должен пересчитываться для каждой новой точки данных, а ошибка 80% будет применяться к предвзвешенной ошибке тестирования в реальном времени. Она сложнее для понимания, менее гибкая и часто менее полезна, несмотря на отказ от нее с некоторым уровнем

достоверности (см. [21.57], дополнительную информацию о тестировании в реальном времени можно найти в *главе 25. Основы машинного обучения*). Конкретные способы вычисления обоих типов интервалов приведены в работах [21.33 и 21.48].

Идея использования нормальной, чебышевской и экспоненциальной функций потерь для доверительных интервалов является оригинальной. Вы можете выполнить умноже-

ние чебышевских потерь на  $l = \frac{\sqrt{1/a_{\text{target}}}}{\text{точная длина}}$ , где целевое значение для более четкой

интерпретации выбирается равным 1, но это не имеет значения для сравнения. Потери, по Чебышеву, следуют другой идее: если процедура постоянно пропускает  $a$ , ее также можно запускать с *дробным значением*  $a$  — например  $1/2$  или другим значением, которое выбирается в зависимости от пропуска. Не существует хорошо обоснованного способа выбора дробной доли без знания предметной области. Также для нормального предположения, применительно к которому это наиболее полезно, хвосты с меньшей вероятностью будут точными, поэтому, например, при половинном уровне и 95%-ной достоверности величины  $z \approx 2,25$  может быть недостаточно, чтобы иметь значение.

Для вычисления доверительного интервала корреляции Пирсона может быть полезно дальнейшее уменьшение смещения в асимптотическом приближении. В работе [21.20] приведена более подробная информация.

Оригинальной является идея начальной загрузки смешанного интервала Чебышева, который обобщает интервал Вальда даже для не связанных с начальной загрузкой процедур. Использование неравенства Чебышева для ограничения начальной загрузки и определения диапазона бинарного поиска при калиброванном интервальном поиске также является оригинальной идеей. Вы можете рекурсивно повторять калибровку с уже откалиброванным интервалом с целью улучшения порядка, но ошибка оценки и увеличение вычислительных затрат не позволяют использовать этот метод.

Во многих источниках приводятся проверки гипотез только для принятия/отклонения нулевого значения без вычисления  $p$ -значений с использованием таблиц значимости, которые представляют допустимые статистические значения для определенных уровней достоверности.  $p$ -значения можно вычислить с помощью интерполяции (см. *главу 23. Численные алгоритмы: работа с функциями*), но обычно  $p$ -значение можно получить напрямую с помощью немного другого метода. Доверительные интервалы, полученные из тестов, задаются непосредственно таблицами.

Еще один подход к проверке гипотез называется *проверкой эквивалентности*. Допустим, нам нужно показать, что препарат дженерик по лечебному эффекту сопоставим с брендовым. Null-значение состоит в том, что они различаются. Требуется выбрать специфичную для предметной области точность  $\epsilon$  и показать, что эффективность альтернативы находится в пределах  $\theta \pm \epsilon$ , где  $\theta$  — производительность «золотого стандарта». Null отклоняется, если рассчитанный доверительный интервал производительности  $\in \theta \pm \epsilon$ . С точки зрения проверки гипотез метод отличается, потому что null находится за пределами  $\theta \pm \epsilon$ . *Проверка не меньшей эффективности* является односторонней проверкой эквивалентности и для статистики местоположения эквивалентна обычному одностороннему нулевому тестированию за счет сдвига статистического значения теста на  $\epsilon$ . Это также позволяет повторно использовать известные тесты для проверки эквивалентности, выполняя два односторонних теста при  $a/2$  и возвращая их максимальное значение  $p$ .

*Метаанализ* объединяет результаты нескольких исследований, когда не может напрямую объединить необработанные данные. Комбинация обычно выполняется с использованием *нормализованной величины эффекта* — например,  $z$ -показателя. Для непараметрических  $p$ -значений в качестве простой эвристики можно объединить несколько независимых исследований путем преобразования в эквивалентные  $z$ -показатели, вычисления распределения суммы (которое также является нормальным) и преобразования его  $z$ -показателя в  $p$ -значение. Основная проблема заключается в предвзятости публикаций, т. к. в основном публикуются исследования с низкими  $p$ -значениями. Так что любая комбинация, вероятно, будет предвзятой, и исправить это никак нельзя. Разумнее было бы выбрать одно исследование на основе размера выборки и того, насколько хорошо оно было проведено, и исходить из доверия к нему. Систематическая ошибка публикации менее важна для доверительных интервалов.

При проверке гипотез с использованием начальной загрузки можно проверить, находится ли нулевое значение за пределами вычисленного интервала. Специфичная для предметной области, но более мощная альтернатива состоит в явном формировании нулевого распределения (см. [21.24]). Эти методы работают хуже, чем перестановки, и применяются в том же круге задач. Кроме того, они автоматизированы.

Существует еще один общий, основанный на вычислениях метод под названием *метод эмпирического правдоподобия* (см. [21.78]). Для его использования требуются статистические данные, которые являются функциями взвешенных наблюдений с весами  $p_i \geq 0$ , такими что  $\sum_i p_i = 1$ .  $\prod_i p_i$  — это *эмпирическая вероятность*, вычисляемая по EDF. Учитывая нулевое  $\theta$ , требуется максимизировать эту вероятность так, чтобы статистика( $p$ , данные) была равна  $\theta$ . Без нулевого ограничения это происходит, когда  $p_i = 1/n$  с вероятностью  $n^{-n}$ . Тогда, как и в случае с параметрическими тестами отноше-

ния правдоподобия, логарифмическое отношение правдоподобия 
$$\frac{\sum_i \log(p_i)}{-n \log(n)} \sim \chi^2$$

квадрат с некоторыми степенями свободы при  $n \rightarrow \infty$ . Это определяет проверку гипотез, инвертируя которые, можно получить доверительный интервал. Но, согласно данным некоторых экспериментов (см. [21.102]), этот метод имеет ограниченную применимость. Учитывая недавнее исследование его применения к другим проблемам (см. [21.78]), для обоснования общего применения необходимы дополнительные исследования. Очевидную трудность представляет необходимая числовая оптимизация — область ограничения выпуклая, но ваша цель может быть и другой, в зависимости от выбора статистики. Null отклоняется, если оптимизация не удалась, но для этого требуются специальные методы оптимизации из конкретной выбранной статистики, поскольку методы «черного ящика» ненадежны (см. главу 24. *Численная оптимизация*). Таким образом, приближенная оптимизация даст несколько или существенно более широкие интервалы.

В случае перестановочных тестов, поскольку одно моделирование производит событие «да/нет», оценочное значение  $p$  имеет биномиальное распределение, поэтому интересная идея заключается в использовании доверительных границ 95% вместо среднего значения, чтобы получить более консервативное значение  $p$ . Но хотя идея привлекательна, метод не имеет точного обоснования.

Критерий Неменьи (его не всегда так называют) обычно представляет собой *апостериорный критерий* для *теста Фридмана*, который проверяет, одинаковы ли все  $k$  альтер-



натив, путем вычисления суммы квадратов разностей между суммами рангов и их ожидаемыми значениями (см. [21.89 и 21.28]). Но результат бесполезен, несмотря на мощность большую, чем в отдельных сравнениях с поправкой на множественность. Набор параметрических тестов для этого случая называется *ANOVA*. Несмотря на широкое использование множества его версий (у Фридмана это *повторяющиеся измерения*), он требует применения множества предположений и лишь немногим более эффективен, чем вариант Фридмана. Аналогично *тест Тьюки HSD* является альтернативой критерию Неменьи. Для различных параметрических настроек было разработано много специальных тестов. У них есть преимущество — они вносят менее радикальные поправки, чем тест Бонферрони (см. [21.83]).

Для несогласованных выборок, чтобы получить эквивалент Неменьи, надо вычислить ранги на основе *теста Крускала — Уоллиса* (см. [21.28]). В обоих тестах объединяются все выборки и проверяются различия между средними рангами отдельных групп. Но использование усеченных средних приводит к значимым доверительным интервалам, работая при этом столь же надежно.

Аналогично *FDR коэффициент ложного охвата* дает многократное улучшение при тестировании доверительных интервалов (см. [21.23]).

Использование поиска по сетке с *RWM* — тоже оригинальная идея. Еще один популярный многомерный метод — *семплер Гиббса*, но для него нужны некоторые знания о распределении, и он не всегда применим.

В литературе по оптимизации моделирования описано несколько конкурентов ОСБА, которые также являются *последовательными методами*. Здесь выводы делаются по мере поступления данных в реальном времени или пакетами. Одним из преимуществ этого метода является, например, использование первой ступени для управления *FDR* и второй ступени — для *FWER*. Для остановки моделирования первый этап используется для оценки мешающих параметров, таких как дисперсия, а второй — для основной работы. Остановка в реальном времени позволяет в особых случаях создать более экономичный дизайн. Последовательные методы не популярны, но в работе [21.55] о них рассказано подробнее.

В производстве основная идея *статистического контроля качества* заключается в отслеживании отклонений и вмешательстве в случаях, когда они становятся достаточно высокими, чтобы порождать дефекты. Для этого нужны специальные инструменты. Существует простой метод — *контрольная диаграмма*, которая отслеживает наблюдения за пределами трех стандартных отклонений от среднего значения, что соответствует методу обнаружения выбросов. Общая идея состоит в том, чтобы определить метрики, оценить текущую производительность и использовать план экспериментов для повышения производительности. Наша цель — сделать так, чтобы большая часть производства составляла *шесть сигм*, что означает очень маленькую поправку на вероятность отказа. Идея в том, что если собрать много независимых компонентов, то общая вероятность отказа из-за многократного тестирования должна быть небольшой. Это достигается за счет уменьшения дисперсии.

Общие методы не подходят для *оценки экстремальных значений* (см. [21.19]). Типичным примером является оценка высоты плотины, с высокой вероятностью достаточной для предотвращения наводнений в течение нескольких столетий, при измерениях уровня воды за несколько лет. Моделирование/повторная выборка здесь не сработают, по-

тому что экстремальное проявление, которое может появиться в будущем, перевешивает все, что было получено ранее. Это некорректно поставленная проблема. Разумно предположить, что уровни воды подчиняются нормальному или какому-либо другому распределению, оценить его параметры, смоделировать процессы в течение нескольких столетий и взять 99% или около того. Это еще один пример того, что проблему можно решить с помощью той или иной формы моделирования.

Выборки специально отбираются для вычисления некоторого интересующего количества. Типичный пример задачи — оценка количества зараженных деревьев в большом лесу. Невозможно проверить каждое дерево, поэтому случайным образом нужно выбрать несколько участков площадью в квадратную милю, подсчитать зараженные деревья в них, выявив тем самым ожидаемое количество зараженных деревьев на квадратную милю, и умножить его на общее количество квадратных миль, чтобы оценить итоговое количество зараженных деревьев. Единственная хитрость заключается в учете конечной популяции (квадратных миль) для уменьшения дисперсии (см. [21.77]). Если заранее известно, что на некоторых участках мало деревьев, а на некоторых много, можно уменьшить дисперсию, используя *стратифицированную выборку*, — взять больше выборок и взвесить их, чтобы учесть это в расчетах. Это позволяет уменьшить дисперсию и работает аналогично выборке по важности. Существует много способов сделать это (см. [21.77]).

*Исследовательский анализ данных* (Exploratory Data Analysis, EDA) стремится представить данные в интуитивно понятном виде. Идея состоит в том, что, имея визуальное представление о данных, можно легко проверять предположения, выявлять необычные закономерности или выпадающие значения и т. д. Это, скорее, философия, чем конкретный набор методов. Был разработан ряд методов визуализации — таких как классический график *ящик с усами*. Некоторые затратные с точки зрения вычислительных ресурсов современные методы — такие как кластеризация, анализ основных компонентов и оценка плотности, также могут считаться исследовательскими. EDA предназначен для интуитивного понимания данных, в отличие от традиционного подтверждающего анализа, который фокусируется на получении статистически значимых выводов. В условиях ограниченного времени корпоративной работы традиционный экспериментальный план позволяет легко представить результаты работы, что дает важное практическое преимущество.

EDA часто бывает полезен для решения некорректных задач. Следует опробовать несколько алгоритмов, каждый со своими предположениями, и посмотреть, чем в итоге будут отличаться выходные данные. А затем применить подтверждающую технику, чтобы увидеть, являются ли значимыми полученные наблюдения. Но следует быть осторожным и использовать разнообразные данные. Метод EDA критикуют за то, что он недостаточно автоматизируем и субъективен. Кроме того, при слишком большом объеме данных человек не способен сам выполнить визуализацию, а алгоритмы справляются лучше. Следует прибегать к EDA только в том случае, если известные алгоритмы для решения проблемы недостаточно хороши и ожидается, что потребуется человеческий взгляд.

Многие «аналитики больших данных» подсчитывают события и используют подсчеты для вычисления других статистик.

Алгоритм Сальтелли также позволяет оценивать чувствительность взаимодействий. При формировании  $XC$  нужно брать все взаимодействующие переменные из  $X4$ . Но

неясно, на каком из  $D^2$  взаимодействий нужно сосредоточиться. Для больших  $D$  можно использовать основные эффекты в априорном (см. главу 29. *Машинное обучение: другие задачи*) процессе для выбора  $O(D)$  взаимодействий. Алгоритм может быть расширен вычислением *индексов общей чувствительности*, которые измеряют вклад переменной из-за всех взаимодействий с другими переменными.

ARIMA — самая популярная модель для анализа временных рядов, но существуют многие другие модели и расширения:

- ◆ *модели экспоненциального сглаживания* работают аналогично, но не полностью включены в ARIMA (0, 1, 1) (см. [21.43]). Они представляют собой средневзвешенные значения последних наблюдений и хорошо работают для небольшого количества обучающих данных (см. [21.16]);
- ◆ *многомерные модели* работают для многих коррелированных временных рядов с общей корреляционной матрицей и являются прямым обобщением ARIMA;
- ◆ *модели пространства состояний* можно использовать для выражения всего отмеченного и выполнения вычислений с использованием *фильтра Калмана*;
- ◆ *нелинейные модели* выходят за рамки линейной модели в пространстве состояний, но еще неизвестно, работают ли они лучше на практике. Существуют также модели ARCH и GARCH, которые допускают непостоянную дисперсию, хотя в остальном являются линейными.

## 21.43. Советы по дополнительной подготовке

- ◆ Изучите и внедрите методы оценки CDF для других распространенных распределений. Используйте их для тестирования соответствия распределения и генераторов случайных чисел. В качестве вспомогательной функции реализуйте численно безопасное вычисление биномиального коэффициента.
- ◆ Если невозможно определить асимптотическую дисперсию аналитически, можно смоделировать ее с удвоением размера, а затем выполнить регрессию в логарифмическом масштабе, чтобы определить скорость уменьшения. Поэкспериментируйте с этим.
- ◆ Реализуйте корреляцию Кендалла и сравните ее с Пирсоном и Спирменом на смоделированных данных.
- ◆ Если среднее значение корреляции известно, можно попробовать реализовать корреляцию знаков, т. е. вычисление среднего числа совпадающих ходов. Поэкспериментируйте с этим.
- ◆ Начальная загрузка не устойчива к малым  $n$ , а чрезмерное дублирование приводит к тому, что у функции  $f$  возникает неопределенное поведение (например, корреляция со всеми данными, являющимися одной и той же точкой). Измените доверительные интервалы начальной загрузки, чтобы в таких случаях разрешались значения NaN, и отбросьте их.
- ◆ Реализуйте начальную загрузку- $t$  со средним абсолютным отклонением вместо стандартного отклонения. Она должна быть более устойчивой к большим значениям. Этому способствуют даже небольшие отклонения от нормы (по крайней мере,

в случае загрязненной нормальной модели — см. [21.41]). Улучшается ли от этого производительность?

- ◆ Повторите моделирование начальной загрузки с меньшим значением  $a$  — например, 0,01 и 0,001. Работают ли процедуры по-прежнему должным образом? Изменится ли принцип выбора?
- ◆ Используйте мультिवыборочную начальную загрузку, чтобы создать доверительный интервал для разницы медиан. Сравните ее с простой разницей доверительных интервалов медианы одной выборки.
- ◆ Протестируйте процедуры доверительных интервалов для метрик местоположения с использованием распределения Лапласа. Это должно способствовать обращению перестановочного теста из-за симметрии распределения. Так ли это?
- ◆ Используйте начальную загрузку для изучения эффективности выборочного среднего, медианы и усеченного среднего на нескольких не слишком малых наборах реальных данных путем вычисления и сравнения стандартных отклонений оценок повторных выборок. Дает ли предположение о нормальности какие-то преимущества?
- ◆ Протестируйте другие методы начальной загрузки 2-го порядка с большими  $n$ .
- ◆ Для разницы средних значений двух независимых выборок узнайте, что работает лучше — CLT, начальная загрузка или перестановочное тестирование? А как насчет медианы и усеченного среднего?
- ◆ У критерия Неменьи есть проблемы независимости и случайности, т. к. добавление или удаление другой альтернативы иногда может повлиять на результат сравнения. У тестов с парными знаками такой проблемы нет. Важно ли это на практике? Проведите моделирование, чтобы сравнить два подхода.
- ◆ Для сравнения множества независимых выборок вместо вычисления различий  $k$  доверительных интервалов узнайте, не оказывается ли статистически более эффективным вычисление интервалов на  $\frac{k(k-1)}{2}$  различиях. Проведите моделирование для медиан и нормального распределения, чтобы выяснить, при каком  $k$  это дает преимущества. Сделайте то же самое для Коши. Зависит ли что-то от значения  $a$ ?
- ◆ Выполните моделирование для всех представленных процедур доверительного интервала, чтобы изучить значения  $a$ .
- ◆ Исследуйте поправку на непрерывность для интервалов квантилей и баллов Вильсона. Повышается ли точность в обоих случаях?
- ◆ Выполните моделирование анализа мощности для доверительных интервалов на медиане для нормального распределения. Какого размера выборки обычно достаточно для некоторого не слишком короткого интервала? Попробуйте сделать то же самое для некоторых двух выборочных интервалов. Какое моделирование выполняется проще?
- ◆ Реализуйте генерацию дополнительных примитивных полиномов для последовательности Соболя. Для этого потребуется почитать книги по теории чисел.

- ◆ Для семплера Соболя реализуйте общую функциональность преобразования, которая позволяет, например, семплировать диапазон логарифмической шкалы как частный случай.
- ◆ Примените анализ чувствительности к некоторым моделям классификации (см. главу 26. *Машинное обучение: классификация*). Помогает ли это в задаче выбора переменных?

## 21.44. Список рекомендуемой литературы

- 21.1. Adèr H. J., Mellenbergh G. J., & Hand D. J. (2008). *Advising on Research Methods: A Consultant's Companion*.
- 21.2. Johannes van Kessel Publishing (can order from <https://jvank.nl/ARMHome>).
- 21.3. Andrews D. W. (2000). Inconsistency of the bootstrap when a parameter is on the boundary of the parameter space. *Econometrica*, 68(2), 399–405.
- 21.4. Anguita D., Ghelardoni L., Ghio A., & Ridella S. (2013). A survey of old and new results for the test error estimation of a classifier. *Journal of Artificial Intelligence and Soft Computing Research*, 3(4), 229–242.
- 21.5. Audibert J. Y., Munos R., & Szepesvári C. (2009). Exploration–exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science*, 410(19), 1876–1902.
- 21.6. Basu A., Shioya H., & Park C. (2011). *Statistical Inference: The Minimum Distance Approach*. CRC.
- 21.7. Beisbart C., & Saam N. J. (Eds). (2018). *Computer Simulation Validation: Fundamental Concepts, Methodological Frameworks, and Philosophical Perspectives*. Springer.
- 21.8. Bicke, P. J., & Freedman D. A. (1981). Some Asymptotic Theory for the Bootstrap. *The Annals of Statistics*, 1196–1217.
- 21.9. Boos D. D., & Stefansk, L. A. (2013). *Essential Statistical Inference*. Springer.
- 21.10. Bratley P., & Fox B. L. (1988). Algorithm 659: implementing Sobol's quasirandom sequence generator. *ACM Transactions on Mathematical Software (TOMS)*, 14(1), 88–100.
- 21.11. Brophy A. L. (1987). Efficient estimation of probabilities in the t distribution. *Behavior Research Methods*, 19(5), 462–466.
- 21.12. Brown L. D., Cai T. T., & DasGupta A. (2001). Interval Estimation for a Binomial Proportion. *Statistical Science*, 16(2), 101–133.
- 21.13. Canal L. (2005). A normal approximation for the chi-square distribution. *Computational Statistics & Data Analysis*, 48(4), 803–808.
- 21.14. Casella G. (1996). The Ghosh-Pratt Identity. Technical report, Cornell University.
- 21.15. Chatfield C. (1995). *Problem Solving: A Statistician's Guide*. CRC.
- 21.16. Chatfield C. (2001). *TimeSeries Forecasting*. CRC.
- 21.17. Chen C. H., Lee L.H. (2010). *Stochastic Simulation Optimization: an Optimal Computing Budget Allocation*. World Scientific.
- 21.18. Chihara L. M., & Hesterberg T. C. (2018). *Mathematical Statistics with Resampling and R*. Wiley.
- 21.19. Coles S. (2001). *An Introduction to Statistical Modeling of Extreme Values*. Springer.
- 21.20. DasGupta A. (2008). *Asymptotic Theory of Statistics and Probability*. Springer.
- 21.21. Davies P. L. (2014). *Data Analysis and Approximate Models: Model Choice, Location-Scale, Analysis of Variance, Nonparametric Regression and Image Analysis*. CRC.
- 21.22. Davison A. C., & Hinkley D. V. (1997). *Bootstrap Methods and Their Application*. Cambridge University Press.

- 21.23. Efron B., & Hastie T. (2016). *Computer Age Statistical Inference*. Cambridge.
- 21.24. Efron B., & Tibshirani R. J. (1993). *An Introduction to the Bootstrap*. CRC.
- 21.25. Florescu I. (2014). *Probability and Stochastic Processes*. Wiley.
- 21.26. Franses P. H., Legerstee R., & Paap R. (2017). Estimating loss functions of experts. *Applied Economics*, 49(4), 386–396.
- 21.27. Gelman A., Carlin J. B., Stern H. S., Dunson D. B., Vehtari A., & Rubin D. B. (2013). *Bayesian Data Analysis*. CRC.
- 21.28. Gibbons J. D., & Chakraborti, S. (2011). *Nonparametric Statistical Inference*. Springer.
- 21.29. Goos P., & Jones B. (2011). *Optimal Design of Experiments: A Case Study Approach*. Wiley.
- 21.30. Gigerenzer G. (2011). What are natural frequencies? *BMJ*, 343, d6386.
- 21.31. Greenland S., Senn S. J., Rothman K. J., Carlin J. B., Poole C., Goodman S. N., & Altman D. G. (2016). Statistical tests, P values, confidence intervals, and power: a guide to misinterpretations. *European Journal of Epidemiology*, 31(4), 337–350.
- 21.32. Johnston W. (2015). *The Lebesgue Integral for Undergraduates*. MAA.
- 21.33. Hahn G. J., Meeker W. Q., & Escobar L. A. (2017). *Statistical Intervals: A Guide for Practitioners*. Wiley.
- 21.34. Hand D. J. (2014). *The Improbability Principle: Why Coincidences, Miracles, and Rare Events Happen Every Day*. Scientific American.
- 21.35. Hesterberg T. (2014). What Teachers Should Know about the Bootstrap: Resampling in the Undergraduate Statistics Curriculum. arXiv preprint arXiv:1411.5279.
- 21.36. Hjorth J. U. (1994). *Computer Intensive Statistical Methods: Validation, Model Selection, and Bootstrap*. Routledge.
- 21.37. Howard R. A. & Abbas A. E. (2015). *Foundations of Decision Analysis*. Pearson.
- 21.38. Howell D. C. (2016). Randomization/Permutation Tests. <https://www.uvm.edu/~dhowell/StatPages/>. Accessed October 16, 2016.
- 21.39. Hubbard D. W. (2014). *How to Measure Anything: Finding the Value of Intangibles in Business*. Wiley.
- 21.40. Huber P. J. (2011). *Data Analysis: What Can Be Learned from the Past 50 Years*. Wiley.
- 21.41. Huber P. J., & Ronchetti E. M. (2009). *Robust Statistics*. Wiley.
- 21.42. Hubert L., & Wainer H. (2012). *A Statistical Guide for the Ethically Perplexed*. CRC.
- 21.43. Hyndman R. J., & Athanasopoulos G. (2018). *Forecasting: Principles and Practice*. OTexts.
- 21.44. Imbens G. W., & Rubin D. B. (2015). *Causal Inference in Statistics, Social, and Biomedical Sciences*. Cambridge.
- 21.45. Katzourakis N., & Varvaruca E. (2018). *An Illustrative Introduction to Modern Analysis*. CRC.
- 21.46. Kiefer J. C. (1987). *Introduction to Statistical Inference*. Springer.
- 21.47. Korostelev A. P., & Korosteleva O. (2011). *Mathematical Statistics: Asymptotic Minimax Theory*. AMS.
- 21.48. Krishnamoorthy K. (2016). *Handbook of Statistical Distributions with Applications*. CRC.
- 21.49. Kucherenko S., Albrecht D., & Saltelli A. (2015). Exploring multi-dimensional spaces: a Comparison of Latin Hypercube and Quasi Monte Carlo Sampling Techniques. arXiv preprint arXiv:1505.02350.
- 21.51. Lehmann E. L., & Casella G. (1998). *Theory of Point Estimation*. Springer.
- 21.52. Lehmann E. L., & Romano J. P. (2005). *Testing Statistical Hypotheses*. Springer.
- 21.53. Lemieux C. (2009). *Monte Carlo and Quasi-Monte Carlo Sampling*. Springer.

- 21.54. Lunneborg C. E. (2000). *Data Analysis by Resampling: Concepts and Applications*. Duxbury.
- 21.55. Mukhopadhyay N., & De Silva B. M. (2008). *Sequential Methods and Their Applications*. Chapman and Hall/CRC.
- 21.56. Nelson G. S. (2015). *A User-friendly Introduction to Lebesgue Measure and Integration*. American Mathematical Society.
- 21.57. Nelson M. J. (2011). You might want a tolerance interval. <http://tinyurl.com/tol-interval>.
- 21.58. Neyman J. (1977). Frequentist probability and frequentist statistics. *Synthese*, 36(1), 97–131.
- 21.59. Panaretos V. M. (2016). *Statistics for Mathematicians*. Springer.
- 21.60. Press W.H. et al. (2007). *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge.
- 21.61. Proschan M. A., & Shaw P. A. (2016). *Essentials of Probability Theory for Statisticians*. CRC.
- 21.62. Rissanen J. (2012). *Optimal Estimation of Parameters*. Cambridge University Press.
- 21.63. Romano J. P. (1990). On the behavior of randomization tests without a group invariance assumption. *Journal of the American Statistical Association*, 85(411), 686–692.
- 21.64. Ryan T. P. (2013). *Sample Size Determination and Power*. Wiley.
- 21.65. Saltelli A., Ratto M., Andres T., Campolongo F., Cariboni J., Gatelli D., & Tarantola S. (2008). *Global Sensitivity Analysis: The Primer*. Wiley.
- 21.66. Samaniego F. J. (2010). *A Comparison of the Bayesian and Frequentist Approaches to Estimation*. Springer.
- 21.67. Santana L. (2009). *Contributions to the m-out-of-n Bootstrap* (Dissertation). North-West University.
- 21.68. Shumway R. H., & Stoffer D. S. (2000). *Time Series Analysis and Its Applications*. Springer.
- 21.69. Self S. G., & Liang K. Y. (1987). Asymptotic properties of maximum likelihood estimators and likelihood ratio tests under nonstandard conditions. *Journal of the American Statistical Association*, 82(398), 605–610.
- 21.70. Shao J. (2003). *Mathematical Statistics*. Springer.
- 21.71. Shao J., & Tu D. (1995). *The Jackknife and Bootstrap*. Springer.
- 21.72. Sobol I. M., Asotsky D., Kreinin A., & Kucherenko S. (2011). Construction and Comparison of High-Dimensional Sobol'Generators. *Wilmott*, 2011(56), 64–79.
- 21.73. StackExchange (2016a). How to investigate properties of the trimmed mean for a Cauchy variable?. <http://stats.stackexchange.com/questions/87354/how-to-investigate-properties-of-the-trimmed-mean-for-a-cauchy-variable>. Accessed October 16, 2016.
- 21.74. StackExchange (2016b). Does the Hodges-Lehmann estimator perform better than trimmed/winsorized means? <https://stats.stackexchange.com/questions/235392/does-the-hodges-lehmann-estimator-perform-better-than-trimmed-winsorized-means>. Accessed October 16, 2016.
- 21.75. Stoyanov J. M. (2013). *Counterexamples in Probability*. Dover.
- 21.76. Temme N. M. (1994). A set of algorithms for the incomplete gamma functions. *Probability in the Engineering and Informational Sciences*, 8(02), 291–307.
- 21.77. Thompson S. K. (2012). *Sampling*. Wiley.
- 21.78. Vexler A., Hutson A. D., & Chen X. (2016). *Statistical Testing Strategies in the Health Sciences*. CRC.
- 21.79. Wasserman L. (2004). *All of Statistics: A Concise Course in Statistical Inference*. Springer.
- 21.80. Wei F., & Dudley R. M. (2012). Two-sample Dvoretzky–Kiefer–Wolfowitz inequalities. *Statistics & Probability Letters*, 82(3), 636–644.
- 21.81. Wellek S. (2010). *Testing Statistical Hypotheses of Equivalence and Noninferiority*. CRC.
- 21.82. Westfall P., & Henning K. S. (2013). *Understanding Advanced Statistical Methods*. CRC.

- 21.83. Westfall P. H., Tobias R. D., & Wolfinger R. D. (2011). Multiple Comparisons and Multiple Tests Using SAS. SAS Institute.
- 21.84. Wikipedia (2014a). Bootstrapping. [http://en.wikipedia.org/wiki/Bootstrapping\\_\(statistics\)](http://en.wikipedia.org/wiki/Bootstrapping_(statistics)). Accessed June 25, 2014.
- 21.85. Wikipedia (2014b). Arrow's impossibility theorem. [http://en.wikipedia.org/wiki/Arrow's\\_impossibility\\_theorem](http://en.wikipedia.org/wiki/Arrow's_impossibility_theorem). Accessed November 23, 2014.
- 21.86. Wikipedia (2014c). Error function. [http://en.wikipedia.org/wiki/Error\\_function](http://en.wikipedia.org/wiki/Error_function). Accessed June 25, 2014.
- 21.87. Wikipedia (2015a). Rule of three. [https://en.wikipedia.org/wiki/Rule\\_of\\_three\\_\(statistics\)](https://en.wikipedia.org/wiki/Rule_of_three_(statistics)). Accessed December 31, 2015.
- 21.88. Wikipedia (2015b). Kolmogorov-Smirnov test. [https://en.wikipedia.org/wiki/Kolmogorov-Smirnov\\_test](https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test). Accessed November 18, 2015.
- 21.89. Wikipedia (2015c). Friedman test. [https://en.wikipedia.org/wiki/Friedman\\_test](https://en.wikipedia.org/wiki/Friedman_test). Accessed November 18, 2015.
- 21.90. Wikipedia (2016a). Low-discrepancy sequence. [https://en.wikipedia.org/wiki/Low-discrepancy\\_sequence](https://en.wikipedia.org/wiki/Low-discrepancy_sequence). Accessed September 15, 2016.
- 21.91. Wikipedia (2016b). Total variation. [https://en.wikipedia.org/wiki/Total\\_variation](https://en.wikipedia.org/wiki/Total_variation). Accessed September 15, 2016.
- 21.92. Wikipedia (2016c). Secretary problem. [http://en.wikipedia.org/wiki/Secretary\\_problem](http://en.wikipedia.org/wiki/Secretary_problem). Accessed September 2, 2016.
- 21.93. Wikipedia (2016e). Efficiency (statistics). [https://en.wikipedia.org/wiki/Efficiency\\_\(statistics\)](https://en.wikipedia.org/wiki/Efficiency_(statistics)). Accessed October 16, 2016.
- 21.94. Wikipedia (2017b). Time management. [https://en.wikipedia.org/wiki/Time\\_management](https://en.wikipedia.org/wiki/Time_management). Accessed July 23, 2017.
- 21.95. Wikipedia (2018a). Student's t-distribution. [https://en.wikipedia.org/wiki/Student's\\_t-distribution](https://en.wikipedia.org/wiki/Student's_t-distribution). Accessed June 30, 2018.
- 21.96. Wikipedia (2018b). Pearson's chi-squared test. [https://en.wikipedia.org/wiki/Pearson's\\_chi-squared\\_test](https://en.wikipedia.org/wiki/Pearson's_chi-squared_test). Accessed October 6, 2018.
- 21.97. Wikipedia (2019a). Beta-binomial distribution. [https://en.wikipedia.org/wiki/Beta-binomial\\_distribution](https://en.wikipedia.org/wiki/Beta-binomial_distribution). Accessed April 12, 2019.
- 21.98. Wikipedia (2019b). Fisher transformation. [https://en.wikipedia.org/wiki/Fisher\\_transformation](https://en.wikipedia.org/wiki/Fisher_transformation). Accessed June 15, 2019.
- 21.99. Wikipedia (2019c). Binomial proportion confidence interval. [https://en.wikipedia.org/wiki/Binomial\\_proportion\\_confidence\\_interval](https://en.wikipedia.org/wiki/Binomial_proportion_confidence_interval). Accessed July 20, 2019.
- 21.100. Wikipedia (2019d). Approval voting. [https://en.wikipedia.org/wiki/Approval\\_voting](https://en.wikipedia.org/wiki/Approval_voting). Accessed October 19, 2019.
- 21.101. Wikipedia (2021). Kendall rank correlation coefficient. [https://en.wikipedia.org/wiki/Kendall\\_rank\\_correlation\\_coefficient](https://en.wikipedia.org/wiki/Kendall_rank_correlation_coefficient). Accessed October 10, 2021.
- 21.102. Wilcox R. R. (2016). Introduction to Robust Estimation and Hypothesis Testing, 4th ed. Academic Press.
- 21.103. Yerukala R., Boiroju N. K., & Reddy M. K. (2013). Approximations to the t-distribution. International Journal of Statistika and Matematika, 8(1).



## 22. Численные алгоритмы: введение и матричная алгебра

Цель вычислений — понимание, а не числа.

*Ричард Хэмминг*

Бог создал целые числа, все остальное — дело рук человека.

*Леопольд Кронекер*

Если бы ошибки с плавающей точкой исчезли, 90% численного анализа не изменилось бы

*Ник Трефетен*

Снижение стоимости вычислений заставляет пользователей выбирать надежность, а не эффективность.

*Джон Ламберт*

Гораздо проще изучить численные методы, чем внедрить надежное численное программное обеспечение.

*Гандер и соавт.*

### 22.1. Введение

Чтобы разобраться в материале этой главы, надо предварительно освоить курс по основным численным методам и быть знакомым с распространенными численными задачами — такими как вычисление интегралов. Для предварительной подготовки вы можете воспользоваться каким-либо студенческим учебником с достаточным количеством примеров — например, работой [22.17]. Изучение численного анализа может потребовать наличия большего количества базовых знаний, чем остальной материал книги. Как минимум нужно прослушать курс линейной алгебры, математического анализа, обыкновенных дифференциальных уравнений (если не читать соответствующие разделы) и иметь понятие о комплексных числах.

Цель численного анализа состоит в том, чтобы создать числовые процедуры, которые позволяют достаточно хорошо решать математические задачи и которые можно реализовать в виде алгоритмов. Стоит подчеркнуть, что численные алгоритмы — такие как статистические алгоритмы и алгоритмы машинного обучения, не являются «пуленепробиваемыми “черными ящиками”», за исключением некоторых особых случаев. Даже самые удачно спроектированные реализации в лучшем случае имеют более или менее документированные и хорошо известные ограничения (позже в этой главе мы обсудим, откуда они берутся). В идеальном случае типичный алгоритм позволяет обосновать тот или иной выбор, отдавая предпочтение простоте, а также понять, какие задачи с помощью этого алгоритма решить нельзя. Алгоритм — это всего лишь процедура, выполняемая компьютером, и его автоматизация не гарантирует качества.

В этой главе сначала рассматриваются общие темы численного анализа — например, корректность в отношении ошибок аппроксимации и ошибок с плавающей точкой. Затем мы рассмотрим реализации наиболее важных матричных алгоритмов.

## 22.2. Арифметика с плавающей точкой

У вычислительных средств существуют ограничения точности, возникающие из-за комбинаторного взрыва. Например, для представления произведения двух 32-битных чисел требуется 64 бита. Но формат с плавающей точкой ограничивает точность, т. е. в таком умножении остается только 32 старших бита, а остальные отбрасываются путем округления. Арифметика с плавающей точкой была стандартизирована с 1985 года, а последний стандарт принят в 2019 году.

Число  $x$  любого типа точности (одинарное, двойное и т. д.) представляется как  $\pm(1 + m)2^e$ , где:

- ♦  $m$  — *мантисса*, нормированная на диапазон  $(0, 1)$ ;
- ♦  $e$  — *степень*.

Например, для числа формата `double` из 64 битов один бит выделяется на знак, 52 — на  $m$  (получается 53 бита точности с неявной единицей, что дает около 16 десятичных цифр) и 11 на  $e$ . Формат позволяет представлять несколько типов чисел:

- ♦ *нормальные* —  $m$  использует все выделенные биты. С такими числами выполняются почти все вычисления;
- ♦ *субнормальные* — когда нормальное число становится слишком маленьким ( $e$  большое и отрицательное), чтобы сделать возможным постепенное изменение до 0, некоторые биты  $m$  отдаются  $e$ . Для формата `double` числа  $|x| < 2^{-1022} \approx 1,8 \times 10^{-308}$  являются субнормальными. Субнормаль существует только для обеспечения некоторых гарантий операций с нормальными — таких как точное округление;
- ♦ *специальные константы* — наиболее важными из них являются 0 (все биты = 0),  $-0$  ( $= 0$ , но бывает полезен),  $\pm\infty$  и несколько значений NaN. Они ведут себя интуитивно ожидаемо — например:  $1/0 = \infty$  и  $0/0 = \text{NaN}$ . NaN и  $\pm\infty$  гарантируют, что все операции четко определены.

Точный ноль обманчив, поскольку многие близлежащие числа при округлении также могут считаться равными нулю. Поэтому в алгоритмах не следует вводить проверку на равенство нулю. Точные детали кодирования чисел с плавающей точкой не важны (посмотрите работу [22.29], если любопытно), а полагаться на детали представления не стоит. C++ и большинство других языков обеспечивают несложный доступ к ряду параметров, но регулярно используются только некоторые из них. В частности, итерация по всем числам с плавающей точкой от 0 до  $\infty$  выполняется с шагом в 1 **ulp** (unit in the last place, единица в последнем разряде). Для  $x > 0$   $\text{ulp}(x)$  = (наименьшее число с плавающей точкой  $> x$ ) – (наибольшее число с плавающей точкой  $\leq x$ ). Это общее определение, которое охватывает большинство случаев. Тогда *машинный epsilon*  $\varepsilon = \text{ulp}(1)$ , т. е.  $\varepsilon$  — это наименьшее значение  $x > 0$ , такое что  $1 + x \neq 1$ . В коллекции `T = numeric_limits<double>` содержится несколько важных чисел (табл. 22.1).

Таблица 22.1

Число	Аксессор	Приблизительное стандартное значение
Минимальный $x$ такой, что $1 + x \neq 1$	<code>T::epsilon()</code>	2.2E-16
Минимальный нормальный $x$	<code>T::min()</code>	1,8E-308

Таблица 22.1 (окончание)

Число	Аксессор	Приблизительное стандартное значение
Максимальный нормальный $x$	<code>T::max()</code>	2.2E+308
бесконечность	<code>T::infinity()</code>	$\infty$
NaN	<code>T::quiet_NaN()</code>	Н/Д; равные NaN значения ложны

Соответствующие отрицательные числа равны  $-x$ . Функция `isfinite` из библиотеки `<cmath>` позволяет проверить, равно ли значение  $\infty$  или NaN (также можно использовать `isnan` для проверки на NaN).

Измерение с помощью `ulr` допускает ошибки измерения. Положим, что для реального  $x$  имеем  $fl(x) = x$ , округленное до соответствующего числа с плавающей точкой, в зависимости от режима округления:

- ◆ ближайшее — с использованием обычного округления, т. е.  $0,5 \rightarrow 1$ ,  $0,49 \rightarrow 0$ ;
- ◆ вниз — т. е.  $0,5 \rightarrow 0$ ;
- ◆ вверх — т. е.  $0,49 \rightarrow 1$ .

По умолчанию используется округление до ближайшего, поскольку ошибка в этом случае  $\leq ulp(x)/2$ . При округлении вниз и вверх ошибка  $\leq ulp(x)$ , но эти методы округления полезны в задачах интервальной арифметики (обсуждаются в разделе комментариев). Существует также режим равенства нулю, но он не используется). Результат операции считается *правильно округленным*, если он соответствует наилучшему приближению числа с плавающей точкой при заданном режиме округления. Элементарные операции с плавающей точкой  $\{+, -, \times, /\}$ ,  $\sqrt{\phantom{x}}$  и остаток от деления округляются правильно (см. [22.29], в последнем стандарте могут правильно округляться и другие операции).

Вычисления с числами с плавающей точкой происходят в ЦП, и *регистры с плавающей точкой* имеют большую точность (обычно 80 битов). Это позволяет сохранить точность промежуточных результатов последовательностей операций, если вычисления остаются в регистрах.

Для обсуждения корректности многих численных алгоритмов может использоваться реальная модель оперативной памяти (см. главу 1. *С чего всё начинается...*). Также иногда делают *предположение об общем положении*, в рамках которого входные данные предполагаются неравными. Хотя это может быть полезно аналитически, в коде должно обрабатываться равенство.

## 22.3. Ошибки при использовании арифметики с плавающей точкой

Даже при правильном округлении могут возникнуть логические ошибки:

- ◆ *переполнение* — например:  $10!! = \infty$ ;
- ◆ *недостаточная величина* — например:  $2-10\,000 = 0$ ;

♦ *относительная недостаточность* — например:  $1 + \varepsilon/2 = 1$  (но операция сработает как промежуточный шаг в регистре).

В мире арифметики с плавающей точкой это четко определенные правильные результаты. Но в мире идеальной арифметики могут возникнуть сюрпризы, когда доказуемо правильная формула приводит к неправильному результату, поскольку почти все теоремы об алгоритмах предполагают точную арифметику. Неточность является необходимым приближением и, как правило, весьма неплохим, но ее нужно уметь выявить. В целом можно игнорировать ошибку с плавающей точкой, за исключением случаев, когда это невозможно. А чтобы понять, можно ли ее игнорировать, требуется специальный анализ.

Стандарт с плавающей точкой дает предсказуемость результатов, но с некоторыми оговорками. Например, когда функция  $f$  детерминирована, выражение  $f(x) = f(x)$  может быть ложным, если результат первого вызова хранится в памяти, а второго — нет, и разница возникает из-за округления до разной точности. Если предположить, что в обоих случаях промежуточные шаги  $f$  вычисляются в одном и том же шаблоне памяти/регистра, разница результатов должна быть  $\leq$  отброшенной округлением, но в противном случае это может не выполняться. В общем случае число в памяти, преобразованное в регистр и обратно, остается прежним, но обратное неверно.

Одна из сложных ошибок возникает из-за *накопления округлений* — последовательность операций может быть неправильно округлена. Например, попробуйте вычислить  $\pi$ , просуммировав сходящийся ряд, — в конце концов члены станут слишком малы, чтобы численно влиять на частичную сумму (см. [22.22]). Наивные реализации многих алгоритмов не срабатывают, несмотря на корректность работы в реальной оперативной памяти.

Стандартные законы арифметики — такие как коммутативность и ассоциативность, не выполняются, поэтому оптимизацию компилятора, например путем перестановки выражений, становится проблематично реализовать. Кроме того, трудными становятся некоторые структурные решения. Например, зададим такой вопрос: является ли матрица симметричной, если некоторые ее симметричные элементы различны с точностью до некоторой малой точности  $> \varepsilon$ ? Ответ может быть любым.

К счастью, существенного накопления округлений обычно не происходит, а если и происходит, то можно подправить имеющийся алгоритм или использовать другой. Большая часть численного анализа заключается в обеспечении того, чтобы алгоритмы, работающие с арифметикой с плавающей точкой, давали ответы, максимально приближенные к реальным результатам оперативной памяти. Но есть и другие типы ошибок. На самом деле численные методы ошибаются по тем же причинам, что и машинное обучение и статистические алгоритмы:

- ♦ *ошибка приближения* — решается немного другая задача;
- ♦ *ошибка оценки* — вычисления выполняются с ограниченной точностью и могут опираться на неточные входные данные;
- ♦ *ошибка оптимизации* — вычислений выполнено недостаточно.

Чтобы уменьшить ошибку оценки, часто используют другую математическую формулу для того же вычисления. Возможно, формула и будет совершенно другой, но чаще всего речь идет о переписывании имеющегося выражения так, чтобы уменьшить количе-

ство ошибок. Например, для корней квадратных уравнений относительная погрешность может быть высокой, если  $4ac < \epsilon b$ , потому что вычисление  $b - \sqrt{b(b + \epsilon)}$  в этом случае дает некорректное вычитание. Но можно вычислить корень со знаком «+» и получить из него корень со знаком «-» по другой формуле. Сложение/вычитание сохраняют абсолютное значение точности, а умножение/деление — относительное. В работе [22.4] описана более точная процедура решения квадратных уравнений.

Всегда необходимо анализировать формулу, чтобы убедиться, что в ней нет некорректных операций вроде вычитания равных значений или деления на 0. Забудьте обо всем, что вы знаете о реальной арифметике, и не делайте никаких предположений. Представьте себе фрагмент общего кода C++, в котором каждый возможный оператор перегружен и может вести себя непредсказуемым образом — например, случайным образом генерировать исключения. В работе [22.1] есть примеры по этой теме. Они напоминают вам о том, что, хотя некоторые строительные блоки алгоритмов хорошо известны и доступны, решение численных задач иногда требует некоторой изобретательности, а не просто прямого применения строительных блоков. Например, чтобы избежать неправильного вычитания, вам может потребоваться заменить функцию  $\cos$  степенным рядом с нужным числом членов.

Общая схема минимизации потери точности при выполнении последовательности вычислений заключается в том, чтобы по возможности сначала выполнять операции, в которых потери точности больше. Тогда последующие операции, потери в которых меньше, будут отбрасывать только мусорные биты.

Большинству алгоритмов для получения точных ответов требуются разумно масштабированные входные данные. Автоматически определить масштаб входных данных не поможет никакая изобретательность, потому что достаточно малые числа неотличимы от 0. В этой ситуации становится важна не *относительная точность*, а *абсолютная*. Поскольку  $T::\min()$  намного меньше, чем  $\epsilon$ , и даже одна операция может привести к абсолютной ошибке  $O(\epsilon)$ , ответу, значение которого меньше  $O(\epsilon)$ , доверять нельзя. Чтобы этого избежать, можно выполнить некоторое эвристическое масштабирование.

Доказательства правильности вычисления с плавающей точкой даже в простых программах весьма сложны из-за множества особых случаев. Для некоторых алгоритмов — таких как вычисление элементарных функций, разрабатываются автоматизированные помощники, выполняющие доказательство (см. [22.27]).

Рассмотрим простую задачу: проверим, выполняется ли условие  $a = b$ . Выполнять прямое сравнение обычно бессмысленно (хотя оно сработает, если  $a$  и  $b$  представлены точно). Более разумной альтернативой является проверка вида  $|a - b| \leq \epsilon \max(|a|, |b|, \min normal)$ . Это лучшее, чего можно добиться от стандарта, но обычно критерий задается мягче:

- ◆ используйте некоторое число  $\epsilon \geq$  единицы округления;
- ◆ используйте некоторую абсолютную точность, которая намного больше, чем  $\epsilon \times \min normal$ . Использование значения  $\epsilon$  удобно при наличии правильно масштабированных входных данных. В некоторых источниках используют гораздо меньшую абсолютную точность — например,  $\epsilon^2$ , но при надлежащем масштабировании кажется неразумным выходить за пределы  $\epsilon/10^6$  или около того, и важно не потерять эту дополнительную точность позже.

Удобно использовать сравнения с числом  $\epsilon$  вместо прямого равенства. Для эффективности и правильного поведения с NaN сначала стоит проверить условие  $a < b$ :

```
bool isELess(double a, double b,
    double eRelAbs = numeric_limits<double>::epsilon())
{return a < b && b - a >= eRelAbs * max(1.0, max(abs(a), abs(b)));}
bool isEEqual(double a, double b,
    double eRelAbs = numeric_limits<double>::epsilon())
{return !isELess(a, b, eRelAbs) && !isELess(b, a, eRelAbs);}
```

Результаты должны быть правильными для всех возможных значений  $a$  и  $b$ . Используйте юнит-тест, который проверяет:

- ◆ NaN и  $\pm\infty$ . Помните, что сравнение с NaN не транзитивно, потому что  $\text{NaN} \neq \text{NaN}$ ;
- ◆ небольшие числа — такие как 0,  $\epsilon$ , минимальные нормальные значения и т. д.

```
void testELessAuto()
{
    double nan = numeric_limits<double>::quiet_NaN(),
        inf = numeric_limits<double>::infinity();
    double es[4] = {numeric_limits<double>::epsilon(), highPrecEps,
        defaultPrecEps, 0.1};
    for(int i = 0; i < 4; ++i)
    {
        double eps = es[i];
        // nan
        assert(isELess(nan, nan, nan) == false);
        assert(isELess(nan, nan, eps) == false);
        assert(isELess(nan, 1, eps) == false);
        assert(isELess(1, nan, eps) == false);
        assert(isELess(nan, inf, eps) == false);
        assert(isELess(inf, nan, eps) == false);
        // бесконечность
        assert(isELess(inf, inf, eps) == false);
        assert(isELess(-inf, -inf, eps) == false);
        assert(isELess(-inf, inf, eps) == true);
        assert(isELess(-inf, 1, eps) == true);
        assert(isELess(-1, inf, eps) == true);
        // нормальное значение
        for(double x = numeric_limits<double>::min() * 10;
            x < numeric_limits<double>::max()/10; x *= 10)
        {
            double dx = eps * max(1.0, abs(x));
            assert(isELess(x, x + 2 * dx, eps) == true);
            assert(isELess(x, x + 0.5 * dx, eps) == false);
            assert(isELess(x - 2 * dx, x, eps) == true);
            assert(isELess(x - 0.5 * dx, x, eps) == false);
        }
    }
}
```

Обратите внимание на юнит-тест: сравнение не пытается перебрать все возможные стандартно разрешенные числа с плавающей точкой между  $a$  и  $b$  — например, можно использовать погрешность  $\epsilon/2$ , когда по умолчанию задействуется режим округления

до ближайшего, но на практике это ничего не дает. Написать на 100% надежный оператор сравнения сложно, и в некоторых подходах `ulp` сравниваются напрямую (см. <http://floating-point-gui.de/errors/comparison/> и <https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/>). Гораздо понятнее реализуются сравнения с числом  $\epsilon$ . Во многих алгоритмах используется операция  $\langle 1 + |x| \rangle$  вместо  $\max(1, |x|)$  для простоты и автоматической обработки NaN.

Проверка условия  $x = 0$  зависит от задачи. Использование некоторой абсолютной точности по умолчанию, описанное ранее, во многих случаях полезно, но далеко не во всех. В финансовой торговле значения вроде цены имеют бесконечную точность, потому что цена  $\times$  количество (без учета некоторых других деталей) определяет стоимость сделки. Даже небольшая ошибка округления цены при достаточно большом количестве товара превратится в серьезную ошибку в стоимости. На практике этого не происходит, но финансовые программисты должны помнить об этом, особенно при выполнении расчетов цены и доходности и связанных с ними расчетов.

## 22.4. Ошибка приближения

Даже в точной арифметике численный алгоритм может не работать, если:

- ◆ используемый итеративный процесс не гарантирует сходимости;
- ◆ требуется вычислить некоторое свойство функции «черного ящика»  $f$ , которое известно только посредством оценки на некотором количестве точек, а полного аналитического описания нет.

Первое условие рассматривается в каждом отдельном случае. Последнее характерно для многих задач. В частности, не существует разумного способа решить задачи с плохими свойствами — такие как минимизация функции с пиковым значением  $-\infty$ . Тогда, если, например,  $f(10,02) = -100\,000$  и  $f(10,0201) = -10\,100$ , лучший ответ алгоритма минимизации будет выглядеть так: «10,02 с точностью 0,001». Такова природа некорректно поставленных задач. Поэтому алгоритмы обычно гарантируют работу только на некотором ограниченном подмножестве функций.

Выборка  $f$  на конечном числе точек по-прежнему ограничивает размер множества всех возможных  $f$  до  $\infty$ . Если  $f$  не является липшицевой или не имеет другого подходящего ограничения, заставляющего все эти возможные функции вести себя одинаково при любом сделанном выводе, такой вывод не имеет силы, и задачи, основанные на работе с «черными ящиками», — такие как интерполяция, интегрирование и дифференцирование (обсуждаемые в главе 23. *Численные алгоритмы: работа с функциями*), не имеют решения. Это ограничение применимо только к детерминистическим алгоритмам. Например, интегрирование методом Монте-Карло работает для прерывистого  $f$  (но все же границы ошибки лучше, если у нее есть ограниченный диапазон).

Многие методы численного анализа основаны на *теореме Тейлора* (см. [22.52]: при заданной  $f$  с  $k + 1$  непрерывными производными  $\in(a, x)$ :

$$f(x) = \sum_{0 \leq i \leq k} \frac{f^{(i)}(a)}{i!} (x-a)^i + \frac{f^{(k+1)}(\xi)}{(k+1)!} (x-a)^{(k+1)} \quad \text{для некоторого } \xi \in (a, x). \quad \text{Для слу-}$$

чая  $x < a$  результат выглядит симметрично. Часто предполагают, что  $\|f^{(k+1)}\|_\infty$  или другое выражение ограничено некоторой константой  $M$ .

## 22.5. Показатели устойчивости и состояния

Пусть численный алгоритм представляет собой вычисление  $A(x)$  вместо  $y = f(x)$  для некоторых  $f$  и  $x$ . По части накопления ошибок с плавающей точкой — нужно знать *прямую ошибку*  $(A, x) = |y - A(x)|$ . К сожалению, для ее вычисления требуется знать значение  $y$ , а оно обычно неизвестно. Алгоритм с прямой ошибкой  $O(\epsilon)$  обладает *прямой устойчивостью*.

Обычно работают с *обратной ошибкой*  $(A, x) = \max |\Delta x = x_2 - x|$  — такой что в точной арифметике  $A(x) = f(x_2)$ . Эти определения распространяются на более общие входные и выходные пространства с помощью соответствующих норм. Алгоритм с обратной ошибкой  $O(\epsilon)$  обладает *обратной устойчивостью* или просто *устойчив*. Таким образом:

- ♦ в точной арифметике обратная ошибка равна нулю;
- ♦ в арифметике с плавающей точкой правильно округленные операции обладают обратной стабильностью.

Правильное округление — это обратное, а не прямое управление ошибками. Например, рассмотрите отмену при вычитании почти равных чисел: учитывая исходные точные аргументы, представления которых с плавающей точкой точны до последнего бита, точность результата будет низкой, за исключением редкого случая, когда округление не изменяет аргументы. Правильно округленный ответ подходит только для округленных аргументов, а не для точных. Предположение точности аргумента дает определение обратной ошибки, но для точных аргументов правильно округленные младшие биты являются мусорными. Чем ближе аргументы, тем больше будет потеряно точности.

Модуль непрерывности  $f$  равен  $w_f(a) = \max_{\|\Delta x\| \leq a} \|f(x + \Delta x) - f(x)\|$  (см. [22.32]).

Обычно он определяется в одном измерении, но работает с любой нормой. Число абсолютного условия задачи вычисляется таким образом, что определенная обратная ошибка дает границу прямой ошибки — для абсолютной обратной ошибки  $\leq a - \kappa_{abs}(a) =$

$= \max_{0 < \|\Delta x\| \leq a} \frac{w_f(a)}{\|\Delta x\|}$ . Тогда  $\|\Delta y\| \leq w_f(a) = \kappa_{abs}(a) \|\Delta x\|$ . Число относительного условия

зависит от относительной ошибки и определяется перестановкой ранее приведенного  $\frac{\|\Delta y\|}{\|y\|} \leq \kappa_{rel}(a) \frac{\|\Delta x\|}{\|x\|}$ , где  $\kappa_{rel}(a) = \kappa_{abs}(a) \frac{\|x\|}{\|y\|}$ . Обратите внимание на неравенство: пря-

мая ошибка может быть небольшой, даже если ограничивающие переменные не равны, что иногда случается. Тогда конечное число условия гарантирует, что задача *корректна* — т. е. результат непрерывно зависит от входа.

Функция  $f$  является *липшицевой* на интервале  $I$ , если  $\forall x_1, x_2 \exists$  константа  $L$  такая, что  $\|f(x_1) - f(x_2)\| < L \|x_1 - x_2\|$ . То есть для  $f$ , которое выбирается в некоторых точках, нет произвольно различного поведения между выборками, как это могло бы быть, например, у непрерывной функции. Если  $f$  — липшицева в интересующей области с константой  $L$ ,  $\kappa_{abs} \leq L$  и  $w_f(a) \leq La$ . Также  $f(x) \leq L \leq \max |f|$ . Часто важно учитывать ограничения предметной области, потому что, например,  $f(x) = x^3$  изменяется гораздо медленнее вблизи нуля.



Работать со значением  $a$  неудобно, поэтому обобщенные определения полезны только интуитивно. Вместо этого лучше работать с линейным приближением — для дифференцируемой  $f$ , основанной на первых нескольких членах разложения в ряд Тейлора функции  $f$ :

$$\kappa_{abs}(f, x) = f'(x) + O(\|\Delta x\|^2) \text{ и } \kappa_{rel}(f, x) = \frac{x}{f(x)} \kappa_{abs}(f, x) \text{ (см. [22.22]).}$$

При малых  $\|\Delta x\|$  можно игнорировать член 2-го порядка, а в противном случае для определения границ нужен модуль непрерывности. Например, сложение двух чисел  $b$  и  $c$  является линейной функцией в двух и одном измерении. Пусть норма — это норма вектора  $L_1$ , расширенная на норму матрицы (это обсуждается далее в главе). Тогда  $\|f(b, c)\| = 1$  (см. [22.15]). Соответственно:  $\kappa_{abs} = 1$  и  $\kappa_{rel} = \frac{|b| + |c|}{|b + c|}$ . Получается, что если

$u$   $b$  и  $c$  одинаковая величина и разные знаки, имеет место плохое вычитание, что видно из значения  $\kappa_{rel}$ . В работе [22.12, с. 58]) приведен аналогичный анализ с нормой  $\infty$ . Задачи с большими и малыми  $\kappa$  называются соответственно *хорошо обусловленными* и *плохо обусловленными*.

При работе с негладкими функциями — такими как  $f(x) = |x|$  или ломаными линиями, можно использовать субградиент максимальной нормы (см. главу 24. *Численная оптимизация*). Ряд Тейлора в этом случае уже работает, поэтому нельзя напрямую отбросить члены второго порядка и получить полезное значение. При работе с произвольными нормами используйте производную Фреше или соответствующий субградиент. Если разрыв  $\kappa = \infty$ , то, зная  $w_f(a)$ , можно теоретически ограничить прямую ошибку, которая, скорее всего, и будет ограничена. Таким образом, линейный анализ условий представляет собой наихудший случай, который является очень пессимистичным для небольшого числа скачкообразных разрывов. Можно получить локальное значение  $\kappa$ , которое, однако, само по себе неустойчиво, если поблизости находится разрыв. На практике анализ условий почти всегда используется как качественное руководство, и я никогда не видел граничных значений, основанных на  $w_f(a)$ , несмотря на то, что это худшая, а не асимптотическая граница.

Таким образом, если алгоритм является устойчивым в обратном направлении, а  $\kappa$  равна  $O(1)$  по отношению к  $\varepsilon$ , алгоритм является устойчивым в прямом направлении. Значение  $\kappa$  необходимо вычислять для конкретной задачи, а обратная устойчивость является свойством алгоритма. Теорема (см. [22.10]):  $\kappa$  (абсолютное или относительное) субмультипликативно относительно композиции функций, т. е.  $\kappa_f \leq \kappa_h \kappa_g$  для  $f(x) = h(g(x))$ .

Обратная устойчивость  $\sigma$  не имеет такой зависимости, поскольку последовательность операций может увеличивать погрешность результата. Например, для  $g(x) = f(\lg(x))$  небольшая линейная ошибка в  $\lg(x)$ , которая находится в обратном направлении от  $f$ , может быть увеличена в  $x$  раз. Но есть и другие полезные отношения (см. [22.15]):

- ◆  $\sigma_f \kappa_f \leq (\sigma_h + \sigma_g \kappa_g) \kappa_h$ ;
- ◆ если  $\kappa_f \leq \kappa_h \kappa_g$ , что имеет место, когда  $f$  и  $g$  скалярны и дифференцируемы, то  $\sigma_f \leq \frac{\sigma_h}{\kappa_g} + \sigma_g$ .

Некоторые алгоритмы, корректно работающие в реальной оперативной памяти (которые проходят на занятиях по математике), такие как *ортогонализация Грама — Шмидта*, неустойчивы. Некоторые задачи — такие как оценка производной, имеют  $\max_x(\kappa) = \infty$ . Но это не означает, что такой случай совсем безнадежен — поведение в наихудшем случае не мешает средним случаям давать удовлетворительные прямые ошибки. В некоторых редких ситуациях не существует точного входного значения, которое соответствует выходному (см. [22.10]), поэтому обратный анализ невозможен, но значение  $\kappa$  по-прежнему важно.

Устойчивость алгоритма означает, что если его входные данные имеют неопределенность  $\geq$  обратной ошибке, вычисленный ответ будет максимально качественным, даже если прямая ошибка окажется большой из-за плохой обусловленности. В этом случае ни один другой алгоритм не может работать лучше, если он не более устойчив. Устойчивые алгоритмы можно использовать без сожаления и дальнейшего анализа эффектов, связанных с арифметикой с плавающей точкой, что очень полезно, поскольку анализировать такие эффекты обычно нецелесообразно. Одна из основных целей численного анализа — разработка более устойчивых алгоритмов. Поэтому в идеале следует всегда:

- ◆ использовать только устойчивые (насколько это возможно) алгоритмы;
- ◆ решать в качестве промежуточных шагов только хорошо обусловленные задачи.

Элементарные арифметические операции устойчивы, поэтому теряют устойчивость только в том случае, если их последовательность решает плохо обусловленную задачу.

## 22.6. Ошибка оптимизации

Вычислительные сложности обычно бывают двух типов:

- ◆ прямой алгоритм, корректный при работе в реальной оперативной памяти, доступен, но слишком дорог для выполнения на некоторых входных данных. Например, у многих матричных алгоритмов время выполнения  $O(n^3)$  не допускает значения  $n > 10\,000$  или около того. Общее решение состоит в том, чтобы заменить такой алгоритм более дешевым приближенным алгоритмом, что часто делается в матричной алгебре;
- ◆ итеративный алгоритм сходится слишком медленно, чтобы достигать полной точности. Обычно работают только с приемлемой низкой точностью.

Первая проблема рассматривается в каждом отдельном случае, а вторая является основной темой численного анализа. Итеративные алгоритмы не точны на реальной оперативной памяти из-за конечного числа итераций, но могут выполняться до тех пор, пока не будет достигнута какая-либо конкретная точность.

*Погрешность* точности, выбираемая для завершения для итеративного алгоритма, может быть:

- ◆ *абсолютной* — разница некоторых значащих чисел относительно 0 бессмысленна для больших чисел, но вы можете ограничивать количество итераций для достижения определенной точности, если известно уменьшение ошибки на итерацию;
- ◆ *Коши* — разность между двумя последовательными членами убывающей последовательности. Обычно условием остановки является  $|\text{answer}_{i+1} - \text{answer}_i| \leq \varepsilon \times \text{answer}_i$  для некоторого  $\varepsilon$  и  $i$  не слишком малого, чтобы избежать *преждевре-*

менного завершения, когда два найденных подряд ответа случайно оказываются слишком близкими. Часто используется задающее условие остановки свойства задачи, такое как  $\infty$ -норма входных данных, чтобы убедиться, что вычисления не выходят за рамки ожидаемого округления.  $\infty$ -норма привлекательна с точки зрения наихудшего случая, но иногда для удобства используют 2-норму. Определение  $\infty$ -нормы имеет значение с математической точки зрения. При использовании максимума рассматриваются все значения, а предел нормы  $L_p$  для  $p \rightarrow \infty$  игнорирует наборы меры 0;

- ♦ *относительная* — в отличие от первых двух вариантов, мы не знаем ответ, с которым выполняется сравнение, поэтому смотрим на ожидаемые ошибки округления. Это значит, что процесс выполняется до тех пор, пока не будет достигнута точность выше  $\epsilon_{\text{machine}}$ . Такое условие остановки часто используется в матричных алгоритмах.

Эти и любые другие условия остановки не являются полным доказательством, поскольку точный ответ обычно неизвестен и процесс может сходиться в неверном направлении.

Относительной ошибкой многих алгоритмов управлять намного труднее, чем абсолютной. Например, для решения уравнения бинарного поиска на интервале следует сокращать интервал по абсолютной исходной длине, а не по значащим цифрам. Это ошибка оптимизации. Но на практике контроль относительной ошибки почти всегда работает хорошо, потому что алгоритмы улучшают абсолютную ошибку до тех пор, пока относительная ошибка не станет достаточно хорошей, хотя в худшем случае это невозможно. Пользователи API должны всегда понимать, какой контроль ошибок используется и что он означает.

*Скорость сходимости* алгоритма — это точность, достигаемая за одну итерацию. Ее можно выразить несколькими способами:

- ♦ интуитивным, где ошибка  $\leq O$  (некоторая функция числа итераций  $n$ ). Эта функция определяет *порядок метода* по аналогии с многочленом. Но некорректно говорить, что « $O(n)$  означает линейную сходимость», потому что это определение зарезервировано в численном анализе для другого понятия;
- ♦ специфичным для итерации,  $\text{error}_{i+1} = O$  (некоторая функция  $\text{error}_i$ ) — т. е. количество дополнительных цифр, полученных за итерацию. Для *линейной сходимости* ошибка уменьшается на постоянный коэффициент на каждой итерации, что приводит к экспоненциальной сходимости по  $n$ . *Квадратичная сходимость* является суперэкспоненциальной по  $n$  за счет добавления квадрата ошибки.

Рассмотренные способы также называют *арифметической* и *геометрической сходимостью* соответственно. Большая часть численного анализа связана с поиском приближений и процессов, которые приводят к асимптотически более быстрой сходимости. В последнем случае усилия часто стоят результата, но, как правило, достаточно линейной сходимости. В первом случае обычно большее значение имеют другие факторы, поэтому нередко стараются добиться квадратичного или кубического порядка.

Одно из интересных решений — выбор явной погрешности завершения для решения той или иной задачи, в очень общем случае определяемой путем нахождения  $f(x)$  для соответствующего функционала  $f$ . Возможные варианты:

- ♦ делайте так же, как работают элементарные функции, — т. е. вычисляйте  $f(x)$  с полной точностью и правильно округляйте результат. Как правило, это слишком слож-

но даже для устойчивого алгоритма, поэтому нужно просто добиться как можно большей точности. Но обратите внимание, что подпрограммы элементарных функций не принимают количество оценок или точность от пользователя и выбирают правильное округление или, по крайней мере, как можно более точное вычисление;

- ◆ достигайте необходимой точности минимальным числом вычислений, которое обычно измеряется с точки зрения оценки функции. Это может привести к существенной экономии в задачах вроде машинного обучения, где требуется всего несколько цифр точности.

Первый вариант является более сложной задачей, потому что иногда требуется вычисление как можно большего количества цифр, а чрезмерная экономия на оценках функций и других ресурсах не имеет смысла. Кроме того, обычно гладкие функции оцениваются дешево, а дорогие для оценки функции, такие как выводы моделирования, не являются липшицевыми, поэтому не стоит надеяться на большую точность. Чтобы получить лучшее от обоих подходов, используйте алгоритмы, которые по мере необходимости обеспечивают полную точность вычислений и меньшую точность завершения, хотя в некоторых случаях другие алгоритмы работают лучше, получая низкую точность. В будущем, вероятно, произойдет переход с двойной точности на учетверенную и т. д., и в идеале алгоритмы не должны будут думать о точности, но расчеты с максимальной точностью все еще будут осуществимы.

Таким образом, алгоритмы, которым требуется погрешность завершения, обычно используют по умолчанию одно из двух (что лучше подходит):

```
double defaultEps = sqrt(numeric_limits<double>::epsilon());  
double highPrecEps = 100 * numeric_limits<double>::epsilon();
```

Число 100 выбрано произвольно, но вообще для учета накопления ошибки округления подойдет число 1000 или около того.

## 22.7. Другие общие темы

Как и статистические алгоритмы и алгоритмы машинного обучения, численные алгоритмы работают, хотя и не всегда. По функционалу различных библиотек создается впечатление, что задачи, для которых существуют реализации, решаются полностью, но это далеко не так:

- ◆ никакой код, как бы хорошо он ни был сделан, не может решить задачу, не имеющую решений, и схожие плохо обусловленные задачи. В лучшем случае хорошие библиотеки справляются с типичными задачами. Например, в работе [22.31] приводится обзор и периодическая критика многих инструментов для оптимизации;
- ◆ «черный ящик», который вы получаете, может оказаться не тем «черным ящиком», который вам нужен, потому что некоторые процедуры осуществить невозможно, не имея дополнительной информации — такой как точность завершения;
- ◆ реплики проприетарного программного обеспечения для численных вычислений с открытым исходным кодом — такие как совместимая с MATLAB система Octave, могут не реализовывать одни и те же алгоритмы для одних и тех же задач. В них используются как новейшие библиотеки с открытым исходным кодом, так и проприетарные библиотеки.

По сравнению с базовым алгоритмом, приведенным в книге, библиотека в лучшем случае даст вам следующее:

- ◆ выбор алгоритма;
- ◆ реализации, обрабатывающие крайние случаи;
- ◆ обширные наборы тестов, включая варианты использования «у пределов возможностей машины».

Для примера вы можете взглянуть на документацию и исходный код научной библиотеки GNU (см. [22.19]). В ней повторно реализованы многие алгоритмы из высококачественных классических библиотек, таких как QUAD-PACK, содержатся полезные тесты для общих случаев использования и ссылки на документы, из которых были взяты детали реализации. Но неясно, как логика надежности, поведение «у пределов возможностей машины» и выбор алгоритма соотносятся с известными хорошими проприетарными алгоритмами — такими как в MATLAB.

*Аналитическая предварительная обработка* важна в случаях, когда методы «черного ящика» не работают (в работах [22.8 и 22.7] приведены соответствующие сборники задач). Она не всегда выполняется успешно, и у многих задач нет хороших решений, которых можно было бы добиться с помощью имеющихся (и, возможно, будущих) методов. Основной задачей численного исследования является расширение круга задач, решаемых методами «черного ящика» или полуаналитическими методами. Помимо необходимости сильных математических знаний, практическая проблема с аналитической предварительной обработкой заключается в том, что современные задачи весьма серьезны и могут оказаться слишком сложными и крупномасштабными для человеческого анализа.

Хороший численный анализ подразумевает изучение:

- ◆ ошибок аппроксимации и оптимизации алгоритма;
- ◆ условий проблемы;
- ◆ свойств устойчивости алгоритма.

Дальнейшее свидетельство, оспаривающее универсальную эффективность «черных ящиков», исходит из теории сложности точной арифметики. Есть и отрицательные результаты — например, детерминированное интегрирование без дополнительных предположений неразрешимо (конечное число вызовов функций приводит к конечному количеству *информации* — см. [22.41]). Обратите внимание на теоремы о сходимости и производительности различных алгоритмов. Если условия, при которых они работают, не выполняются, в лучшем случае стоит использовать эвристические алгоритмы.

Для сходящихся и устойчивых алгоритмов при вычислениях с двойной точностью безопасно предположить точность в 10 цифр или около того. Велика вероятность, что правильнее будет использовать большее количество цифр, если задача не является плохо обусловленной. Общее эмпирическое правило (см. [22.8]) заключается в том, что когда несколько несвязанных алгоритмов дают один и тот же ответ с определенной точностью, то это и есть правильный ответ с этой точностью. Использование нескольких алгоритмов может иметь смысл, если цена принятия неточных ответов высока.

Общий метод проверки устойчивости и отсутствия ошибок кодирования состоит в том, чтобы запустить алгоритм со случайно искаженными данными и возмущениями той же

величины, что и точность данных, и проверить распределение выходных данных. Если есть представление о том, насколько сильно нужно возмущать данные, это происходит полностью автоматически — в случае многомерных выходных данных можно проверить нормы по невозмущенному решению.

## 22.8. Разработка надежного численного программного обеспечения

Учитывая, что полностью защищенные решения «черного ящика» невозможны, нужно сделать так, чтобы программное обеспечение было максимально надежным. Численные алгоритмы имеют разные уровни разработки (см. [22.9]):

1. Код или псевдокод в учебниках — часто не описывает важные детали, такие как условия сходимости и завершения, используемые структуры данных и вторичные аргументы известных подалгоритмов.
2. Алгоритмы в различных публикациях с описанными основными подробностями. Программы в этой книге обычно имеют минимальный уровень 2, достаточный только для базовой надежности и эффективности. То же самое для уровня 3, где выполняются лишь простые проверки.
3. Алгоритмы в различных широко используемых библиотеках — они, как правило, более сложные, чем в пункте (2), поскольку в них учитываются специфичные для языка и некоторые независимые от языка проблемы, — такие как проверка входных данных и т. п.

Часто создают *метаалгоритм*, т. е. алгоритм уровня 1, усиленный логикой уровня 2, состоящей из других алгоритмов уровня 1. Но металогики обычно гораздо более эвристичны, чем основной алгоритм, и у нее тоже есть проблемы, связанные с ошибками аппроксимации, оценивания и оптимизации.

Общепринятой хорошей практикой в программной инженерии является разработка *спецификации* алгоритма, состоящей как минимум из пред- и постусловий. Но для численных алгоритмов такие спецификации невозможны, за исключением нескольких основных, т. к. из-за различных типов ошибок возникают определенные проблемы. Обычно в документации хорошего API упоминается основной алгоритм, используемый для решения проблемы, с расчетом на то, что пользователь сможет примерно определить, насколько код надежен, но в этих случаях описываются только детали уровня 1. Ожидается, что алгоритм дает хорошую оценку ответа и его ошибки, если только требуемая точность не будет удовлетворена. То есть, как и в случае статистического вывода, необходимо применять научное суждение, чтобы понять, насколько можно доверять выводам, и в большинстве случаев только всестороннее тестирование интересующих проблем позволяет добиться доверия. В общем случае идеальные оценки ошибок невозможны (см. [22.41]).

В то время как большинство численных алгоритмов являются стандартными и могут иметь стандартные простые реализации, повышение надежности происходит редко, и многие реализации в конечном итоге сходятся к какому-то одному выбору. В этой книге я попытался объяснить логику любого выбора. Рано или поздно в литературе будет описана логика надежности многих алгоритмов. Лучшим исследовательским ре-

шением является изучение доступных хорошо протестированных реализаций. В работе [22.26] описаны свойства надежности нескольких алгоритмов, а в работе [22.33] приведены примеры проверки реализации библиотеки на надежность (в последней работе рекомендуется применять так называемую логику чрезмерной надежности).

Чтобы алгоритм был признан пригодным для включения в высококачественную библиотеку, нужно максимально уменьшить фактор надежды и молитвы и обеспечить выполнение следующих условий:

- ◆ выбранный базовый алгоритм обладает хорошими теоретическими свойствами — т. е. быстро сходится и устойчив;
- ◆ гарантируется его завершение, если завершаются все подалгоритмы, — неверные входные данные приводят к сообщениям об ошибках, а не к сбоям;
- ◆ код написан в защищенном стиле — например, обрабатываются неожиданные значения  $\infty$ , NaN и возможные значения вне допустимого диапазона во входных данных и *информационных запросах* (вызовы функций или производных в терминологии информационной сложности) во время вычислений. Следует ожидать, что любой вызов функции или даже простая операция может выдать исключение (представьте себе программу на C++, в которой каждый оператор перегружен и может выдать исключение из-за таких вещей, как нехватка памяти), и не только из-за арифметики с плавающей точкой;
- ◆ круг решаемых алгоритмом задач максимально расширен. Часто бывает трудно выбрать входные данные с плавающей точкой, для которых возвращается достаточно хорошее решение. Разные трюки — такие как предотвращение переполнения/недостаточного заполнения, позволяют расширить этот набор, но все же не помогают его идентифицировать. Например, может быть стоит проверить переполнение в  $x^2$ , а не в  $2x$ . Технически этот метод работает не для всех возможных входных данных, но он быстрее и проще почти для всех полезных входных данных;
- ◆ алгоритм обнаруживает неверные входные данные, особенно когда решения не существует или входные данные не удовлетворяют требуемым условиям. Некоторые алгоритмы также оценивают числа обусловленности. Но помните, что проверка структурных предварительных условий плохо обусловлена, поэтому выполняются такие проверки с малой точностью. Общий шаблон состоит в том, чтобы в любом случае выполнять основные вычисления, если это не слишком дорого, и обрабатывать результат NaN, если он получен, — это безопаснее, чем проверка ввода, но может выполняться в дополнение к ней. Все проверки, которые можно разумно провести, должны быть сделаны;
- ◆ алгоритм обеспечивает оценку ошибки — в большинстве случаев просто консервативную эвристику, которая является разумным предположением. В общем случае достоверно оценить погрешность во всех ситуациях невозможно. Например, любые адаптивные критерии сходимости для таких задач, как интегрирование (см. главу 23. *Численные алгоритмы: работа с функциями*), можно обмануть. Поэтому нельзя гарантировать, что метод сообщит о большой или небольшой ошибке. Здесь тоже нужно проявлять осторожность — например, ответы, равные  $\infty$  и NaN, не могут иметь конечной погрешности. Также следует подумать о том, чтобы проверить, удовлетворяет ли ответ некоторым условиям — таким как проверка диапазона, где это уместно;

- ◆ API алгоритма хорошо спроектирован — у алгоритма минимум параметров, и по возможности для них заданы разумные значения по умолчанию. В целом выбор стоит между надежностью и удобством/эффективностью, потому что универсальный интерфейс не может дать и того и другого. Например, если подалгоритм имеет параметр с хорошим значением по умолчанию, следует ли его включать в интерфейс вызова? Это частая проблема метаалгоритмов — разные методы решения одной и той же задачи делают разные предположения, и поэтому им часто требуются разные входные данные. Что лучше делать в случае различающихся входных данных: запрашивать у пользователя надмножество или принимать для подалгоритмов значения по умолчанию? Может показаться, что при наличии двух функций, одна из которых принимает надмножество, а другая вызывает первую со значениями по умолчанию, вы можете сделать что угодно, но вычисление надмножества может оказаться неэффективным или громоздким, поэтому для полноты может потребоваться несколько функций, что противоречит простоте. Например, одному решателю дифференциальных уравнений может понадобиться якобиан уравнений, а другому — нет. В работе [22.36] авторы предпочли численно вычислять якобиан там, где это необходимо, не спрашивая пользователя;
- ◆ алгоритм должен избегать чрезмерной неэффективности. По крайней мере, следует подумать о критериях завершения и выборе алгоритма. К сожалению, может также потребоваться «срезать углы» — например, согласиться на гораздо более дешевую, но менее точную оценку ошибки;
- ◆ алгоритм тщательно протестирован — нужен обширный, хорошо продуманный набор тестов, способный охватить как можно больше различных случаев, даже числа с плавающей точкой «у пределов возможностей машины». Хороший тестовый пример — такой, при котором базовый алгоритм дает точный теоретический ответ, и тесты должны включать некоторые задачи, где ответ близок к наихудшему случаю. У состоящих из одного числа входных данных можно попробовать все числа с плавающей точкой и посмотреть, не обнаружатся ли какие-нибудь сюрпризы. Для больших входных данных можно попробовать случайно сгенерированные слова. Как правило, трудно определить приемлемую точность для прохождения тестового примера. Можно использовать малую погрешность с одинарной точностью и надеяться на успех. Хороший метод проверки состоит в том, чтобы проверить различные алгебраические тождества из ответов на несколько отличающиеся задачи. Например, чтобы протестировать алгоритм интегрирования, нужно проверить, что  $\int_{[a,b]} f + \int_{[b,c]} f = \int_{[a,c]} f$  достигает возвращенных оценок ошибки. В противном случае для проверки ответов может потребоваться арифметика произвольной точности. Кроме того, хочется иметь возможность автоматически решать, достаточно ли точен ответ на тест, но необходимо придерживаться более низкой точности, чтобы на него не влияли тестовые случаи, которые приводят к гораздо меньшей точности, чем типичные.

При разработке максимально надежных алгоритмов было бы полезно понять, как они будут работать в режиме ручного управления. Иногда в этом и заключается лучший способ автоматизации определенных задач. Но ничего из этого не делает алгоритм полностью надежным.



Даже у самой лучшей универсальной библиотеки нет шансов, если:

- ◆ задача очень плохо обусловлена — ее трудно обнаружить, даже если постараться;
- ◆ масштаб задачи неожиданный — он влияет на критерии сходимости, основанные на нормах и ошибках округления меньших переменных;
- ◆ основная функция  $f$  в задаче недостаточно гладкая, обладает сингулярностями или другими чертами особого поведения, вызывающими численные трудности, — обычно их невозможно обнаружить. Например,  $f$  может иметь сингулярность между двумя машинными числами, которая не заметна ни на одном из них (см. [22.46]).

Численные методы обычно теоретически хорошо работают у определенных задач и у более широкого класса задач на практике. Но согласно теории игр у разработчика по-прежнему есть оптимальная рандомизированная стратегия реализации метода. Она не обязательно должна быть известна, но после значительной работы обычно можно определить, что стоит делать, а что нет. Как правило, решение о прекращении дальнейшей разработки оправдано, когда:

- ◆ текущая реализация хорошо протестирована и хорошо работает;
- ◆ было проведено разумное исследование опубликованной литературы, чтобы не пропустить ни одного важного улучшения реализации. Нужен правильный выбор алгоритма, улучшение надежности и тестовые варианты использования, т. е. основная экономическая ценность, которую вносят библиотеки. Особенно трудно найти хорошие наборы тестов, и, как правило, они обсуждаются в некоторых, наиболее часто цитируемых статьях в предметной области. Обосновать все возможные задачи невозможно, как бы ни хотелось. Наборы тестов для нескольких задач в настоящее время, похоже, не создаются, за исключением, возможно, библиотек с открытым исходным кодом. Обычно их необходимо пересматривать каждые несколько лет, а новые результаты требуют повторного тестирования, поэтому библиотеки алгоритмов обычно не включают последние исследования.

Пользователь также может проводить экспериментальную оценку при решении интересующей его проблемы. Например, при заданных входных данных  $x$  можно выполнить анализ чувствительности, рассматривая выполненное вычисление как некоторую функцию  $f$  и вычисляя  $f'(x)$  с помощью метода конечных разностей (см. главу 23. Численные алгоритмы: работа с функциями). Если результаты оказываются неожиданными, это может быть признаком проблемы. Также можно попробовать изучить случайные возмущения  $x$  той же величины, что и значения конечной разности по умолчанию, и посмотреть, как изменится от этого результат. Производная (или субградиент — см. главу 24. Численная оптимизация) либо существует, либо имеет скачки и плохую обусловленность. Перебор номеров машин на 1 ulp также может быть полезен, как и случайные возмущения на  $O(\text{machine } \epsilon)$ .

Пользователь сам решает, когда принимать ту или иную реализацию, и часто это не просто, но лучше, чем вообще не иметь решения из-за требования 100%-ной корректности. Та же логика применима к статистическим алгоритмам и алгоритмам машинного обучения. Сравнивать алгоритмы тоже сложно. Надежные алгоритмы могут работать прямо «из коробки», а ненадежным нужна опытная няня.

Численные методы и многие другие алгоритмические задачи — такие как машинное обучение, криптография, оптимизация, сжатие данных, всегда попадают в библиотеки.

Глупо пытаться использовать собственный код вместо готовых библиотек. Лишь изредка бывает необходимо воспроизводить их на новом языке или специальном оборудовании. Исследователи работают над улучшением библиотек и теоретической базы. В ходе обучения сначала рассматриваются численные методы и научные вычисления, затем обсуждается, как использовать библиотеку и, возможно, некоторые очень простые алгоритмы для всеобщего ознакомления, что полезно для понимания ограничений различных «черных ящиков». Например, анализ условий актуален для любой реализации, а анализ устойчивости зависит от конкретного алгоритма. Кто-то должен писать книги для исследователей о настоящих численных методах, даже несмотря на то, что обсуждаемые реализации находятся где-то между книжными и библиотечными. Даже при недостаточном количестве подробностей это наилучший возможный случай.

*Валидация* численного программного обеспечения выходит за рамки численного анализа, но необходимо проверять, что решаемая модель вообще имеет смысл. Общая закономерность заключается в том, что постоянное совершенствование алгоритмов и аппаратного обеспечения позволяет решать более сложные модели, что уменьшает необходимость использования в моделях искусственных упрощений.

## 22.9. Матричная алгебра

Большинство алгоритмов являются спецификациями методов линейной алгебры. Часто используют матрицы идентичности вида  $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$  (для размера 2 на 2).

Массив и отображение двумерных координат на его индексы представляют собой матрицу. Как и в линейной алгебре, предположим, что это вектор-столбец. Сложности возникают в случае квадратных матриц с  $n$  строками. Хотя тип элемента для общности считается произвольным, весь анализ устойчивости будет проводиться для типа `double`:

```
template<typename ITEM> struct Matrix: public ArithmeticType<Matrix<ITEM> >
{
    int rows, columns;
    int index(int row, int column) const
    {
        assert(row >= 0 && row < rows && column >= 0 && column < columns);
        return row + column * rows;
    }
    Vector<ITEM> items;
public:
    ITEM& operator()(int row, int column){return items[index(row, column)];}
    ITEM const& operator()(int row, int column) const
    {return items[index(row, column)];}
    Matrix(int theRows, int theColumns): rows(theRows), columns(theColumns),
        items(rows * columns) {}
};
```

Умножение на скаляр выполняется за  $O(n)$ . Например:  $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times 2 = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$ .

```
Matrix operator*=(ITEM const& scalar)
{
    items *= scalar;
```

```

    return *this;
}
friend Matrix operator*(ITEM const& scalar, Matrix const& a)
{
    Matrix result(a);
    return result *= scalar;
}
friend Matrix operator*(Matrix const& a, ITEM const& scalar)
{return scalar * a;}

```

Сложение и вычитание выполняются за  $O(n)$ . Например:  $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 0 & -2 \\ -3 & -3 \end{bmatrix}$ .

```

Matrix& operator+=(Matrix const& rhs)
{
    assert(rows == rhs.rows && columns == rhs.columns);
    items += rhs.items;
    return *this;
}
Matrix& operator-=(Matrix const& rhs)
{
    assert(rows == rhs.rows && columns == rhs.columns);
    items -= rhs.items;
    return *this;
}

```

До сих пор свойства устойчивости были выражены просто. И сложение, и умножение на скаляр выполняются поэлементно и наследуют устойчивость базовой арифметики. Но в случае более сложных операций все усложняется. В работе [22.22] приведен наиболее полный справочник по анализу устойчивости, и все ограничения погрешности вычислений, не указанные явно, взяты из него. Также взгляните на оригинальную трактовку в работе [22.40]. В качестве обозначения покомпонентных оценок для вектора  $x$  положим, что  $|x|$  обозначает вектор или матрицу, где  $x_i$  заменено на  $|x_i|$  (не путать с обозначением определителя). Также пусть  $\gamma_n = \frac{n\epsilon_{machine}}{1 + n\epsilon_{machine}}$  и  $fl$ (точное количество) отно-

сятся к вычислению этого количества с плавающей точкой некоторым алгоритмом. Тогда для внутреннего умножения получается устойчивость:  $fl(xy) = (x + \Delta x)y = x(y + \Delta y)$ , где  $\Delta x \leq \gamma_n |x|$  и  $\Delta y \leq \gamma_n |y|$ . Также возможно прямое ограничение:  $|xy - fl(xy)| \leq \gamma_n |x| |y|$ . Плохая отмена может произойти, например, в ситуации, когда  $x = [1, -1 + \epsilon]$ . Но в некоторых случаях — таких как вычисление суммы квадратов, существует гарантированная небольшая прямая ошибка. Для внешнего произведения имеем прямую границу  $x \otimes y - fl(x \otimes y) = \Delta$ , где  $|\Delta| \leq \epsilon_{machine} |x \otimes y|$ , а обратной границы не существует. Можно получить нормированную границу из более общей покомпонентной границы, заменив операцию « $\otimes$ » нормой.

Умножение выполняется за время  $O(n^2)$ . Например,  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$  демонстрирует умножение строк на столбцы.

Умножение на вектор слева является частным случаем этого умножения. Правое умножение эквивалентно левому, но с транспонированием:

```

Matrix& operator*=(Matrix const& rhs)
{
    assert(columns == rhs.rows);
    Matrix result(rows, rhs.columns);
    for(int i = 0; i < rows; ++i) // строка
        for(int j = 0; j < rhs.columns; ++j) // на столбец
        {
            ITEM sum(0);
            for(int k = 0; k < rhs.rows; ++k)
                sum += (*this)(i, k) * rhs(k, j);
            result(i, j) += sum;
        }
    return *this = result;
}

Vector<ITEM> operator*(Vector<ITEM> const& v) const
{
    assert(columns == v.getSize());
    Vector<ITEM> result(rows);
    for(int i = 0; i < rows; ++i)
        for(int j = 0; j < columns; ++j)
            result[i] += (*this)(i, j) * v[j];
    return result;
}

friend Vector<ITEM> operator*(Vector<ITEM> const& v, Matrix const& m)
{
    return m.transpose() * v;
}

```

Умножение матрицы на вектор имеет устойчивость:  $f(Ax) = (A + \Delta A)x$ , где  $|\Delta A| \leq \gamma_n |A|$ . Также известна прямая оценка:  $|Ax - f(Ax)| \leq \gamma_n |A| |x|$ . Как и внешнее произведение, матричное умножение в целом не является обратно устойчивым, если не рассматривать конкретный столбец. А еще есть прямая граница:  $|AB - f(AB)| \leq \gamma_n |A| |B|$ .

Создание и транспонирование выполняются за  $O(n)$ , обе операции совершенно ста-

бильны. Например,  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$ .

```

static Matrix identity(int n)
{
    Matrix result(n, n);
    for(int i = 0; i < n; ++i) result(i, i) = ITEM(1);
    return result;
}

Matrix transpose() const
{
    Matrix result(columns, rows);
    for(int i = 0; i < rows; ++i)
        for(int j = 0; j < columns; ++j) result(j, i) = (*this)(i, j);
    return result;
}

```

Также бывает полезно работать с подматрицами (функция присвоения является функцией-членом, а функция чтения — нет):

```
template<typename ITEM, typename MATRIX> Matrix<ITEM> submatrix(
    MATRIX const& A, int r1, int r2, int c1, int c2)
{
    assert(r1 >= 0 && r2 < A.getRows() && c1 >= 0 && c2 < A.getColumns() &&
        r1 <= r2 && c1 <= c2);
    Matrix<ITEM> result(r2 - r1 + 1, c2 - c1 + 1);
    for(int r = r1; r <= r2; ++r)
        for(int c = c1; c <= c2; ++c) result(r - r1, c - c1) = A(r, c);
    return result;
}
```

Вы можете преобразовывать различные разреженные и неявные матрицы (обсуждаемые далее в этой главе) в плотные:

```
template<typename ITEM, typename MATRIX> Matrix<ITEM> toDense(MATRIX const& A)
{ return submatrix<ITEM>(A, 0, A.getRows() - 1, 0, A.getColumns() - 1); }
void assignSubmatrix(Matrix const& sub, int r1, int c1)
{
    assert(r1 >= 0 && c1 >= 0 && r1 + sub.getRows() <= rows &&
        c1 + sub.getColumns() <= columns);
    for(int r = r1; r - r1 < sub.getRows(); ++r)
        for(int c = c1; c - c1 < sub.getColumns(); ++c)
            (*this)(r, c) = sub(r - r1, c - c1);
}
```

## 22.10. Матричные нормы

Норма должна удовлетворять метрическим свойствам — таким как неравенство треугольника. Обычная конструкция состоит в приведении к соответствующей векторной норме по формуле  $\|A\| = \max_{\|x\|>0} \frac{\|Ax\|}{\|x\|}$ . Она работает для любой векторной нормы. Моя

реализация векторной евклидовой нормы наивна, но разумна, поскольку не порождает проблем с реалистичными входными данными. Можно внести усовершенствование для предотвращения переполнения — исключить самый большой элемент. В работе [22.4] описаны другие усовершенствования.

Полученную  $\infty$ -норму легко вычислить:  $\|A\|_\infty = \max_r \sum_c |A[r, c]|$ . Для вектора  $x$  —  $\|x\|_\infty = \max |x_i|$ . Для комплексных чисел  $|x_i| = \text{radius}(x_i)$  (см. [22.18]). Вычисление 2-нормы намного дороже и выполняется с помощью SVD (обсуждается далее в этой главе):

```
template<typename X> double normInf(Vector<X> const& x)
{ // работает и для комплексных векторов
    int xInf = 0;
    for(int i = 0; i < x.getSize(); ++i)
    {
        double ax = abs(x[i]);
        if(isnan(ax)) return ax; // проверка на NaN перед поиском максимума
        xInf = max(xInf, ax);
    }
}
```

```

    return xInf;
}
double normInf(Matrix<double> const& A)
{
    double m = A.getRows();
    Vector<double> rowSums(m);
    for(int r = 0; r < m; ++r)
        for(int c = 0; c < A.getColumns(); ++c) rowSums[r] += abs(A(r, c));
    return valMax(rowSums.getArray(), m);
}

```

Еще одна распространенная и легко вычисляемая норма — *норма Фробениуса*, определяемая как  $\|A\|_F = \sqrt{\sum_{r,c} A[r,c]^2}$ . В некоторых особых случаях легче работать аналитически, но в остальных случаях эта норма используется редко:

```

double normFrobenius(Matrix<double> const& A)
{
    double sum = 0;
    for(int r = 0; r < A.getRows(); ++r)
        for(int c = 0; c < A.getColumns(); ++c) sum += A(r, c) * A(r, c);
    return sqrt(sum);
}

```

Использование  $\infty$ -нормы для функций и дискретных объектов, таких как векторы и матрицы, работает иначе. У дискретных объектов любая норма находится в пределах зависящего от размера постоянного фактора любой другой нормы, а у функций это не так (см. [22.39]). Например, нормы  $L_p$  определяются интегрированием, которое не чувствительно к поведению  $f$  на множествах меры 0, но влияет на  $\infty$ -норму. На практике это почти никогда не вызывает проблем, поэтому зачастую предпочтение отдается последнему из-за наихудшего значения и обычно простых вычислений.

## 22.11. LUP-разложение

Можно разложить квадратную матрицу  $A$  так, что  $LU = A$ .  $L$  — это нижнетреугольная матрица с 0 элементами на диагонали и над ней, а  $U$  — верхнетреугольная матрица с 0 элементами под диагональю. Это следует из гауссова исключения с поворотом, которое проходят в курсе линейной алгебры.

Транспонирование строки в расширенной матрице не меняет решение матричного уравнения.  $P$  — это матрица перестановок, полученная путем транспонирования строк, но представленная в виде массива. Перестановка выполняется с целью выбрать опорные точки с наибольшим абсолютным значением, чтобы избежать нулевых опорных точек и повысить числовую точность — т. е. вычислить  $LU = PA$ . В реализации одна матрица содержит  $L$  и  $U$ . Метод исключения Гаусса с частичным поворотом вычисляет перестановку за время  $O(n^3)$  (см. [22.11]). В курсе линейной алгебры этот метод преподается для решения матричных уравнений. Алгоритм для матрицы, дополненной правой частью:

1. Для любого столбца:
2. Среди строк  $\leq$  диагональной строки выбрать строку с наибольшим абсолютным значением в качестве опорной.

3. Поменять местами опорную строку с диагональной строкой, если они не совпадают.
4. Для любой строки  $<$  диагональной:
5. Вычесть произведение диагональной строки на коэффициент, необходимый для обнуления значения в столбце

Получится верхнетреугольная матрица  $U$  и некоторая правая часть, которая упрощает решение уравнения. Например, опуская дополнение, для матрицы  $3 \times 3$  получаем:

$$\begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \end{bmatrix} \rightarrow \begin{bmatrix} x & x & x \\ 0 & x & x \\ 0 & x & x \end{bmatrix} \rightarrow \begin{bmatrix} x & x & x \\ 0 & x & x \\ 0 & 0 & x \end{bmatrix}.$$

В случае с LUP нужно дополнить правую часть  $L$  единичной матрицы и применить к ней обновления. Помимо отдельного массива для хранения перестановки, нужно поместить  $U$  и  $L$  в одну и ту же матрицу для экономии места.

Для единственного числа  $A$  опорная точка равна нулю, но мы все равно переходим к следующему шагу (см. [22.21]). Декомпозиция выполняется все так же правильно, но из-за сингулярности она вряд ли будет полезна. Определение сингулярности не обязательно должно быть точным и допускает «почти сингулярность». Возможно, наиболее устойчивый метод — это использовать SVD для вычисления ранга  $A$  (обсуждается далее в этой главе):

```
template<typename ITEM = double> struct LUP
{
    Matrix<ITEM> d;
    Vector<int> permutation;
    bool isSingular;
    LUP(Matrix<ITEM> const& a): d(a), isSingular(false)
    {
        assert(d.rows == d.columns); // сперва создается единичная матрица P
        for(int i = 0; i < d.rows; ++i) permutation.append(i);
        for(int i = 0; i < d.rows; ++i)
        {
            ITEM p = 0;
            int entering = -1;
            for(int j = i; j < d.rows; ++j)
                if(abs(d(i, j)) > p)
                {
                    p = abs(d(i, j));
                    entering = j;
                }
            if(entering == -1)
            {
                isSingular = true;
                continue;
            }
            swap(permutation[i], permutation[entering]);
            for(int j = 0; j < d.rows; ++j) swap(d(i, j), d(entering, j));
            for(int j = i + 1; j < d.rows; ++j)
            {
                d(j, i) /= d(i, i);
            }
        }
    }
};
```

```

        for(int k = i + 1; k < d.rows; ++k)
            d(j, k) -= d(j, i) * d(i, k);
    }
}
};

```

Частичный разворот на занятиях линейной алгебры обычно называется просто «разворот». Существует также отсутствие разворота и *полный разворот* (см. [22.21]). Базовая устойчивость для всех вариантов равна  $\|L\| \|U\|_\infty \leq (1 + 2(n^2 - n)\rho_n) \|AP\|_\infty$ , где  $\rho_n$  — это *фактор роста*. Существует известный статистический феномен, который заключается том, что на практике как частичный, так и полный поворот имеют тенденцию к низкому  $\rho_n$ , хотя для последнего  $\rho_n \leq O(\sqrt{nn^{1/4 \log(n)}})$ , а для первого  $\rho_n \leq 2^{n-1}$ . Таким образом, во всех основных реализациях используются частичные, а не полные развороты, поскольку они эффективнее.

Определитель равняется  $\prod$  диагональных элементов  $U$ . Его знак меняется, когда при перестановке четный элемент отображается в нечетный или наоборот. Математически определитель важен, но численно — нет. Например, если матрица  $A$  масштабируется константой  $c$ , ее определитель масштабируется как  $c^n$ , поэтому проверка того, что определитель  $<$  некоторого  $\varepsilon$ , бессмысленна. Когда определитель нужен по математическим причинам, обычно лучше использовать его логарифм, чтобы избежать недополнения/переполнения:

```

double logAbsDet() const // -inf при нуле
{
    double result = 0;
    for(int i = 0; i < d.rows; ++i) result += log(abs(d(i, i)));
    return result;
}

int signDet() const
{
    int sign = 1;
    for(int i = 0; i < d.rows; ++i)
    {
        if(d(i, i) < 0) sign *= -1;
        if(permutation[i] % 2 != i % 2) sign *= -1;
    }
    return sign;
}

```

В работе [22.22, с. 279] приведено обсуждение вопросов устойчивости.

*Обратная замена* решает уравнение  $Ux = b$ :

```

Vector<double> backsubstitution(Matrix<double> const& U, Vector<double> b)
{ // перезаписываем b решением уравнения Ux = b
    int n = b.getSize();
    assert(U.getRows() == n && U.getColumns() == n);
    for(int i = n - 1; i >= 0; --i)
    {
        for(int j = i + 1; j < n; ++j) b[i] -= b[j] * U(i, j);
    }
}

```



```

        b[i] /= U(i, i);
    }
    return b;
}

```

Функция применяется ко всем разложениям, которые могут решать уравнения — такие как QR (обсуждается позже в этой главе). Устойчивость достигается при решении с неособой треугольной системой  $T = U$  или  $T = L$ :  $(T + \Delta T)f_l(x) = b$ , где  $|\Delta T| \leq \gamma_n |T|$ .

Также можно получить покомпонентную прямую ошибку:

$$\frac{\|x - f_l(x)\|_\infty}{\|x\|_\infty} \leq \frac{\text{cond}(T, x) \gamma_n}{1 - \text{cond}(T) \gamma_n},$$

где

$$\text{cond}(A, x) = \frac{\|A\| \|A^{-1}\| \|x\|_\infty}{\|x\|_\infty} \text{ и } \text{cond}(A) = \|A\| \|A^{-1}\|_\infty.$$

Эти покомпонентные числа состояния отличаются от основанных на норме  $\kappa(A)$ .

Передняя замена решает уравнение  $Ax = b$  за время  $O(n^2)$ , решая  $Ly = Pb$ , а затем  $Ux = y$ :

```

Vector<ITEM> solve(Vector<ITEM> const& b) const
{
    Vector<ITEM> y(d.rows);
    for(int i = 0; i < d.rows; ++i)
    {
        y[i] = b[permutation[i]];
        for(int j = 0; j < i; ++j) y[i] -= y[j] * d(i, j);
    }
    return backsubstitution(d, y);
}

```

У полного решения уравнения граница устойчивости равна  $(AP + \Delta AP)f_l(x) = b$ , где  $|\Delta AP| \leq \gamma_{3n} \|f_l(L)\| \|f_l(U)\|$ . Анализ условий, описывающий изменение решения при изменении  $A$  и  $b$ , ведет к:

$$\frac{\|\Delta x\|}{\|x\|} \leq \left( \frac{\kappa(A)}{1 - \kappa(A) \|\Delta A\| / \|A\|} \right) \left( \frac{\|\Delta A\|}{\|A\|} + \frac{\|\Delta b\|}{\|b\|} \right),$$

где  $A$  является несингулярной,  $b$  не равно нулю и  $\|\Delta A\| < \frac{1}{\|A^{-1}\|}$  (см. [22.12, с. 65]). По-

компонентный анализ приведен в работах [22.12, с. 143 и 22.22, с. 123]).

Дополнение матрицы единичными значениями того же размера вычисляет  $A^{-1}$  за время  $O(n^3)$ . Для любого единичного столбца вычисление решает матрично-векторное уравнение (это относится и к другим разложениям):

```

template<typename DECOMPOSITION> Matrix<double> inverse(DECOMPOSITION const& d,
    int n)
{
    Vector<double> identityRow(n, 0);
    Matrix<double> result(n, n);
}

```

```

for(int i = 0; i < n; ++i)
{
    identityRow[i] = 1;
    Vector<double> column = d.solve(identityRow);
    identityRow[i] = 0;
    for(int j = 0; j < n; ++j) result[j, i] = column[j];
}
return result;
}

```

Избегайте обратных вычислений, т. к. они ведут к ненужным потерям точности. Лучше всего работать с LUP напрямую. В работах [22.46 и 22.12, с. 137] можно прочесть о некоторых приемах, позволяющих избежать инверсии. Когда нужно ее вычислить, оценка устойчивости равна:

$$\|A f(A^{-1}) - I\| \leq \gamma_{O(n)} \|f(L)\| \|f(U)\| \|f(A^{-1})\|.$$

После решения уравнения  $Ax = b$  можно вычислить *невязку*  $r = Ax - b$ , где  $r$  — наблюдаемая обратная ошибка, т. е. не худший случай. Решение уравнения LUP устойчиво, за исключением очень маловероятных худших случаев, поэтому  $\|r\| = O(\epsilon_{\text{machine}})$  для любой нормы, если  $A$  не является сингулярной. В частности, имеем  $\frac{\|\Delta x\|}{\|x\|} \leq \kappa(A) \frac{\|r\|}{\|b\|}$  (см. [22.12, с. 140]).

Обычно требуется получить оценку ошибки для  $x$  — т. е. прямую ошибку. Если истинное решение равно  $x^*$ ,  $x - x^* = A^{-1}r$ , поэтому  $\|x - x^*\| \leq \|A^{-1}\| \|r\|$ , что является хорошей оценкой абсолютной ошибки. Относительная ошибка равна  $\frac{\|x - x^*\|}{\|x^*\|}$ , а из неравенства

$\|A\| \|x^*\| \geq \|b\|$  обратная ошибка равна  $\frac{\|x - x^*\|}{\|x^*\|} \leq \kappa(A) \frac{\|r\|}{\|b\|}$ , что является хорошей оценкой

для относительной ошибки.  $\kappa(A) = \|A\| \|A^{-1}\|$  — это (относительное) *условное число*  $A$ .

Вычисление  $\|A^{-1}\|_\infty$  для оценок  $\kappa(A)$  быстро не выполняется. Стандартной альтернативой является оценка  $\kappa(A)$  другими способами. В работе [22.22] описаны некоторые алгоритмы, но они сложны и не обсуждаются далее. SVD позволяет точно вычислить  $\|A^{-1}\|_2$  (обсуждается далее в этой главе), поэтому, когда нужна оценка ошибки, обычно используют  $\kappa_2(A)$ :

```

pair<double, double> estimateMatrixEquationError2Norm(Matrix<double> const& A,
    Vector<double> const& b, Vector<double> const& x)
{ // абсолютное, относительное
    SVD svd(A);
    double nr = norm(A * x - b);
    return make_pair(nr * svd.norm2Inv(), nr * svd.condition2() / norm(b));
}

```

Однако эта оценка дорогая, имеет время выполнения  $O(n^3)$  и гораздо большую константу, чем в случае с LUP (см. оценки SVD в работе [22.21]), поэтому использовать ее следует только при необходимости.

## 22.12. Разложение Холецкого

Если матрица  $A$  симметрична и положительно определена, можно вычислить нижне-треугольную матрицу  $L$  такую, что  $A = LL^T$ . Это полезно, например, для многомерных ковариационных матриц.

$L$  вычисляется столбец за столбцом:

1. Для любого  $c$  и  $r \geq c$ :
2. 
$$s = A(c, c) - \sum_{0 \leq k < c} L(r, k)L(c, k).$$
3. 
$$L(c, c) = \sqrt{s}.$$
4. 
$$L(r, c) = \frac{s}{L(c, c)}.$$

```
template<typename ITEM> struct Cholesky
{
    Matrix<ITEM> l;
    bool failed;
    Cholesky(Matrix<ITEM> const& a): l(a.rows, a.columns), failed(false)
    { // матрица a должна быть симметричной и положительно определенной
        for(int c = 0; c < l.columns; ++c)
            for(int r = c; r < l.rows; ++r)
            {
                ITEM sum = a(r, c);
                for(int k = 0; k < c; ++k) sum -= l(r, k) * l(c, k);
                if(r == c)
                {
                    if(sum <= 0){failed = true; return;}
                    l(c, c) = sqrt(sum);
                }
                else l(r, c) = sum/l(c, c);
            }
    }
};
```

Вычисление выполняется за время  $O(n^3)$ . Устойчивость алгоритма:

$$fl(L)fl(L)^T = A + \Delta A, \text{ где } |\Delta A| \leq \gamma_{n+1} |fl(L)| |fl(L)^T|.$$

Разложение Холецкого можно использовать в том числе для вычисления определителей и решения уравнений:

- ◆  $\det(A) = \det(L)^2$ .
- ◆ чтобы решить уравнение  $Ax = b$ , можно решить  $L^T x = (Lx = b)$ . Для последнего используется обратная замена.

```
double logDet()const // -inf при нуле
{
    assert(!failed);
    double result = 0;
```

```

    for(int i = 0; i < l.rows; ++i) result += log(abs(l(i, i)));
    return 2 * result;
}
Vector<ITEM> solve(Vector<ITEM> b) const
{
    int n = b.getSize();
    assert(l.getRows() == n && l.getColumns() == n);
    assert(!failed);
    for(int i = 0; i < n; ++i)
    {
        for(int j = 0; j < i; ++j) b[i] -= b[j] * l(i, j);
        b[i] /= l(i, i);
    }
    return backsubstitution(l.transpose(), b);
}

```

Решение уравнения имеет оценку устойчивости:  $(A + \Delta A)f_l(x) = b$ , где  $|\Delta A| \leq \gamma_{3n+1} \|f_l(L)\| \|f_l(L)^T\|$ .

## 22.13. Ленточные матрицы

Матрицы часто бывают симметричны и равны 0, за исключением нескольких диагона-

лей, близких к главной, — например:  $\begin{bmatrix} 1 & 2 & 0 \\ 3 & 4 & 5 \\ 0 & 6 & 7 \end{bmatrix}$ , поэтому их можно представить более

эффективно. Типичным примером являются *тридиагональные матрицы*. В реализации описаны только некоторые функции (неполный набор общих матричных функций):

```

template<int D = 3, typename ITEM = double> struct BandMatrix
{ // ниже главной диагонали d < 0
    Vector<Vector<ITEM> > diagonals;
    int diagIndex(int d) const { return D/2 + d; } // "0" - это главная диагональ
    int rowIndex(int r, int d) const { return r + min(0, d); }
    int diag(int r, int c) const { return c - r; } // индекс хранения
    int getRows() const { return diagonals[diagIndex(0)].getSize(); }
    int getColumns() const { return getRows(); }
    void setupBands(int n)
    { // инициализация ленточных частей
        assert(n > 0 && D % 2 == 1);
        for(int d = -D/2; d <= D/2; ++d)
            diagonals[diagIndex(d)] = Vector<ITEM>(n - abs(d));
    }
    BandMatrix(int n): diagonals(D) { setupBands(n); }
    BandMatrix(Matrix<ITEM> const& A): diagonals(D)
    { // копирование ленточной части
        int n = A.getRows();
        setupBands(n); // эта функция вызывается первой
        assert(n == A.getColumns());
        for(int r = 0; r < n; ++r)
            for(int c = max(0, r - D/2); c <= min(n - 1, r + D/2); ++c)

```

```

        (*this)(r, c) = A(r, c);
    }
    ITEM operator()(int r, int c) const
    {
        int d = diag(r, c);
        assert(r >= 0 && r < getRows() && c >= 0 && c < getColumns());
        return abs(d) <= D/2 ? diagonals[diagIndex(d)][rowIndex(r, d)] : 0;
    }
    ITEM& operator()(int r, int c)
    {
        int d = diag(r, c);
        assert(r >= 0 && r < getRows() && c >= 0 && c < getColumns() &&
            abs(d) <= D/2);
        return diagonals[diagIndex(d)][rowIndex(r, d)];
    }
    static BandMatrix identity(int n)
    {
        BandMatrix result(n);
        result.diagonals[result.diagIndex(0)] = Vector<ITEM>(n, 1);
        return result;
    }
    BandMatrix& operator+=(BandMatrix const& A)
    {
        assert(getRows() == A.getRows());
        diagonals += A.diagonals;
        return *this;
    }
    BandMatrix& operator*=(ITEM const& a)
    {
        diagonals *= a;
        return *this;
    }
    friend BandMatrix operator*(ITEM const& scalar, BandMatrix const& A)
    {
        BandMatrix result(A);
        return result *= scalar;
    }
    BandMatrix operator-() const
    {
        BandMatrix result(*this);
        return result *= -1;
    }
    BandMatrix& operator-=(BandMatrix const& A) {return *this += -A;}
    void assignSubmatrix(Matrix<ITEM> const& sub, int r1, int c1)
    { // присвоение только тридиагональной части
        assert(r1 >= 0 && c1 >= 0 && r1 + sub.getRows() <= getRows() &&
            c1 + sub.getColumns() <= getColumns());
        for(int r = r1; r - r1 < sub.getRows(); ++r)
            for(int c = c1; c - c1 < sub.getColumns(); ++c)
                if(abs(diag(r, c)) <= D/2) (*this)(r, c) = sub(r - r1, c - c1);
    }
};

template<typename ITEM> using TridiagonalMatrix = BandMatrix<3, ITEM>;

```

## 22.14. Решение тридиагональных матричных уравнений

Такие матрицы встречаются в специальных алгоритмах — таких как вычисление кубических сплайн-интерполянтов и некоторые методы решения уравнений в частных производных (в следующей главе описаны и те и другие).

Простой подход к решению — исключение Гаусса без поворота (часто называемое *алгоритмом Гаусса — Томаса*), где диагональная запись используется в качестве опорной, а ее строка — для исключения нижних строк:

```
Vector<double> solveTridiag(TridiagonalMatrix<double> const& A,
    Vector<double> r)
{ // U - вышележащая диагональ, D - диагональ, L - нижележащая диагональ.
  // R перезаписывается решением
  Vector<double> U = A.diagonals[2], D = A.diagonals[1];
  Vector<double> const& L = A.diagonals[0];
  int n = r.getSize();
  assert(n > 1 && D.getSize() == n);
  // прямое удаление
  U[0] /= D[0];
  r[0] /= D[0];
  for(int i = 1; i < n; ++i)
  {
    double denom = D[i] - L[i - 1] * U[i - 1];
    if(i < n - 1) U[i] /= denom; // ноль в знаменателе допускается.
    // Пользователь сам обрабатывает значения inf или NaN
    r[i] = (r[i] - L[i - 1] * r[i - 1])/denom;
  } // обратная замена
  for(int i = n - 2; i >= 0; --i) r[i] -= U[i] * r[i + 1];
  return r;
}
```

Время выполнения равно  $O(n)$ . Разворот нарушил бы тридиагональную структуру, но без него алгоритму нужно *диагональное доминирование* для устойчивости (каждая диагональная запись  $\geq \sum |\text{других записей в своей строке}|$ ). Это позволяет избежать отмены диагональных элементов и последующего деления на 0. Пользователь должен убедиться, что это условие выполняется, прежде чем применять метод.

Исключение без разворота легко распространяется на более общие ленточные матрицы, хотя производительность будет зависеть от количества диагоналей, и диагональное доминирование становится менее вероятным. Решение диагонально доминирующих систем методом исключения Гаусса без поворота является устойчивым, поскольку для неявно вычисляемого LU разложение равно  $\|L\|U\|_{\infty} \leq (2n - 1)\|A\|_{\infty}$ .

## 22.15. Ортогональные преобразования

В разложении LUP для преобразования матриц используются элементарные операции со строками. Они соответствуют умножениям на определенные матрицы, последовательность которых приводит к полезным вычислениям. Отогональное умножение мат-

риц тоже дает полезные преобразования. Матрица  $Q$  ортогональна, если  $QQ^T = I$ . Для ортогональной матрицы число обусловленности равно 1 (см. [22.42]) и последовательность умножений тоже имеет число обусловленности 1. Умножение на ортогональную матрицу устойчиво (см. [22.12, с. 40]). Алгоритмы на основе ортогональных последовательностей матриц, как правило, устойчивы, хотя это явно и не следует из изложенного.

*Сокращение Хаусхолдера* (также называемое *отражением* или *преобразованием*) — это такое умножение на ортогональную матрицу, что в результате все элементы ниже указанного элемента становятся равны нулю. Его также можно применять к подматрицам. В частности, пусть  $x$  будет вектором  $A$ , в котором нужно удалить все записи, кроме первой. Затем (см. [22.22]) матрица Хаусхолдера:

$$H = I - \left( \frac{2}{\|v\|} \right) v \otimes v,$$

где  $v = x$ , кроме  $v[0] = x[0] + \text{sign}(x[0])\|x\|$ . Здесь значение  $x_0 = 0$  не вызывает проблем — знаковая функция не является неопределенной в 0, а представляет собой субградиент (см. главу 24. Численная оптимизация)  $f(x) = |x|$ . Эта формула так же стабильна, как и формула, приведенная в работе [22.21], но намного проще. Для удобства предусмотрен особый случай для  $x$  размера 2.

Для эффективности  $H$  не формируется явно, поэтому умножение  $x$  размера  $n$  выполняется за время  $O(n^2)$ :

$$\blacklozenge \quad HA = \left( I - \left( \frac{2}{\|v\|} \right) v \otimes v \right) A = A - \left( \frac{2}{\|v\|} \right) v \otimes vA;$$

$$\blacklozenge \quad AH = A - A \left( \frac{2}{\|v\|} \right) v \otimes v.$$

```
double sign(double x){return x > 0 ? 1 : -1;}
Vector<double> HouseholderReduction(Vector<double> x)
{
    x[0] += sign(x[0]) * norm(x);
    double normX = norm(x);
    if(normX > 0) x *= 1/normX;
    return x;
}
Vector<double> HouseholderReduction2(double a, double b)
{
    Vector<double> x;
    x.append(a);
    x.append(b);
    return HouseholderReduction(x);
}
template<typename MATRIX> void HouseholderLeftMult(MATRIX& M,
    Vector<double> const& h, int r, int c = 0, int c2 = -1)
{
    if(c2 == -1) c2 = M.getColumns() - 1;
    assert(r + h.getSize() <= M.getRows());
```

```

    Matrix<double> sub = submatrix<double>(M, r, r + h.getSize() - 1, c, c2);
    sub -= outerProduct(h * 2, h * sub);
    M.assignSubmatrix(sub, r, c);
}
template<typename MATRIX> void HouseholderRightMult(MATRIX& M,
    Vector<double> const& h, int c, int r = 0, int r2 = -1)
{
    if(r2 == -1) r2 = M.getRows() - 1;
    Matrix<double> sub = submatrix<double>(M, r, r2, c, c + h.getSize() - 1);
    sub -= outerProduct(sub * h, h * 2);
    M.assignSubmatrix(sub, r, c);
}

```

В целом операции с внешними произведениями более эффективны, если выполняются неявно. Умножение матрицы на вектор — это еще один такой случай. Пусть  $B = A + u \otimes v$ . Тогда  $Bx = Ax + u(vx)$  и  $xB = xA + (xu)v$ :

```

Vector<double> outerProductMultLeft(Vector<double> const& u, // столбец u,
                                     // строка v
    Vector<double> const& v, Vector<double> const& x)
{return u * dotProduct(v, x);}
Vector<double> outerProductMultRight(Vector<double> const& u, // столбец u,
                                     // строка v
    Vector<double> const& v, Vector<double> const& x)
{return v * dotProduct(x, u);}

```

Умножение  $A$  на последовательность матриц Хаусхолдера является неявно устойчивым (см. [22.22, с. 359]).

## 22.16. QR-разложение

Подобно LUP создание  $A = Q^T R$ , где матрица  $R$  является верхнетреугольной, а  $Q$  — ортогональной, позволяет более эффективно выполнять многие последующие операции — например, вычисление собственных значений.

Алгоритм общего случая проходит столбец за столбцом, используя сокращение Хаусхолдера  $H_i$  для обнуления элементов ниже диагонали, как в алгоритме Гаусса. Тогда  $R = HA$  и  $Q = HI$ , где  $H = \prod_{n-1 \geq i \geq 0} H_i$ . Понадобится время  $O(n^3)$ .

Матрица представлена в *форме Хессенберга*, когда ниже первой поддиагонали она заполнена нулями, — например:

$$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \end{bmatrix}.$$

Последовательность  $n$  ортогональных преобразований сверху вниз устраняет поддиагональ за время  $O(n^2)$ :

```

struct QRDecomposition
{
    Matrix<double> Q, R;

```



```

void doHessenbergQR()
{
    for(int c = 0; c < R.getRows() - 1; ++c)
    {
        Vector<double> x = HouseholderReduction2(R(c, c), R(c + 1, c));
        HouseholderLeftMult(R, x, c, c);
        HouseholderLeftMult(Q, x, c);
    }
}

QRDecomposition(Matrix<double> const& A, bool isHessenberg = false):
    Q(Matrix<double>::identity(A.getRows())), R(A)
{
    int n = R.getRows();
    assert(R.getColumns() == n);
    if(isHessenberg) doHessenbergQR(); // уже в форме Хессенберга
    else // обычный случай
        for(int c = 0; c < n - 1; ++c)
        {
            Vector<double> x(n - c);
            for(int j = c; j < n; ++j) x[j - c] = R(j, c);
            x = HouseholderReduction(x);
            HouseholderLeftMult(R, x, c, c);
            HouseholderLeftMult(Q, x, c);
        }
}
};

```

QR-разложение устойчиво, как и решение уравнений с его помощью (см. [22.22, с. 360–361]).

Решение уравнений и вычисление определителей выполняется почти так же, как и у LUP. Но у матрицы QR не известен знак определителя, потому что  $\det(Q) = 1$  или  $-1$ , а  $\det(QQ^T) = 1$ :

```

Vector<double> solve(Vector<double> const& b) const
{ // Решение Rx = Qb для решения Ax = b
    assert(Q.getRows() == b.getSize());
    return backsubstitution(R, Q * b);
}

double logAbsDet() const // -inf при нуле
{
    double result = 0;
    for(int i = 0; i < R.rows; ++i) result += log(abs(R(i, i)));
    return result;
}

```

QR допускает  $O(n^2)$  обновлений ранга 1 — т. е. добавление внешнего произведения. Пусть  $A = Q^T R$  (в работе [22.21] авторы используют  $Q$  вместо  $Q^T$ ). Затем  $A + u \otimes v = Q^T (R + w \otimes v)$  для  $w = Qu$ . Если применить последовательность ортогонального преобразования матрицы  $Q_w$ , получаем  $(QQ_w)^T (Q_w R + Q_w w \otimes v)$ . Пусть  $Q_w$  состоит из последовательности, исключаяющей последний ненулевой элемент  $w$  один за другим. После этого только первый элемент  $Q_w w$  отличен от нуля, поэтому  $Q_w w \otimes v$  равно 0, за исключением первой строки. А матрица  $Q_w R$  находится в форме Хессенбер-

га из-за характера обновлений, поэтому сумму можно эффективно разложить.  $Q$  не обязательно должна начинаться как единичная:

```
void rank1Update(Vector<double> const& u, Vector<double> const& v)
{ // Сперва вычисляется  $Q_{wR}$ 
  int n = u.getSize();
  assert(v.getSize() == n && Q.getRows() == n);
  Vector<double> w = Q * u;
  for(int r = n - 1; r > 0; --r)
  { // Поиск ортогонального преобразования для удаления  $w[r]$ 
    Vector<double> x = HouseholderReduction2(w[r - 1], w[r]);
    Matrix<double> W(2, 1); // применение преобразования
                           // к временной подматрице
    W(0, 0) = w[r - 1];
    W(1, 0) = w[r];
    HouseholderLeftMult(W, x, 0);
    w[r - 1] = W(0, 0); // копирование результата
    w[r] = W(1, 0);
    // применение также к  $R$ 
    HouseholderLeftMult(R, x, r - 1, r - 1);
    HouseholderLeftMult(Q, x, r - 1);
  }
  for(int c = 0; c < n; ++c) R(0, c) += w[0] * v[c];
  doHessenbergQR(); // обработка формы Хессенберга для  $R$ 
}
```

## 22.17. Собственные значения и собственные векторы симметричной матрицы

Для матрицы  $A$  размером  $n \times n$  (см. [22.50]):

- ◆ *собственное значение*  $\lambda$  таково, что  $\det(A - \lambda I) = 0$ ;
- ◆ *собственный вектор*  $v$ , соответствующий  $\lambda$ , таков, что  $Av = \lambda v$ ;
- ◆ собственные значения и собственные векторы вещественной симметричной матрицы  $A$  вещественны, а в противном случае могут быть комплексными;
- ◆ пусть  $V$  — матрица, столбцами которой являются  $n$  линейно независимых собственных векторов, соответствующих собственным значениям, составляющим диагональ диагональной матрицы  $\Lambda$ , если они существуют. Тогда  $A = V\Lambda V^{-1}$  — это *спектральное разложение*  $A$ .

Поскольку собственные значения являются корнями полинома, для  $n > 4$  не существует алгоритма их вычисления за конечное время. Это значит, что алгоритмы должны быть итеративными. Обычно используют *смещенную итерацию QR* (см. [22.21]):

1. Преобразование  $A$  к форме Хессенберга (для эффективности).
2. Пока алгоритм не сойдется:
3. Сократить  $A$  (удалить сходящуюся нижнюю правую подматрицу, если она есть — пояснения будут приведены позже).

4. Выбрать сдвиг  $\mu$ , который вызывает быстрое сокращение.
5.  $QR = A - \mu I$ .
6.  $A = RQ$ .

Алгоритм вычисляет  $D = QAQ^T$  для  $Q = \prod Q_i$ , представляющую собой совокупность ортогональных преобразований (во многих источниках используется эквивалент  $D = Q^T A Q$ ). Различные случаи приводят к различным типам  $D$ . Для реального симметричного  $A$ :

- ◆  $A = D$ ;
- ◆  $V = Q^T$  и  $V^{-1} = Q$ ;
- ◆ использование *сдвигов Уилкинсона* (обсуждается позже) гарантирует сходимость (см. [22.42]);
- ◆ форма Хессенберга является тридиагональной из-за симметрии (постоянный коэффициент эффективности сохраняется, но здесь не задействуется для повторного использования кода. Оптимизированное преобразование приведено в работе [22.21].

Первым шагом является преобразование в форму Хессенберга. Любую квадратную матрицу  $A$  можно преобразовать за  $O(n^3)$  операций, в результате чего  $H = QAQ^{-1}$  (пока что для  $Q$ ). В работе [22.21] приведено обоснование алгоритма:

```
pair<Matrix<double>, Matrix<double>> toHessenbergForm(Matrix<double> A)
{
    int n = A.getRows();
    assert(A.getColumns() == n);
    Matrix<double> Q = Matrix<double>::identity(n);
    for(int c = 0; c < n - 2; ++c)
    {
        Vector<double> x(n - (c + 1));
        for(int j = c + 1; j < n; ++j) x[j - (c + 1)] = A(j, c);
        x = HouseholderReduction(x);
        HouseholderLeftMult(A, c + 1, c, x);
        HouseholderLeftMult(Q, c + 1, c, x);
        HouseholderRightMult(A, 0, c + 1, x);
    }
    return make_pair(A, Q);
}
```

Сдвиг Уилкинсона для текущей итерации равен наименьшему собственному значению нижней правой подматрицы размером  $2 \times 2$ . Используется стабильная формула (см. [22.21]):

```
template<typename MATRIX> double wilkinsonShift(MATRIX const& A, int n)
{ // используется стабильная формула
    double d = (A(n - 2, n - 2) - A(n - 1, n - 1))/2,
        temp = A(n - 1, n - 2) * A(n - 1, n - 2);
    return A(n - 1, n - 1) - temp/(d + sign(d) * sqrt(d * d + temp));
}
```

Подробности цикла:

- ◆ используется тот же процесс ортогонального преобразования, что и в  $QR$  Хессенберга. Но преобразование сохраняется для вычисления  $RQ$ ;

- ◆ все записи выше диагонали становятся равны нулю, если  $|A[c+1, c+1]| \leq tol \times (|A[c, c]| + |A[c+1, c+1]|)$ . Значение *tol* предоставляется пользователем. Это пример относительной точности погрешности, основанной на ограничениях с плавающей точкой, — это также подтверждается теоремой Гершгорина о круге (обсуждается далее в этой главе);
- ◆ сокращение исключает любую правую нижнюю подматрицу *A*, где недиагональные элементы = 0. В коде сокращается значение *n*. Но (необязательно) алгоритм работает с полным *Q*, чтобы правильно его накапливать, когда нужны собственные векторы;
- ◆ для безопасности проверяется симметричность ввода с одинарной точностью.

```
template<typename MATRIX> Vector<double> QREigenTridiagonal(MATRIX A,
    Matrix<double>* Q = 0, int maxIter = 1000, double prec = highPrecEps)
{
    int n = A.getRows();
    assert(A.getColumns() == n);
    while(maxIter--)
    {
        int lastNZ = 0;
        for(int c = 0; c < n - 1; ++c) if(abs(A(c + 1, c)) <= prec *
            (abs(A(c, c)) + abs(A(c + 1, c + 1))))
        {
            A(c + 1, c) = 0;
            A(c, c + 1) = 0;
        }
        else lastNZ = c + 1;
        // сокращение
        n = lastNZ + 1;
        if(n < 2) break;
        // сдвиг
        double shift = wilkinsonShift(A, n);
        A -= shift * MATRIX::identity(A.getRows());
        // QR = A
        Vector<Vector<double>> reductions(n - 1);
        for(int c = 0; c < n - 1; ++c)
        {
            reductions[c] = HouseholderReduction2(A(c, c), A(c + 1, c));
            HouseholderLeftMult(A, reductions[c], c, c, min(n - 1, c + 2));
            if(Q) HouseholderLeftMult(*Q, reductions[c], c);
        } // A = RQ
        for(int c = 0; c < n - 1; ++c)
            HouseholderRightMult(A, reductions[c], c, max(0, c - 1), c + 1);
        // обращение сдвига
        A += shift * MATRIX::identity(A.getRows());
    }
    Vector<double> eig(A.getRows());
    for(int d = 0; d < A.getRows(); ++d) eig[d] = A(d, d);
    return eig;
}
```

```

bool isESymmetric(Matrix<double> const& A, double eRelAbs = defaultPrecEps)
{
    int n = A.getRows();
    if(n != A.getColumns()) return false;
    for(int r = 0; r < n; ++r)
    {
        if(!isfinite(A(r, r))) return false;
        for(int c = r + 1; c < n; ++c) if(!isfinite(A(r, c)) ||
            !isEEqual(A(r, c), A(c, r), eRelAbs)) return false;
    }
    return true;
}

pair<Vector<double>, Matrix<double>> QREigenSymmetric(Matrix<double> A,
    int maxIter = 1000, double prec = highPrecEps)
{ // строки Q являются собственными векторами
    assert(isESymmetric(A));
    pair<Matrix<double>, Matrix<double>> TQ = toHessenbergForm(A);
    Vector<double> eigve = QREigenTridiagonal(TQ.first, &TQ.second, maxIter, prec);
    return make_pair(eigve, TQ.second);
}

```

Сходимость является квадратичной по количеству итераций в худшем случае, но обычно кубической (см. [22.42]).

Обусловливание собственных значений и собственных векторов сложнее, чем у линейных систем. Подробности приведены в работах [22.12 и 22.47]. Основные выводы о возмущении  $A$  значением  $\Delta A$ :

- ◆ все собственные значения симметричных матриц идеально обусловлены;
- ◆ многие собственные значения несимметричных матриц могут быть плохо обусловлены;
- ◆ в обоих случаях собственные векторы могут быть очень плохо обусловлены в зависимости от того, насколько близки собственные значения.

В работе [22.43] авторы утверждают, что плохо обусловленные собственные пары бесполезны с научной точки зрения и представляют собой альтернативу.

## 22.18. Разложение по сингулярным значениям (SVD)

При заданной матрице  $A$  размером  $m \times n$ , без ограничения общности при  $m \geq n$ ,  $A = U\Sigma V$ , где (см. [22.51]):

- ◆ Матрица  $U$  ортогональная размером  $m \times m$ ;
- ◆  $\Sigma$  — матрица размером  $m \times n$  такая, что верхняя подматрица  $n \times n$  равна неотрицательной диагонали, а нижняя подматрица  $(m - n) \times n$  равна 0. Первая — это *сингулярные значения*  $A$ ;
- ◆  $V$  — ортогональная матрица  $n \times n$ .

В некоторых источниках  $U$  и  $V$  транспонированы в соответствии с результатами алгоритма. Если  $n > m$ , вычислите разложение по сингулярным значениям (SVD, Singular

Value Decomposition) для  $A^T$ , чтобы получить  $A = U^T \Sigma V^T$ . Можно считать, SVD — это разбиение  $A$  на сумму внешних произведений, т. е.  $A = \sum \sigma_i u_i \otimes v_i$ . Например, эта интерпретация позволяет использовать SVD для сжатия, где задействовано только некоторое количество самых больших  $\sigma_i$ .

Простой алгоритм, полученный путем вычисления собственных значений  $A^T A$  с использованием реального симметричного алгоритма, приводит к  $\Sigma^2 = Q(A^T A)Q^T$ . Формирование  $A^T A$  явно приводит к численной нестабильности из-за потенциального чрезмерного накопления округлений. Алгоритм Голуба — Кахана неявно поддерживает  $A^T A$ :

1. Преобразование  $A$  в двудиagonalную форму, где  $A = I +$  верхняя поддиагональ.
2. Неявно выполняется смещенная итерация  $QR$  на  $A^T A$  до слияния.

Для удобства все функции и результат помещены в класс:

```
struct SVD
{
    Vector<double> svds;
    Matrix<double> U, V;
    SVD(Matrix<double> A, int maxIter = 100, double prec = highPrecEps):
        svds(A.getColumns()), U(Matrix<double>::identity(A.getRows())),
        V(Matrix<double>::identity(A.getColumns()))
    { // обмен U и V, если нужно
        bool needSwap = A.getColumns() > A.getRows();
        if(needSwap)
        {
            svds = Vector<double>(A.getRows());
            A = A.transpose();
            swap(U, V);
        }
        toBidiagonalForm(A); // преобразование в двудиagonalную форму
        int n = A.getColumns();
        // работа с квадратной подматрицей, если нужно
        if(n < A.getRows()) A = submatrix<double>(A, 0, n - 1, 0, n - 1);
        SVDBidiagonal(A, maxIter, prec);
        if(needSwap) // обратный обмен, если нужно
        {
            Matrix<double> temp = V.transpose();
            V = U.transpose();
            U = temp;
        }
    }
};
```

Для объяснения двудиagonalизации  $O(n^3)$  обратитесь к работе [22.21]:

```
void toBidiagonalForm(Matrix<double>& A)
{
    int m = A.getRows(), n = A.getColumns();
    assert(m >= n);
    for(int c = 0; c < n; ++c)
    {
        Vector<double> x(m - c);
```

```

    for(int j = c; j < m; ++j) x[j - c] = A(j, c);
    x = HouseholderReduction(x);
    HouseholderLeftMult(A, x, c, c);
    HouseholderLeftMult(U, x, c);
    if(c < n - 2)
    {
        x = Vector<double>(n - (c + 1));
        for(int j = c + 1; j < n; ++j) x[j - (c + 1)] = A(c, j);
        x = HouseholderReduction(x);
        HouseholderRightMult(A, x, c + 1, c);
        HouseholderRightMult(V, x, c + 1);
    }
}
}

```

Симметричная итерация  $QR$  нуждается в некоторых изменениях:

- ◆ сдвиг вычисляется из  $A^T A$ , поэтому необходимо сформировать его правую нижнюю подматрицу  $2 \times 2$ , что не приводит к неустойчивости (формула приведена в работе [22.21]);
- ◆ в имеющемся виде алгоритм может возвращать отрицательные сингулярные значения (будьте осторожны — это не упоминается в работе [22.21]), но их можно переместить в  $V$  (см. [22.37]):
  - $\Sigma V = \Sigma I V$ , где  $I$  — тождество, но при  $I(i, i) = -1$ ;
  - Чтобы найти  $I V$ , выполняем умножение  $V(i, *)$  на  $-1$ ;
- ◆ применение ортогональных преобразований к  $A$  определенным образом дает как  $U$ , так и  $V$  в готовом виде (объясняется в работе [22.21]).

```

void SVDBidiagonal(Matrix<double>& A, int maxIter = 100,
    double prec = highPrecEps)
{
    assert(A.getColumns() == A.getRows());
    int n = A.getRows();
    while(maxIter--)
    {
        int lastNZ = 0;
        for(int c = 0; c < n - 1; ++c) if(isEEqual(A(c, c), 0) ||
            abs(A(c, c + 1)) <= prec *
                (abs(A(c, c)) + abs(A(c + 1, c + 1)))) A(c, c + 1) = 0;
        else lastNZ = c + 1;
        // сокращение
        n = lastNZ + 1;
        if(n < 2) break;
        Matrix<double> T22 =
            submatrix<double>(A, n - 2, n - 1, n - 2, n - 1);
        double dm = T22(0, 0), dn = T22(1, 1), fm = T22(0, 1),
            frml = (n >= 3 ? A(n - 3, n - 2) : 0);
        T22(0, 0) = dm * dm + frml * frml;
        T22(0, 1) = dm * fm;
        T22(1, 0) = dm * fm;
        T22(1, 1) = dn * dn + fm * fm;
    }
}

```

```

double shift = wilkinsonShift(T22, 2),
    t00 = A(0, 0) * A(0, 0), t01 = A(0, 0) * A(0, 1),
    y = t00 - shift, z = t01;
for(int c = 0; c < n - 1; ++c)
{
    Vector<double> x = HouseholderReduction2(y, z);
    HouseholderRightMult(A, x, c);
    HouseholderRightMult(V, x, c);
    y = A(c, c);
    z = A(c + 1, c);
    x = HouseholderReduction2(y, z);
    HouseholderLeftMult(A, x, c);
    HouseholderLeftMult(U, x, c);
    if(c < n - 2)
    {
        y = A(c, c + 1);
        z = A(c, c + 2);
    }
}
}
n = A.getRows();
for(int d = 0; d < n; ++d)
{ // обеспечивается неотрицательность SVD путем перемещения
  // отрицательных значений в V
  if(A(d, d) < 0)
  {
      A(d, d) *= -1;
      for(int r = 0; r < n; ++r) V(r, d) *= -1;
  }
  svds[d] = A(d, d);
}
}

```

Поскольку вычисление SVD эквивалентно вычислению симметричного собственного значения  $A^T A$ , оно просто стабильнее реализовано и сходится с той же скоростью. В работах [22.21, 22.38 и 22.40] приведен анализ условий/возмущений SVD. В последней, в частности, доказаны многие теоремы, которые в других источниках лишь формулируются.

У SVD множество применений. В частности (см. [22.42]):

- ◆  $\text{rang}(A)$  = количество сингулярных значений  $> 0$ . Используется некоторое абсолютное значение  $\varepsilon$ ;
- ◆  $\|A\|_2$  = максимальное сингулярное значение;
- ◆  $\|A^{-1}\|_2$  = минимальное сингулярное значение;
- ◆  $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$ .

```

int rank(double precFor0 = highPrecEps) const
{
    int non0 = 0;
    for(int i = 0; i < svds.getSize(); ++i) non0 += (svds[i] >= precFor0);
}

```



```

return non0;
}
double norm2()const{return valMax(svds.getArray(), svds.getSize());}
double norm2Inv()const{return 1/valMin(svds.getArray(), svds.getSize());}
double condition2()const{return norm2() * norm2Inv();}

```

## 22.19. Собственные значения и собственные векторы асимметричной матрицы

Итерация со смещенным  $QR$  работает и для асимметричных матриц, но с модификациями и без гарантированной сходимости. Модификации:

- ♦ цикл сходится к квазитреугольной матрице с блоками  $2 \times 2$  на диагонали, которые представляют собой комплексно-сопряженные собственные значения. Они решаются аналитически с использованием формулы следа определителя, как и для сдвига Уилкинсона;
- ♦ ни один реальный сдвиг не может заставить комплексный блок сократиться, поэтому — используя *алгоритм неявного сдвига* — выполняются два последовательных реальных сдвига. В работах [22.21 и 22.18] приведена более подробная информация. В работе [22.48] автор объясняет это по-другому, используя сокращение. Ключевой трюк заключается в том, что двойной сдвиг эквивалентен двум комплексно-сопряженным сдвигам  $a_1$  и  $a_2$ . В частности, используйте нижний правый угол  $2 \times 2$   $H$  для их вычисления в надежде, что у вас получатся хорошие оценки собственных значений, которые являются идеальными сдвигами для извлечения последней строки/столбца. В вычислении используются трассировка  $= a_1 + a_2$  и определитель  $= a_1 a_2$ , и оба являются вещественными;
- ♦  $Q$ , если накапливается, не дает собственные векторы напрямую. Можно найти их по  $Q$  (и в некоторых случаях это более эффективно — см. [22.21]), но гораздо проще использовать обратную итерацию по  $H$  и его  $Q$  с вычисленными собственными значениями (обсуждается позже).

```

pair<Vector<complex<double> >, Matrix<double> > QREigen(Matrix<double> A,
    int maxIter = 1000, double prec = highPrecEps)
{
    pair<Matrix<double>, Matrix<double> > TQ = toHessenbergForm(A);
    Vector<complex<double> > eigs = QREigenHessenberg(TQ.first, maxIter, prec);
    return make_pair(eigs, findEigenvectors(TQ.first, TQ.second, eigs));
}

```

Известны контрпримеры, в которых сходимость не достигается (см. [22.18]) — например, матрицы перестановок, простейшая из которых:

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

с собственными значениями  $\{1, -0,5 \pm \sqrt{3}i/2\}$ , все с амплитудой  $1$  (корни  $x^3 - 1$ ). Реализация в этом примере фактически сходится за 38 итераций, потому что незначительные ошибки округления в ортогональных преобразованиях нарушают симметрию.

Теорема (см. [22.49 и 22.47]): если все собственные значения различны, итерация по  $QR$  использует сдвиги Рэлея, и она сходится, то скорость является квадратичной. Эта граница полезна лишь с моральной точки зрения, т. к. используется сдвиг Уилкинсона, и сходимость не гарантируется. Экспериментальное наблюдение состоит в том, что для каждого собственного значения выполняются 1–2 итерации (см. [22.21]). Стабильность доказана в работе [22.2], где также описываются недавно улучшенные алгоритмы для многих особых случаев. В работе [22.12] приведены отличные численные примеры (с небольшими алгоритмическими модификациями) для этого и многих других представленных здесь алгоритмов.

Существует эвристическая стратегия исправления — *исключительные сдвиги*, когда после 10 итераций без сокращения для нарушения симметрии используется специальный сдвиг (см. [22.14]). В этой статье рассматривается несколько детерминированных стратегий сдвига, но ни одна из них не сходится во всех случаях, поэтому мы используем случайные сдвиги.

*Теорема круга Гершгорина* (см. [22.46.]): матрица  $H$  имеет собственное значение с радиусом в комплексном круге с центром  $A[n-1, n-1]$  и радиусом  $|A[n-1, n-2]|$ . Таким образом, можно случайным образом выбрать собственное значение  $a + ib$  с этого круга и вычислить  $\text{trace} = 2a$  и  $\text{determinant} = a^2 + b^2$ . Это сокращает в моих тестах количество итераций на приведенной ранее матрице до 15–17:

```
pair<double, double> traceDet2x2(Matrix<double> const& A, int d)
{
    double trace = (A(d, d) + A(d + 1, d + 1)), det =
        A(d, d) * A(d + 1, d + 1) - A(d, d + 1) * A(d + 1, d);
    return make_pair(trace, det);
}

Vector<complex<double> > QREigenHessenberg(Matrix<double> A,
    int maxIter = 10000, double prec = highPrecEps)
{
    assert(A.getColumns() == A.getRows());
    int n = A.getRows(), lastDeflateIter = maxIter;
    while(maxIter--)
    { // обработка сходящихся значений
        for(int c = 0; c < n - 1; ++c) if(abs(A(c + 1, c)) <= prec *
            (abs(A(c, c)) + abs(A(c + 1, c + 1)))) A(c + 1, c) = 0;
        // сокращение
        while(n >= 3)
        {
            if(A(n - 1, n - 2) == 0) --n;
            else if(A(n - 2, n - 3) == 0) n -= 2;
            else break;
            lastDeflateIter = maxIter;
        }
        if(n < 3) break;
        pair<double, double> td = traceDet2x2(A, n - 2);
        if(lastDeflateIter - maxIter > 10)
        { // маловероятный исключительный сдвиг
            pair<double, double> ab = GlobalRNG().pointInUnitCircle();
            double r = abs(A(n - 1, n - 2));
            ab.first = A(n - 1, n - 1) + ab.first * r;
```

```

        ab.second *= r;
        td.first = 2 * ab.first;
        td.second = ab.first * ab.first + ab.second * ab.second;
        lastDeflateIter = maxIter;
    } // вычисление вектора Хаусхолдера
    Vector<double> xyz(3);
    xyz[0] = A(0, 0) * A(0, 0) + A(0, 1) * A(1, 0) - td.first * A(0, 0) +
        td.second;
    xyz[1] = A(1, 0) * (A(0, 0) + A(1, 1) - td.first);
    xyz[2] = A(1, 0) * A(2, 1);
    for(int c = 0; c < n - 1; ++c)
    { // ее обработка
        xyz = HouseholderReduction(xyz);
        HouseholderLeftMult(A, xyz, c, max(0, c - 1), n - 1);
        HouseholderRightMult(A, xyz, c, 0, min(c + 3, n - 1));
        if(c + 2 == n) break;
        xyz[0] = A(c + 1, c); // обновление для будущей итерации
        xyz[1] = A(c + 2, c);
        if(c + 3 < n) xyz[2] = A(c + 3, c);
        else xyz.removeLast();
    }
} // извлечение собственных значений из блоков
n = A.getRows();
double NaN = numeric_limits<double>::quiet_NaN();
Vector<complex<double>> eigs(n, complex<double>(NaN, NaN));
for(int c = 0; c < n; ++c)
    if(c == 0 || A(c, c - 1) == 0)
    { // поиск блока
        if(c + 1 >= n || A(c + 1, c) == 0) // 1 x 1
            eigs[c] = complex<double>(A(c, c), 0);
        else if(c + 2 >= n || A(c + 2, c + 1) == 0) // 2 x 2
        {
            pair<double, double> td = traceDet2x2(A, c);
            td.first /= 2;
            double temp = td.first * td.first - td.second,
                temp2 = sqrt(abs(temp));
            if(temp > 0)
            { // устойчивая формула для e = trace05 +- temp2
                double eig1 = td.first + sign(td.first) * temp2;
                eigs[c] = complex<double>(eig1, 0);
                eigs[c + 1] = complex<double>(td.second/eig1, 0);
            }
            else
            {
                eigs[c] = complex<double>(td.first, td.second);
                eigs[c + 1] = complex<double>(td.first, -td.second);
            }
            ++c;
        }
        else c += 3; // схождение не достигнуто, отсутствует изоляция
    }
return eigs;
}

```

Случайные сдвиги должны сходиться, потому что в конечном итоге генерируется точное собственное значение, но это не было доказано (и возникает вопрос со скоростью сходимости).

*Обратная итерация* (см. [22.21]) позволяет вычислять точные собственные векторы от точных приближений к собственным значениям. Для заданной матрицы  $A$  с собственным значением  $\lambda$  соответствующий собственный вектор  $v$  определяется как  $(A - \lambda I)v = 0$ . Обратная итерация предполагает приблизительные собственные значения  $a \approx \lambda$ . Тогда получим  $(A - \lambda I)v = (\lambda - a)v$  и  $x = (A - \lambda I)^{-1}v$ , где  $x$  — это константа  $\times v$ . Поскольку собственные векторы нормализованы, получаем итерацию с фиксированной точкой (обсуждается в главе 23. *Численные алгоритмы: работа с функциями*), где начальное значение  $v$  предоставляется пользователем, а затем становится равным  $v = \frac{x}{\|x\|}$ .

Некоторые детали реализации (см. [22.24]):

- ◆ случайно выбранный начальный вектор приводит к хорошему решению за одну итерацию с высокой вероятностью, а среди детерминированных вариантов использование единичного вектора безопасно, хотя иногда и то и другое не срабатывает;
- ◆ у повторяющегося собственного значения собственный вектор притяжения зависит от начального вектора. Поэтому при инициализации используется случайный начальный вектор, чтобы найти разные собственные векторы для повторяющихся собственных значений;
- ◆ 5 — хороший предел количества итераций.

Когда матрица  $A$  находится в форме Хессенберга  $Q^T H Q$ , итерация становится более эффективной, потому что вместо этого мы работаем с  $H$  для вычисления  $v(H, a)$ , используя  $QR$  Хессенберга, чтобы решить уравнение за время  $O(n^2)$ . Тогда  $v(A, a) = Q^T v(H, a)$ :

1. Выбрать произвольный ненулевой стартовый собственный вектор  $v$  — в нашем случае случайный.
2.  $v \neq \|v\|$ .
3. Пока  $\|Hv\|_\infty \leq \epsilon \|H\|_\infty$ :
4. Решить  $(H - aI)x = v$ .
5.  $v = \frac{x}{\|x\|}$ .
6.  $Q^T v$  — это собственный вектор  $A$ , соответствующий  $a$ .

В пункте (4) матрица  $H - \lambda I$  может быть строго сингулярной — в  $x$  оказалась  $\infty$  (из-за ошибки округления/0) или NaN (при 0/0) компонентов в случае  $\lambda$  с кратностью  $> 1$  (см. [22.18]). Этот случай обрабатывается путем установки значения 0. Хотя это недопустимо для решения уравнений, стратегия работает для обратной итерации.

Математический процесс четко определен, даже если  $a = \lambda$ , но не существует обратного понятия, однако вы можете выбрать решение уравнения как  $v$ . Когда  $a \neq \lambda$ , решение будет слегка подтягиваться к другим собственным векторам, и тогда решение  $\rightarrow 0$  при  $a \rightarrow \lambda$ .

Обратная итерация позволяет найти сложные собственные векторы из сложных собственных значений (см. [22.18]), но для простоты реализация возвращает собственные векторы NaN для сложных собственных значений:

```
Vector<double> findEigenvector(Matrix<double> H, double eig,
    int maxIter = 5, double prec = highPrecEps)
{ // обратная итерация
    int n = H.getRows();
    double HInf = normInf(H);
    H -= eig * Matrix<double>::identity(n);
    QRDecomposition qr(H, true);
    Vector<double> x = GlobalRNG().randomUnitVector(n);
    while(maxIter--)
    {
        x = qr.solve(x);
        for(int i = 0; i < n; ++i) if(!isfinite(x[i])) x[i] = 0;
        x *= 1/norm(x);
        if(normInf(H * x) <= prec * HInf) break;
    }
    return x;
}

Matrix<double> findEigenvectors(Matrix<double> const& H,
    Matrix<double> const& Q, Vector<complex<double> > const& eigs)
{
    int n = eigs.getSize();
    Matrix<double> result(n, n);
    for(int i = 0; i < n; ++i)
    {
        if(eigs[i].imag() == 0)
        {
            Vector<double> eigve = findEigenvector(H, eigs[i].real());
            for(int r = 0; r < n; ++r) result(r, i) = eigve[r];
        }
        else for(int r = 0; r < n; ++r)
            result(r, i) = numeric_limits<double>::quiet_NaN();
    }
    return Q.transpose() * result;
}
```

Некоторые результаты сходимости (см. [22.24]):

- ◆ у нормальных (симметричных) матриц невязка  $r = (A - aI)v$  сходится к наилучшему достижимому значению, а итерации принадлежат собственному пространству, связанному с ближайшим собственным значением;
- ◆ у ненормальных диагонализируемых матриц ситуация совершенно иная — невязки обычно увеличиваются после первых итераций, а итерация до сходимости или уменьшения плохого эффекта исходного стартового вектора имеет смысл только в том случае, если матрица собственных векторов  $V$  хорошо обусловлена. Итерации приближаются к подпространству ближайшего собственного вектора даже при увеличении невязок, но не имеют особых гарантий сходимости;

- ◆ при конечной точности плохая обусловленность из-за почти сингулярности  $(A - aI)$  не вызывает проблем — любая плохая обусловленность в вычислении будет вызвана проблемой, а не алгоритмом (см. [22.42]).

Но не ожидайте, что обратная итерация будет успешной во всех случаях — в отличие от симметричной матрицы, несимметричная может быть дефектной — т. е. не иметь собственного разложения (см. [22.42]). Также возможен случай, когда у  $A$  есть повторяющееся собственное значение, но различные собственные векторы — например, при  $A = a$ , кратном  $I$ . Как уже упоминалось, этот случай обрабатывается случайным начальным вектором с высокой вероятностью, пока генерация не приводит к повторению (или около-повторению).

## 22.20. Разреженные матрицы

Матрица  $A$  является разреженной, когда в ней достаточно нулей, чтобы использовать  $O(n)$  ненулевых элементов. Это число иногда называется *числом ненулевых элементов*, или nnz (number of nonzeros). Существуют способы представить разреженную матрицу в виде коллекции вида:

1. *Записи* — список записей. Коллекции просты с точки зрения хранения, но не поддерживают эффективные операции.
2. *Записи* — хеш-таблица. Коллекции обеспечивают эффективное представление, когда  $\text{nnz} = O(1)$ , и поддерживают некоторые операции — такие как произвольный доступ, умножение разреженной матрицы на плотный вектор (с использованием итерации в случайном порядке), но не поддерживают итерацию по столбцам или строкам. Это представление используется как промежуточное для произведения разреженных матриц.
3. *Векторы-столбцы* — массив отсортированных разреженных векторов. Представление, которое мы используем здесь, обеспечивает эффективную итерацию по столбцу, произвольный доступ для чтения с применением двоичного поиска и не слишком затратную вставку элементов.
4. *Векторы-столбцы* — сжатое хранилище столбцов. То же, что и в пункте (3), но все представлено компактно тремя массивами, как сжатый граф (см. главу 11. *Алгоритмы графов*). По сравнению с (3) подход позволяет достичь лучшей экономии памяти в постоянном множителе, но является более громоздким для программирования, а вставка элементов намного дороже. Но если  $A$  содержит  $o(n)$  элементов, это представление оказывается намного эффективнее с точки зрения памяти, потому что в пункте (3) требуется  $O(n)$  места для хранения столбцов.
5. *Векторы-столбцы* — список несортированных разреженных векторов. Аналогичны (3), но эффективнее для некоторых операций (см. [22.16]).
6. *Векторы-строки* — любой эквивалент (3–5). По сути, представление обладает теми же свойствами, но умножение разреженной матрицы на плотный вектор, используемое в итерационных методах, выполняется эффективнее при переборе строк по порядку. Однако другие методы для этой операции по-прежнему эффективны, и, вероятно, из-за традиционного рассмотрения векторов как векторов-строк этот метод менее популярен.

С пунктами (3–6) связан важный вопрос о том, как хранить векторы. Есть простой вариант — использовать вектор пар. Другой вариант — использовать вектор значений и вектор индексов. Второй более эффективен с точки зрения памяти, но с первым проще работать, и он более эффективен с точки зрения кеша.

Типичный шаблон алгоритма разреженных матриц состоит в том, чтобы преобразовать основное представление в более удобное, выполнить некоторую работу и преобразовать обратно. Одним из самых простых преобразований является транспонирование (3–6), которое вместе с исходным представлением дает упорядоченную или несортированную итерацию как по строке, так и по столбцу. Для некоторых операций его можно выполнять неявно с целью экономии памяти.

Еще один шаблон — разрешить наличие нулевых элементов, которые задаются явно (хотя простые проверки могут это обойти).

В специализированных алгоритмах для преобразования в 0 может использоваться абсолютная принадлежность, но это делается редко.

Большая часть реализации пункта (3) проста, но в операциях сложения разреженных векторов и скалярного произведения следует проявлять осторожность, чтобы они выполнялись в правильной последовательности (это хорошие вопросы для собеседования). Вставка в худшем случае выполняется за  $O(n)$ , но амортизируется как  $O(nnz/n)$ :

```
template<typename ITEM = double>
class SparseMatrix: public ArithmeticType<SparseMatrix<ITEM> >
{
    int rows;
    typedef pair<int, ITEM> Item;
    typedef Vector<Item> SparseVector;
    Vector<SparseVector> itemColumns;
    int findPosition(int r, int c) const
    { // возвращается индекс этого элемента или число -1, если
      // такого индекса нет
      assert(0 <= r && r < rows && 0 <= c && c < getColumns());
      SparseVector const& column = itemColumns[c];
      return binarySearch(column.getArray(), 0, column.getSize() - 1,
                          Item(r, 0), PairFirstComparator<int, ITEM>());
    }
public:
    SparseMatrix(int theRows, int theColumns): rows(theRows),
        itemColumns(theColumns){}
    int getRows() const {return rows;}
    int getColumns() const {return itemColumns.getSize();}
    ITEM operator()(int r, int c) const
    { // отсутствующие записи равны нулю
      int position = findPosition(r, c);
      return position == -1 ? 0 : itemColumns[c][position].second;
    }
    void set(int r, int c, ITEM const& item)
    { // сначала выполняется попытка найти и обновить значения
      SparseVector& column = itemColumns[c];
      int position = findPosition(r, c);
      if(position != -1) column[position].second = item;
```

```

    else if(item != 0) // если не удастся, выполняется вставка вектора
                        // с помощью сдвига
    { // вниз по столбцу
        Item temp(r, item);
        column.append(temp);
        position = column.getSize() - 1;
        for(; position > 0 && column[position - 1].first > r; --position)
            column[position] = column[position - 1];
        column[position] = temp;
    }
}

static SparseVector addSparseVectors(SparseVector const& a,
    SparseVector const& b)
{ // элементы берутся из обоих векторов и складываются,
  // если оба существуют
  SparseVector result;
  for(int aj = 0, bj = 0; aj < a.getSize() || bj < b.getSize(); )
  {
      bool considerBoth = aj < a.getSize() && bj < b.getSize();
      int j = aj < a.getSize() ? a[aj].first : b[bj].first;
      ITEM item = aj < a.getSize() ? a[aj++].second : b[bj++].second;
      if(considerBoth) // сначала берется значение a,
                      // затем рассматривается b
          if(j == b[bj].first) item += b[bj++].second;
          else if(j > b[bj].first)
          {
              j = b[bj].first;
              --aj; // отмена инкремента aj
              item = b[bj++].second;
          }
      if(item != 0) result.append(Item(j, item)); // на всякий пожарный
  }
  return result;
}

SparseMatrix& operator+=(SparseMatrix const& rhs)
{ // сложение по столбцам
    assert(rows == rhs.rows && getColumns() == rhs.getColumns());
    SparseMatrix result(rows, getColumns());
    for(int c = 0; c < getColumns(); ++c) result.itemColumns[c] =
        addSparseVectors(itemColumns[c], rhs.itemColumns[c]);
    return *this = result;
}

SparseMatrix& operator*=(ITEM a)
{
    for(int c = 0; c < getColumns(); ++c)
        for(int j = 0; j < itemColumns[c].getSize(); ++j)
            itemColumns[c][j].second *= a;
    return *this;
}

friend SparseMatrix operator*(SparseMatrix const& A, ITEM a)
{
    SparseMatrix result(A);

```



```

        return result *= a;
    }
    SparseMatrix operator-( ) const
    {
        SparseMatrix result(*this);
        return result *= -1;
    }
    SparseMatrix& operator+=(SparseMatrix const& rhs)
        {return *this += -rhs;}
    static SparseMatrix identity(int n)
    {
        SparseMatrix result(n, n);
        for(int c = 0; c < n; ++c) result.itemColumns[c].append(Item(c, 1));
        return result;
    }
    static ITEM dotSparseVectors(SparseVector const& a,
        SparseVector const& b)
    { // прибавление к сумме, если оба элемента существуют
        ITEM result = 0;
        for(int aj = 0, bj = 0; aj < a.getSize() && bj < b.getSize(); )
            if(a[aj].first == b[bj].first)
                result += a[aj++].second * b[bj++].second;
            else if(a[aj].first < b[bj].first) ++aj;
            else ++bj;
        return result;
    }
    static Vector<ITEM> sparseToDense(SparseVector const& sv, int n)
    { // требуется число n, поскольку разреженный хвост неизвестен
        assert(sv.getSize() == 0 || sv[sv.getSize() - 1].first < n);
        Vector<ITEM> v(n);
        for(int i = 0; i < sv.getSize(); ++i) v[sv[i].first] = sv[i].second;
        return v;
    }
    static SparseVector denseToSparse(Vector<ITEM> const& v)
    {
        SparseVector sv;
        for(int i = 0; i < v.getSize(); ++i)
            if(v[i] != 0) sv.append(Item(i, v[i]));
        return sv;
    }
    friend SparseVector operator*(SparseVector const& v, SparseMatrix const& A)
    {
        assert(v.getSize() == 0 || v.lastItem().first < A.getRows());
        SparseVector result;
        for(int c = 0; c < A.getColumns(); ++c)
        { // сложение по одной строке за раз
            ITEM rc = dotSparseVectors(v, A.itemColumns[c]);
            if(rc != 0) result.append(Item(c, rc));
        }
        return result;
    }
}

```

```

friend SparseVector operator*(SparseMatrix const& A, SparseVector const& v)
{return v * A.transpose();}
friend Vector<ITEM> operator*(Vector<ITEM> const& v, SparseMatrix const& A)
{return sparseToDense(denseToSparse(v) * A, A.rows);}
friend Vector<ITEM> operator*(SparseMatrix const& b, Vector<ITEM> const& v)
{return sparseToDense(b * denseToSparse(v), b.getColumns());}
friend double normInf(SparseMatrix const& A)
{ // для оптимизации цикла сначала вычисляется разреженная матрица
  SparseMatrix AT = A.transpose();
  double maxRowSum = 0;
  for(int r = 0; r < A.getRows(); ++r)
  {
    double rSum = 0;
    for(int cj = 0; cj < AT.itemColumns[r].getSize(); ++cj)
      rSum += abs(AT.itemColumns[r][cj].second);
    maxRowSum = max(maxRowSum, rSum);
  }
  return maxRowSum;
}
};

```

Чтобы вычислить транспонированную матрицу, первый столбец исходной матрицы становится первой строкой в новой матрице. Поскольку столбцы обрабатываются по порядку, записи транспонирования также создаются по порядку:

```

SparseMatrix transpose()const
{
  SparseMatrix result(getColumns(), rows);
  for(int c = 0; c < getColumns(); ++c)
    for(int j = 0; j < itemColumns[c].getSize(); ++j)
      result.itemColumns[itemColumns[c][j].first].append(
        Item(c, itemColumns[c][j].second));
  return result;
}

```

В некоторых операциях матрично-векторного произведения транспонирование используется явно, но может делаться и неявно, если транспонированный результат сам по себе не нужен. Мы для простоты опустим это:

```

friend SparseVector operator*(SparseVector const& v, SparseMatrix const& A)
{
  assert(v.getSize() == 0 || v.lastItem().first < A.getRows());
  SparseVector result;
  for(int c = 0; c < A.getColumns(); ++c)
  { // добавляем по одной строке по очереди
    ITEM rc = dotSparseVectors(v, A.itemColumns[c]);
    if(rc != 0) result.append(Item(c, rc));
  }
  return result;
}

friend SparseVector operator*(SparseMatrix const& A, SparseVector const& v)
{return v * A.transpose();}

```

```
friend Vector<ITEM> operator*(Vector<ITEM> const& v, SparseMatrix const& A)
{return sparseToDense(denseToSparse(v) * A, A.rows);}
friend Vector<ITEM> operator*(SparseMatrix const& b, Vector<ITEM> const& v)
{return sparseToDense(b * denseToSparse(v), b.getColumns());}
```

При выполнении умножения матриц обычное скалярное произведение по столбцам требует  $O(n^2)$  скалярных произведений, большинство из которых окажется равным 0. Одна из возможных организаций плотного тройного цикла умножения соответствует сумме внешних произведений:  $AB = \sum_k A[* , k] \otimes B[k, *]$ . Внешние произведения обычно разреженные, потому что вектор  $O(1) \times$  вектор  $O(1) =$  матрица  $O(1)$ . Таким образом, при достаточной разреженности время становится равно  $O(n)$ . Представление вектора-столбца для матриц внешнего произведения  $O(1)$  неэффективно, поэтому лучше использовать представление хеш-таблицы:

1. Транспонирование  $B$  для эффективного доступа к строке.
2. Накопление матриц внешних произведений в хеш-таблицу.
3. Перебор хеш-таблицы, чтобы преобразовать ее в основное представление.
4. Сортировка всех столбцов.

```
SparseMatrix& operator*=(SparseMatrix const& rhs)
{ //  $O(n^2)$  * коэффициент разреженности (от 1 до n)
  assert(getColumns() == rhs.rows);
  SparseMatrix result(rows, rhs.getColumns()), bT = rhs.transpose();
  // вычисление суммы сумм внешних произведений
  typedef typename Key2DBuilder<>::WORD_TYPE W;
  LinearProbingHashTable<W, ITEM> outerSums;
  Key2DBuilder<> kb(max(result.rows, result.getColumns()),
    result.rows >= result.getColumns());
  for(int k = 0; k < rhs.rows; ++k)
    for(int aj = 0; aj < itemColumns[k].getSize(); ++aj)
      for(int btj = 0; btj < bT.itemColumns[k].getSize(); ++btj)
      {
        int r = itemColumns[k][aj].first,
            c = bT.itemColumns[k][btj].first;
        W key = kb.to1D(r, c);
        ITEM* rcSum = outerSums.find(key), rcValue =
            itemColumns[k][aj].second *
            bT.itemColumns[k][btj].second;
        if(rcSum) *rcSum = rcValue;
        else outerSums.insert(key, rcValue);
      }
  // преобразование внешней хеш-таблицы суммы
  // в конечную структуру данных
  for(typename LinearProbingHashTable<W, ITEM>::Iterator iter =
    outerSums.begin(); iter != outerSums.end(); ++iter)
  { // n должно быть больше r и c, чтобы вычисление имело смысл
    pair<unsigned int, unsigned int> rc = kb.to2D(iter->key);
    result.itemColumns[rc.second].append(Item(rc.first, iter->value));
  } // сортируем каждый столбец, чтобы зафиксировать порядок
  for(int c = 0; c < result.getColumns(); ++c) quickSort(
    result.itemColumns[c].getArray(), 0,
```

```

        result.itemColumns[c].getSize() - 1,
        PairFirstComparator<int, ITEM>());
    return *this = result;
}

```

## 22.21. Итерационные методы для разреженных матриц

Итерационные методы работают как есть и с плотными матрицами, но имеют смысл только для разреженных матриц. Наиболее полезные из них выполняются в *подпространстве Крылова* — т. е. для матрицы  $A$  набор векторов имеет вид  $Ax$ ,  $A(Ax)$  и т. д. Все общие алгоритмы плотных матриц — такие как умножение, выполняются за  $n$  итераций с  $O(n^2)$  работы каждая. Итерационные методы в лучшем случае требуют:

- ♦  $< n$  итераций для достаточной точности;
- ♦  $(n)$  работы за итерацию, если матрица  $A$  очень разреженная.

Таким образом, общая работа может даже оказаться равной  $O(n)$ .

Предпочтительным итерационным методом решения уравнений с *симметричными и положительно определенными* матрицами (Symmetric and Positive Definite, SPSPD) является *алгоритм сопряженных градиентов* (Conjugate Gradient, CG) — см. [22.18]. От него происходит одноименный метод нелинейной оптимизации (см. главу 24. Численная оптимизация). В точной арифметике он завершается после  $n$  итераций из-за их определенной ортогональности, но из-за ограничений точности большая часть ортогональности может быть потеряна. Поэтому используйте  $n$  в качестве ограничения на количество итераций, но считайте сходимость при достижении определенной 2-нормовой точности.

Уравнение  $Ax = b$  подразумевает, что  $A^T Ax = A^T b$ . Поскольку матрица  $A^T A$  является SPSPD, сопряженный градиент применяется как к общим уравнениям, так и к задачам наименьших квадратов. Эта модификация называется CGNR (где NR означает normal equations, нормальное уравнение). Скорость сходимости для случая SPSPD равна

$O\left(1 + \frac{2}{\sqrt{\kappa(A)}}\right)$ , где  $\kappa$  — это число обусловленности (см. [22.42]). Для не-SPSPD матриц

не используется корень  $\sqrt{\cdot}$ , потому что фактически получается квадрат  $A$ .

*Предобусловливателем* называется любая матрица  $P$ , «подобная»  $A$  и такая, что в идеале  $P^{-1}A \approx I$ . Часто выбирают значение  $P = I$ , но другой, более удачный выбор может сильно сократить количество итераций при решении  $P^{-1}Ax = P^{-1}b$  из-за лучшей обусловленности  $P^{-1}A$  (см. [22.18]). Дешевая с точки зрения инвертирования матрица  $P$  является *матрицей Якоби* =  $\text{diag}(A)$ . Это почти всегда лучше, чем не использовать предварительные условия, и к тому же улучшается масштаб (см. [22.46]). В случае CGNR матрица  $P$  вычисляется на основе  $A^T A$ , и, поскольку  $P$  изначально является приближенной, явное формирование произведения не создает проблем:

```

template<typename MATRIX> MATRIX findJacobiPreconditioner(MATRIX A,
    bool isSPSPD = true)
{
    if(!isSPSPD) A = A.transpose() * A;
}

```

```

    int n = A.getRows();
    MATRIX diagInv(n, n);
    for(int r = 0; r < n; ++r) diagInv.set(r, r, 1/A(r, r));
    return diagInv;
}

template<typename MATRIX, typename PRECONDITIONER> pair<Vector<double>, double>
conjugateGradientSolve(MATRIX const& A, Vector<double> b,
    PRECONDITIONER const& pInv, bool isSPSD = true,
    Vector<double> x = Vector<double>(), double eFactor = highPrecEps)
{
    MATRIX AT(1, 1); // значение на случай СПО
    int n = A.getRows(), maxIter = n;
    if(x.getSize() == 0) x = Vector<double>(n, 0);
    assert(x.getSize() == n && b.getSize() == A.getColumns());
    if(!isSPSD)
    {
        AT = A.transpose();
        b = AT * b;
    }
    Vector<double> temp = A * x, r = b - (isSPSD ? temp : AT * temp),
        z = pInv * r, p = z;
    while(maxIter-- > 0 && norm(r) > eFactor * (1 + norm(b)))
    {
        Vector<double> ap = A * p;
        if(!isSPSD) ap = AT * ap;
        double rz = dotProduct(r, z), a = rz/dotProduct(p, ap);
        if(!isfinite(a)) break;
        x += p * a;
        r -= ap * a;
        z = pInv * r;
        p = z + p * (dotProduct(r, z)/rz);
    }
    return make_pair(x, norm(r));
}

```

Предобусловливатели, заранее выбранные для предметной области, обычно работают лучше, чем общие.

## 22.22. Итерационные методы для собственных значений

Поиск собственных значений в разреженной матрице устроен сложнее (см. [22.18, 22.42]). Бессмысленно вычислять все собственные векторы, потому что это заняло бы слишком много места, поэтому приходится довольствоваться меньшим. Обычно получают некоторые собственные векторы и собственные значения, а итерационные методы возвращают те из них, которые представляют интерес для определенных задач.

В симметричном случае *цикл Ланцоша* (см. [22.53]) позволяет получить все собственные пары или некоторые приближенные. В случае симметричной матрицы он создает тридиагональную матрицу, аналогичную преобразованию формы Хессенберга, но:

- ♦ вычисляет один столбец матрицы  $V$  за раз за счет одного произведения матрицы на вектор;
- ♦ может остановиться после  $m < n$  итераций, чтобы получить матрицу, собственные значения которой достаточно близки к некоторым из собственных значений  $A$ .

Реализация возвращает 5-диагональную матрицу, чтобы результат можно было передать как есть в симметричный цикл в матрице  $QR$ , которой нужны дополнительные диагонали. Алгоритм производит  $V$  и  $T$ , где в идеале  $V^T T V \approx A$ , если  $m$  достаточно велико или равно  $n$ :

```
template<typename MATRIX> BandMatrix<5> LanczosEigReduce(MATRIX const& A,
    Vector<Vector<double>>*> vs = 0, int m = -1,
    Vector<double> v = Vector<double>())
{
    int n = A.getRows();
    if(m == -1) m = n;
    if(v.getSize() == 0) v = GlobalRNG().randomUnitVector(n);
    assert(A.getColumns() == n && v.getSize() == n);
    BandMatrix<5> result(m);
    double b = 1;
    Vector<double> prevV;
    for(int i = 0; i < m; ++i)
    {
        if(vs) vs->append(v);
        Vector<double> w = A * v;
        double a = dotProduct(w, v);
        result(i, i) = a;
        w -= v * a;
        if(i > 0)
        {
            w -= prevV * b;
            result(i, i - 1) = result(i - 1, i) = b;
        }
        b = norm(w);
        if(b < numeric_limits<double>::epsilon())
        { // продолжать нельзя, поэтому возвращаем то, что есть
            BandMatrix<5> result2(i + 1);
            for(int j = 0; j <= i; ++j)
            {
                result2(j, j) = result(j, j);
                if(j > 0)
                    result2(j, j - 1) = result2(j - 1, j) = result(j, j - 1);
            }
            return result2;
        }
        prevV = v;
        v = w * (1/b);
    }
    return result;
}
```

Примените тридиагональную итерацию  $QR$  к результату, чтобы получить собственное разложение:

```
template<typename MATRIX> pair<Vector<double>, Matrix<double> >
    LanczosEigenSymmetric(MATRIX const& A, int m = -1, int maxIter = 1000,
        double prec = highPrecEps)
{ // строки Q являются собственными векторами
    int n = A.getRows();
    if(m == -1) m = n;
    Vector<Vector<double> > vs;
    BandMatrix<5> T = LanczosEigReduce(A, &vs, m);
    Matrix<double> QT(vs.getSize(), n);
    for(int c = 0; c < vs.getSize(); ++c)
        for(int r = 0; r < n; ++r) QT(c, r) = vs[c][r];
    Vector<double> eigve = QREigenTridiagonal(T, &QT, maxIter, prec);
    return make_pair(eigve, QT);
}
```

Если в матрице  $A$   $O(n)$  записей, время выполнения равно  $O(n)$ . К сожалению, цикл Ланцоша нестабилен в том смысле, что поздние значения  $v_i$  все сильнее отклоняются от ортогональности. Приведенная реализация позволит получить около 10 самых крайних приближенных собственных пар очень большой матрицы. Они могут быть очень приближительными, но все равно будут полезны — например, для оценки числа обусловленности. В разделе комментариев приведен лучший метод.

Для асимметричных матриц есть *цикл Арнольди*, но он вычисляет матрицу Хессенберга по одному столбцу за раз, поэтому его необходимо остановить до того, как закончится память, и получить только некоторые собственные значения. На этом цикле мы останавливаться не будем, но описанный в разделе комментариев метод в этом случае также применим.

## 22.23. Введение в интервальную арифметику

Арифметику с плавающей точкой можно дополнить *интервальной арифметикой*. Ее основные идеи (см. [22.44]) таковы:

- ◆ истинный результат операции находится между округленными значениями в меньшую и большую сторону;
- ◆ это правило позволяет получить простые оценки для стандартных арифметических операций;
- ◆ в C++ есть потокобезопасный API для управления режимом округления, поэтому такую арифметику можно реализовать;
- ◆ используя этот тип, можно выразить множество численных алгоритмов.

Существует также стандарт интервальной арифметики IEEE 1788-2015. Надеюсь, когда-нибудь в нем удастся избавиться от основного препятствия к его использованию — отсутствия в стандартной библиотеке поддержки элементарных функций.

Интервальная арифметика устраняет только ошибку оценки с плавающей точкой, но ошибки аппроксимации и оптимизации никуда не деваются. Для нечисловых вычислений — таких как тест CCW в вычислительной геометрии (см. главу 18. *Вычислительная геометрия*), это очень полезно, потому что вы можете выполнять вычисления в интервальной арифметике и переключаться на точную рациональную арифметику только тогда, когда границы интервала не гарантируют правильность.

В работе [22.44] приводится базовая реализация класса интервальной арифметики. Более чистая и портативная реализация приведена далее. Операция умножения выполняется сложно, поэтому ее лучше посмотреть в работе [22.44], чтобы доказать правильность всех случаев. См. также работу [22.30] и ссылки в ней.

```
// приведенный здесь код, хотя и является частью стандарта C99,
// не поддерживается некоторыми компиляторами, например, clang
#pragma STDC FENV_ACCESS ON
template<typename ITEM = double>
class IntervalNumber: public ArithmeticType<IntervalNumber<ITEM> >
{ // все операции должны использовать ближайшее округление для других вычислений
    ITEM left, right;
public:
    IntervalNumber(ITEM rounded)
    { // точное постоянное сложение - заменить на ulp?
        std::fesetround(FE_DOWNWARD);
        left = rounded * (1 + numeric_limits<ITEM>::epsilon());
        std::fesetround(FE_UPWARD);
        right = rounded * (1 - numeric_limits<ITEM>::epsilon());
        std::fesetround(FE_TONEAREST);
    }
    IntervalNumber(long long numerator, long long denominator)
    {
        std::fesetround(FE_DOWNWARD);
        left = numerator/denominator;
        std::fesetround(FE_UPWARD);
        right = numerator/denominator;
        std::fesetround(FE_TONEAREST);
    }
    bool isfinite()const{return std::isfinite(left) && std::isfinite(right);}
    bool isnan()const{return std::isnan(left) && std::isnan(right);}
    IntervalNumber& operator+=(IntervalNumber const& rhs)
    {
        std::fesetround(FE_DOWNWARD);
        left += rhs.left;
        std::fesetround(FE_UPWARD);
        right += rhs.right;
        std::fesetround(FE_TONEAREST);
        return *this;
    }
    IntervalNumber operator-()const // точное вычитание с плавающей точкой
    {return IntervalNumber(-right, -left);}
    IntervalNumber& operator-=(IntervalNumber const& rhs)
    {return *this += -rhs;}
    IntervalNumber& operator*=(IntervalNumber const& rhs)
    {
        std::fesetround(FE_DOWNWARD);
        ITEM temp1 = min(left * rhs.left, left * rhs.right);
        ITEM temp2 = min(right * rhs.left, right * rhs.right);
        ITEM newLeft = min(temp1, temp2);
        std::fesetround(FE_UPWARD);
```



```

    temp1 = max(left * rhs.left, left * rhs.right);
    temp2 = max(right * rhs.left, right * rhs.right);
    ITEM newRight = max(temp1, temp2);
    std::fesetround(FE_TONEAREST);
    left = newLeft;
    right = newRight;
    return *this;
}
IntervalNumber& operator*=(long long a)
{ // точное постоянное умножение
    std::fesetround(FE_DOWNWARD);
    left *= a;
    std::fesetround(FE_UPWARD);
    right *= a;
    std::fesetround(FE_TONEAREST);
    return *this;
}
bool contains(ITEM a) const { return left <= a && a <= right; }
IntervalNumber& operator/=(IntervalNumber rhs)
{
    if(rhs.contains(ITEM(0)))
    {
        rhs.left = -numeric_limits<ITEM>::infinity();
        rhs.right = numeric_limits<ITEM>::infinity();
    }
    else
    {
        std::fesetround(FE_DOWNWARD);
        rhs.left = ITEM(1)/rhs.right;
        std::fesetround(FE_UPWARD);
        rhs.right = ITEM(1)/rhs.left;
    }
    return (*this) *= rhs;
}
};

```

Следующим шагом является реализация алгоритмов, представленных здесь и в следующей главе, но это не такая простая задача, поэтому для интервальной арифметики было разработано множество специальных алгоритмов. При этом возникают проблемы:

- ◆ различные эквивалентные математические формулы дают интервалы разной ширины. Найти наилучшее синтаксическое дерево вообще практически невозможно, поэтому обычно реализуют естественную последовательность операций, если нет более специфической информации о проблеме;
- ◆ функция интервала обычно является мультиинтервалом. Например, представим решение уравнения  $s f(x) = \sqrt{x}$ . Быстрое решение состоит в том, чтобы взять интервал выпуклой оболочки мультиинтервалов, но это расточительно, особенно для многомерных диагональных множеств, которые малы по сравнению с ограничивающим диапазоном.

Таким образом, интервальная арифметика полезна в некоторых задачах численного анализа, но в целом — не слишком.

## 22.24. Примечания по реализации

Выполняемые через эпсилон сравнения с точностью по умолчанию являются оригинальными. Они встречаются во многих алгоритмах.

В этой главе представлены наиболее полезные матричные алгоритмы, но это лишь малая часть от того, что существует. В работе [22.21] описаны почти все, и главная трудность заключалась в сокращении многих других. Я смог сделать это, основываясь на разложении LUP и QR, без использования специализированных алгоритмов для структурированных матриц с лучшими постоянными факторами.

## 22.25. Комментарии

IEEE 754-2019 также стандартизирует арифметику с плавающей точкой по основанию 10. Это необходимо в финансовых расчетах, которые должны давать точные результаты, — так, число  $\frac{1}{3}$  не является двоичной аппроксимацией. Некоторые законы — например, о налоговых расчетах, определяют правила десятичного счета и округления, которые позволяют точно реализовать десятичную арифметику. В работе [22.29] приведено отличное введение в арифметику с плавающей точкой, а в работе [22.28] и в упомянутом стандарте дается более подробная информация. Возможно, самое важное практическое изменение заключается в том, что формальные доказательства становятся более осуществимыми — например, подробности об инструменте *gappa* приведены в работе [22.6]).

Числа обусловленности обычно определяются расплывчатой «границей величины изменения» и впоследствии считаются производными от истинного решения  $F$  по отношению к входным данным  $x$ . Мое определение, основанное на модуле непрерывности, стремится к большей обобщенности.

Интересная идея состоит в создании *машинно-независимых* проверок допусков (см. [22.20]), в которых между сравниваемой величиной (абсолютной или относительной) и возможным ответом не будет машинных чисел. Многие алгоритмы можно выразить без какого-либо упоминания  $\epsilon$  или с использованием хитрых сравнительных тестов. Плюсы такого подхода:

- ◆ возможность выжать максимальную точность;
- ◆ реализация оказывается немного проще с точки зрения критериев сходимости.

Минусы:

- ◆ полученная точность может оказаться иллюзорной, т. к. ошибки округления все равно присутствуют. Например, рассмотрите возможность нахождения корня  $f(x) = 1 - e^{-x}$ . Ответ — 0, но глупо сводить его к какому-то наименьшему нормальному числу, потому что  $f(x) \approx 0$  при  $x \approx \epsilon$  из-за вычитания;
- ◆ процесс может быть очень неэффективным. В случае относительных сравнений регистры ЦП используют более точные биты, и процесс может попытаться правильно их получить и округлить в меньшую сторону, чтобы сохранить ответ в памяти. Сравнения с 0 выполняются намного хуже, потому что вычисления будут продолжаться до тех пор, пока не будет достигнуто наименьшее нормальное (или, возможно, субнормальное) число.

Умножение матриц кажется простым, но в популярных библиотеках оно сильно оптимизировано. В работе [22.21] авторы обсуждают ряд связанных с этим вопросов. На параллельном компьютере выражение матричного произведения в виде суммы внешних произведений реализуется выгодно, потому что в этом случае используются *операции BLAS-2*, которые могут быть оптимизированы при сборке и специализированы для аппаратного обеспечения. Другое направление — использование быстрых матричных алгоритмов — таких как алгоритм Штрассена, который с небольшим отрывом превосходит обычный алгоритм для  $n > 100$ . Но у этого алгоритма нет покомпонентной устойчивости, а границы на основе норм характеризуются худшими постоянными факторами (см. [22.22]).

Решения уравнений с использованием LUP или QR можно дополнительно улучшить путем *итеративного уточнения*:

- ◆ теоретически получить покомпонентную устойчивость (см. [22.22]);
- ◆ практически получить пониженную чувствительность к исходному масштабу.

Подробности можно найти в работе [22.21]. Стоимость равна  $O(n^2)$ , поэтому решение с той же факторизацией возможно даже для повторяющегося уравнения. Но нужно хранить исходную матрицу, что может сделать процесс нецелесообразным.

*Вращение Гивенса* дает тот же эффект, что и сокращение Хаусхолдера для  $x$  размера 2, но его вычисление выполняется немного эффективнее (см. [22.21]). По некоторым элементам Хаусхолдер быстрее. Поскольку вычисление вращения выполняется быстрее, чем его применение, использование вращения Гивенса является ненужной оптимизацией.

Для вычисления собственных значений реальной симметричной матрицы было установлено несколько конкурирующих методов (см. [22.18]). По сравнению с циклом QR некоторые из них быстрее, а некоторые точнее, но ненамного. Только в случае очень маленьких собственных значений, близких к используемому  $\epsilon$ , может оказаться целесообразным задействовать несколько более медленный *алгоритм Якоби*, который может быть реализован как машинно-независимый (подробности см. в работах [22.18, 22.21 и 22.20]).

Идея использования случайных значений, взятых из кругов Гершгорина, является новой — похоже, она еще не открыта, несмотря на очевидную гарантию сходимости.

*Прямые методы* для разреженных матриц стремятся сохранять разреженность промежуточных результатов. Например, для исключения Гаусса, если первый столбец и строка выбранной опорной точки являются плотными, разреженность теряется. Для решения этой проблемы было предложено множество различных эвристических методов (см. [22.16]), в частности:

- ◆ локальные методы — такие как *стратегия Марковица*: выберите опорную точку, которая сводит к минимуму немедленное заполнение, но для сохранения стабильности она не должна быть намного меньше обычной опорной точки;
- ◆ глобальные методы сужения полосы — такие как обратный *алгоритм Катхилла — Макки*, интерпретируют матрицу как граф и переставляют вершины определенным жадным образом.

Для задач вроде факторизации Холецкого, которая работает с матрицами SPSPD и не требует поворота, обратное переупорядочение Катхилла — Макки сохраняет SPSPD,

только переставляя переменные по диагонали. Затем нужно выполнить факторизацию Холецкого как обычно, но работать уже с разреженной матрицей и использовать специальные алгоритмы (см. [22.16, 22.13]). Неясно, как результат конкурирует с сопряженным градиентом с общим предварительным кондиционированием. В худшем случае первое может значительно увеличить использование памяти, а второе — потребовать множества итераций. Итерационные методы решения уравнений для различных случаев более подробно обсуждаются в работе [22.21] и ссылок в ней. Проблема в том, что CGNR может сходиться слишком медленно, а другие алгоритмы пытаются это исправить ценой других преимуществ:

- ◆ GMRES — основной алгоритм для общих матриц, но он может исчерпать память до сходимости, если не использовать стратегию перезапуска;
- ◆ многие другие методы пытаются сохранить низкое использование памяти CGNR и сходятся быстрее, но у них возникают проблемы устойчивости. Непонятно, какой из них лучший.

Автоматические предобуславливатели тоже становятся сложнее. Например, для CG одним из вариантов является *неполная факторизация Холецкого* — т. е. вычисление факторизации Холецкого разреженной матрицы, но без сохранения элементов там, где их нет в матрице. Для решения результирующих разреженных матрично-векторных уравнений требуется дополнительный код, но метод работает хорошо.

Проблемы с ортогональностью в цикле Ланцоша с *полной реортогонализацией* (см. [22.18]) можно устранить за счет дополнительной работы  $O(m)$ . Это осуществимо только для малых  $m$ , где маловероятное накопление ошибок округления возникает достаточно поздно, чтобы вызвать проблемы. Обычно используется программное обеспечение ARPACK, которое основано на цикле Арнольди/Ланцоша с полной реортогонализацией и методе, называемом *неявным перезапуском*. Предположим, что требуется  $k$  наименьших собственных пар. Тогда:

1. Выполняйте цикл Ланцоша, пока не получите  $2k$  или около того.
2. Пока не достигнуто схождение:
3. Выполняйте сортировку по собственному значению.
4. Удалите самое большое  $k$ .
5. Перезапустите итерацию с наименьшего  $k$ .

Для получения подробной информации, в частности, о том, как перезапустить цикл, обратитесь к работе [22.3]. В работе [22.47] приведена теория. Неясно, сколько именно дополнительных вычислений понадобится, и при каких условиях процесс гарантированно сойдется, но обычно он сходится и является лучшим доступным решением. Тем не менее проблема надежного и эффективного поиска нескольких наименьших собственных значений большой разреженной матрицы остается открытой. Теоретически также есть проблемы, особенно с собственными векторами (см. [22.25]).

Численная линейная алгебра — очень обширная тема, и все упомянутые далее книги будут полезны для дальнейшего чтения. В дополнение к уже упомянутым книгам, в работе [22.23] приведена обширная коллекция обзоров по теоретическим и числовым вопросам линейной алгебры. Работа [22.5] также представляет интерес. Итеративное решение линейных систем наиболее полно описано в книге [22.34]. Об итеративном

вычислении собственных значений и анализе алгоритмов также можно почитать в работе [22.35].

## 22.26. Советы по дополнительной подготовке

- ♦ Создайте конструктор строк для интервальной арифметики. Также допускается создание из широкого интервала, а не из однорядного.
- ♦ Нужно ли ограничивать у тридиагональных систем диагональное доминирование некоторым значением  $\epsilon$  для устойчивости решения? Исследуйте этот вопрос.
- ♦ В реализованном виде предобуславливатель Якоби для CG неустойчив к нулям на диагонали. Реализуйте некоторые стратегии для решения этой проблемы. Простой подход состоит в том, чтобы использовать средневзвешенное значение со средним значением всех диагональных элементов, но оно все еще может быть равно 0, поэтому в последнем случае может быть получено среднее значение 1.
- ♦ Реализуйте неявный перезапуск цикла Ланцоша.
- ♦ Для числа интервала постройте целочисленную дробь.

## 22.27. Список рекомендуемой литературы

- 22.1. Acton F. S. (1996). Real Computing Made Real: Preventing Errors in Scientific and Engineering Calculations. Dover.
- 22.2. Aurentz J. L., Mach T., Robol L., Vandebriel R., & Watkins D. S. (2018). Core-Chasing Algorithms for the Eigenvalue Problem. SIAM.
- 22.3. Bai Z., Demmel J., Dongarra J., Ruhe A., & van der Vorst H. (Eds). (2000). Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide. SIAM.
- 22.4. Beebe N. H. F. (2017). The Mathematical-Function Computation Handbook. Springer.
- 22.5. Björck Å. (2014). Numerical Methods in Matrix Computations. Springer.
- 22.6. Boldo S., & Melquiond G. (2017). Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System. Elsevier.
- 22.7. Bornemann F. (2016). The SIAM 100-digit challenge: a decade later. Jahresbericht der Deutschen Mathematiker-Vereinigung, 118(2), 87–123.
- 22.8. Bornemann F., Laurie D., Wagon S., & Waldvogel J. (2004). The SIAM 100-digit Challenge: a Study in Highaccuracy Numerical Computing. SIAM.
- 22.9. Cody W. J. (1974). The construction of numerical subroutine libraries. SIAM Review, 16(1), 36–46.
- 22.10. Corless R. M., & Fillion N. (2013). A Graduate Introduction to Numerical Methods. Springer.
- 22.11. Cormen T. H., Leiserson C. E., Rivest R. L., & Stein C. (2009). Introduction to Algorithms. MIT Press.
- 22.12. Datta B. N. (2010). Numerical Linear Algebra and Applications. SIAM.
- 22.13. Davis T. A. (2006). Direct Methods for Sparse Linear Systems. SIAM.
- 22.14. Day D. (1996). How the QR algorithm fails to converge and how to fix it.
- 22.15. Deuffhard P., & Hohmann A. (2003). Numerical Analysis in Modern Scientific Computing: An Introduction. Springer.

- 22.16. Duff I. S., Erisman A. M., & Reid J. K. (2017). *Direct Methods for Sparse Matrices*. Oxford University Press.
- 22.17. Fausett L. V. (2003). *Numerical Methods: Algorithms and Applications*. Pearson.
- 22.18. Ford W. (2014). *Numerical Linear Algebra with Applications: Using MATLAB*. Academic Press.
- 22.19. Free Software Foundation (2017). GNU Scientific Library 2.4.  
<https://www.gnu.org/software/gsl/doc/html/index.html>.
- 22.20. Gander W., Gander M. J., & Kwok F. (2014). *Scientific Computing — An Introduction using Maple and MATLAB*. Springer.
- 22.21. Golub G. H., & Van Loan C. F. (2012). *Matrix Computations*. JHU Press.
- 22.22. Higham N. J. (2002). *Accuracy and Stability of Numerical Algorithms*. SIAM.
- 22.23. Hogben L. (2013). *Handbook of Linear Algebra*. CRC.
- 22.24. Ipsen I. C. (1997). Computing an eigenvector with inverse iteration. *SIAM Review*, 39(2), 254–291.
- 22.25. Kuczyński J., & Woźniakowski H. (1992). Estimating the largest eigenvalue by the power and Lanczos algorithms with a random start. *SIAM journal on matrix analysis and applications*, 13(4), 1094–1122.
- 22.26. Miller W. (1984). *The Engineering of Numerical Software*. Prentice-Hall.
- 22.27. Muller J. M. (2016). *Elementary Functions*. Birkhäuser.
- 22.28. Muller J. M., Brunie N., de Dinechin F., Jeannerod C. P., Joldes M., Lefèvre V., Melquiond G., Revol N., and Torres S. (2018). *Handbook of Floating-Point Arithmetic*. Springer.
- 22.29. Overton M. L. (2001). *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM.
- 22.30. Moore R. E., Kearfott R. B., & Cloud M. J. (2009). *Introduction to Interval Analysis*. SIAM.
- 22.31. Nash J. C. (2014). *Nonlinear Parameter Optimization Using R Tools*. Wiley.
- 22.32. Powell M. J. D. (1981). *Approximation Theory and Methods*. Cambridge University Press.
- 22.33. Rice J. R. (1992). *Numerical Methods, Software, and Analysis*. Academic Press.
- 22.34. Saad Y. (2003). *Iterative Methods for Sparse Linear Systems*. SIAM.
- 22.35. Saad Y. (2011). *Numerical Methods for Large Eigenvalue Problems: Revised Edition*. SIAM.
- 22.36. Shampine L. F., & Reichelt M. W. (1997). The MATLAB ode suite. *SIAM Journal on Scientific Computing*, 18(1), 1–22.
- 22.37. StackOverflow (2017). Convention on non-negative singular values?  
<https://math.stackexchange.com/questions/170746/convention-on-non-negative-singular-values>. Accessed March 5, 2017.
- 22.38. Stewart G. W., & Sun J. G. (1990). *Matrix Perturbation Theory*. Academic Press.
- 22.39. Süli E., & Mayers D. F. (2003). *An Introduction to Numerical Analysis*. Cambridge University Press.
- 22.40. Trangenstein J. A. (2018b). *Scientific Computing: Vol. II: Eigenvalues and Optimization*. Springer.
- 22.41. Traub J. F., Werschulz A. G. (1998). *Complexity and Information*. Cambridge University Press.
- 22.42. Trefethen L. N., & Bau III D. (1997). *Numerical Linear Algebra*. SIAM.
- 22.43. Trefethen L. N., & Embree M. (2005). *Spectra and Pseudospectra: The Behavior of Nonnormal Matrices and Operators*. Princeton University Press.
- 22.44. Tucker W. (2011). *Validated Numerics: A Short Introduction to Rigorous Computations*. Princeton University Press.
- 22.45. Ueberhuber C. W. (1997a). *Numerical Computation 1: Methods, Software, and Analysis*. Springer.
- 22.46. Ueberhuber C. W. (1997b). *Numerical Computation 2: Methods, Software, and Analysis*. Springer.
- 22.47. Watkins D. S. (2007). *The Matrix Eigenvalue Problem: GR and Krylov subspace methods*. SIAM.

- 22.48. Watkins D. S. (2011). Francis's algorithm. *The American Mathematical Monthly*, 118(5), 387–403.
- 22.49. Watkins D. S., Elsner L. (1991). Convergence of algorithms of decomposition type for the eigenvalue problem. *Linear Algebra and Its Applications*, 143, 19–47.
- 22.50. Wikipedia (2017b). Eigendecomposition of a matrix. [https://en.wikipedia.org/wiki/Eigendecomposition\\_of\\_a\\_matrix](https://en.wikipedia.org/wiki/Eigendecomposition_of_a_matrix). Accessed March 5, 2017.
- 22.51. Wikipedia (2017c). Singular value decomposition. [https://en.wikipedia.org/wiki/Singular\\_value\\_decomposition](https://en.wikipedia.org/wiki/Singular_value_decomposition). Accessed March 5, 2017.
- 22.52. Wikipedia (2017d). Taylor's theorem. [https://en.wikipedia.org/wiki/Taylor's\\_theorem](https://en.wikipedia.org/wiki/Taylor's_theorem). Accessed May 6, 2017.
- 22.53. Wikipedia (2017e). Lanczos algorithm. [https://en.wikipedia.org/wiki/Lanczos\\_algorithm](https://en.wikipedia.org/wiki/Lanczos_algorithm). Accessed August 13, 2017.

## 23. Численные алгоритмы: работа с функциями

У большинства функций теория сходимости полезна только в качестве моральной поддержки.  
*Джон Бойд*

Не спрашивайте, провалится ли решение. Спрашивайте, когда оно удастся?  
*Уильям Кахан*

### 23.1. Введение

Эта глава продолжается там, где остановилась предыдущая. На этот раз мы сосредоточимся на численных методах для функций. В частности, мы обсудим алгоритмы интерполяции, дифференцирования, интегрирования, решения систем уравнений и ОДУ.

### 23.2. Быстрое преобразование Фурье

Если дано  $n$  комплексных чисел  $x_j$ , *дискретное преобразование Фурье* (ДПФ) вычисляет  $y_k = \sum x_j e^{-2\pi j k i / n}$  ( $i$  здесь является комплексным числом) для  $0 \leq k < n$  (см. [23.65]).

Этот расчет является важной частью многих алгоритмов, в том числе тех, которые не работают с комплексными числами. Вычисление всех  $y_k$  по приведенным ранее формулам выполняется за  $O(n^2)$ .

*Быстрое преобразование Фурье* (БПФ) делает это за  $O(n \lg(n))$ . В самом простом случае  $n$  является степенью двойки. Числа  $e^{-2\pi j k i / n}$  — корни из единицы, равномерно расположенные на единичной окружности комплексной плоскости. Некоторые высокочастотные точки перекрывают некоторые низкочастотные. В частности, некоторые произведения  $x_j e^{-2\pi j k i / n}$  появляются в разных  $y_k$  несколько раз. Таким образом, БПФ вычисляет общие произведения только один раз. В работе [23.13] приведено дальнейшее обсуждение алгоритма. Вопреки стандартному определению ДПФ их реализация использует позитивно-степенные корни (в некоторых других источниках, таких как [23.51], они тоже используются):

```
complex<double> unityRootHelper(int j, int n)
{
    return exp((j * PI() / n) * complex<double>(0, 1));
}
Vector<complex<double>> FFTPower2(Vector<complex<double>> const&x)
{
    int n = x.getSize(), b = lgFloor(n);
    assert(isPowerOfTwo(n));
    typedef complex<double> C;
    Vector<C> result(n);
```



```

for(unsigned int i = 0; i < n; ++i) result[reverseBits(i, b)] = x[i];
for(int s = 1; s <= b; ++s)
{
    int m = twoPower(s);
    C wm = unityRootHelper(-2, m);
    for(int k = 0; k < n; k += m)
    {
        C w(1, 0);
        for(int j = 0; j < m/2; ++j, w *= wm)
        {
            C t = w * result[k + j + m/2], u = result[k + j];
            result[k + j] = u + t;
            result[k + j + m/2] = u - t;
        }
    }
}
return result;
}

```

Относительная прямая ошибка БПФ степени двойки равна  $O(\varepsilon_{\text{machine}} \lg(n))$  в 2-норме, если предполагается вычисление корней объединения вблизи  $\varepsilon_{\text{machine}}$  (см. [23.33]).

*Обратное ДПФ* определяется как  $y_k = \frac{1}{n} \sum x_j e^{2\pi j k i / n}$  (см. [23.65]). Можно вычислить его путем приведения к обычному ДПФ, вычисленному по БПФ. Помимо множителя  $1/n$ , «—» в экспоненте работает с теми же корнями объединения, только в другом порядке для  $j \neq 0$ . Порядок для  $j > 0$  фактически обратный и может изменить выход или вход БПФ для  $x$  (см. [23.43]). Первый вариант позволяет передавать ввод по постоянной ссылке:

```

Vector<complex<double>> > IFFTHelper(Vector<complex<double>> > fftx)
{
    int n = fftx.getSize();
    fftx.reverse(1, n - 1);
    return fftx * (1.0/n);
}
Vector<complex<double>> > inverseFFTPower2(
    Vector<complex<double>> > const& x)
{
    assert(isPowerOfTwo(x.getSize()));
    return IFFTHelper(FFTPower2(x));
}

```

Когда  $n$  не равно степени двойки, оно приводится к подходящей степени двойки. В некоторых задачах можно использовать БПФ путем добавления 0 к данным, чтобы сделать их степенью двойки. В общем случае работает *алгоритм Блестейна* (см. [23.66]). Даны две дискретные последовательности  $a$  и  $b$  размера  $n$ , такие что  $b_{-l} = b_{n-l}$ , их *свертка*  $= y$ , где  $y_k = \sum a_j b_{k-j}$ . Кроме того, свертка  $(a, b) = \text{FFT}^{-1}$  — попарное произведение  $\text{FFT}(a)$ ,  $\text{FFT}(b)$ , и на свертки не влияет 0-заполнение:

```

Vector<complex<double>> > convolutionPower2(Vector<complex<double>> > const& a,
    Vector<complex<double>> > const& b)

```

```

{
    int n = a.getSize();
    assert(n == b.getSize() && isPowerOfTwo(n));
    Vector<complex<double>> fa = FFTPower2(a), fb = FFTPower2(b);
    for(int i = 0; i < n; ++i) fa[i] *= fb[i];
    return inverseFFTPower2(fa);
}

```

Можно привести БПФ к свертке степени двойки. Степень ДПФ является функцией:

$$jk = (j^2 + k^2 - (k - j)^2) / 2.$$

Пусть  $r(j, n) = e^{-\pi j / n}$ . Тогда  $y_k = r(-k^2, n) \sum a_j b_{k-j}$ , где  $a_j = x_j r(-j^2, n)$  и  $b_{k-j} = r(-(k-j)^2, n)$ . Здесь  $a$  и  $b$  — функции, но они должны быть массивами, такими что:

- ◆ они дополнены до длины  $m$ , являющейся степенью двойки;
- ◆  $B_{-l} = b_{n-l}$ , где  $l = k - j$ .

Мы выбираем  $m$  равным следующей степени двойки после  $(2n - 1)$ . Тогда

$$A_j = \begin{cases} a_j, & \text{если } j < n \\ 0, & \text{в противном случае} \end{cases} \text{ и } B_l = B_{m-l} = \begin{cases} b_j, & \text{если } j < n \\ 0, & \text{в противном случае} \end{cases}.$$

```

Vector<complex<double>> FFTGeneral(Vector<complex<double>> const& x)
{ // Алгоритм Блукштейна
    int n = x.getSize(), m = nextPowerOfTwo(2 * n - 1);
    if(isPowerOfTwo(n)) return FFTPower2(x);
    Vector<complex<double>> a(m), b(m); // нулевое дополнение
                                     // в конструкторе по умолчанию

    for(int j = 0; j < n; ++j)
    {
        a[j] = x[j] * unityRootHelper(-j * j, n);
        b[j] = unityRootHelper(j * j, n); // можно вычислить b и БПФ
        if(j > 0) b[m - j] = b[j];
    }
    Vector<complex<double>> ab = convolutionPower2(a, b);
    while(ab.getSize() > n) ab.removeLast();
    for(int k = 0; k < n; ++k) ab[k] *= unityRootHelper(-k * k, n);
    return ab;
}

Vector<complex<double>> IFFTGeneral(Vector<complex<double>> const& x)
{ return IFFTHelper(FFTGeneral(x)); }

```

Используемая здесь математика хорошо описана в работе [23.56].

Входные значения часто бывают действительными, и это позволяет сэкономить ресурсы. В частности, можно упаковать две действительные последовательности  $x$  и  $y$  в одну комплексную последовательность  $z$  и из ее БПФ получить их БПФ за время  $O(n)$ . Алгоритм таков (см. [23.38]):

1. Сформировать  $z$  как  $z_j = x_j + iy_j$ .
2. Пусть  $z_f = \text{FFT}(z)$ .
3. Сформировать  $x_f$  как  $x_f[j] = (z_f[j] + \text{сопряжение}(z_f[n - (j \% n)])) / 2$ .

4. Сформировать  $y_f$  как  $y_f[j] = (z_f[j] - \text{сопряжение}(z_f[n - (j)\%n]))i / 2$ .

5. Вернуть  $x_f$  и  $y_f$ .

```
pair<Vector<complex<double> >, Vector<complex<double> > > FFTReal2Seq(
    Vector<double> const& x, Vector<double> const& y)
{
    int n = x.getSize();
    assert(n == y.getSize());
    typedef complex<double> C;
    typedef Vector<C> VC;
    VC z(n);
    for(int i = 0; i < n; ++i) z[i] = C(x[i], y[i]);
    VC zf = FFTGeneral(z);
    pair<VC, VC> result;
    for(int i = 0; i < n; ++i)
    {
        C temp = conj(zf[(n - i) % n]);
        result.first.append(0.5 * (zf[i] + temp));
        result.second.append(0.5 * (zf[i] - temp) * C(0, -1));
    }
    return result;
}
```

Дальнейшее усовершенствование заключается в использовании этого алгоритма для вычисления БПФ одной действительной последовательности из четных  $n$ . Сокращение представляет собой один шаг рекурсивного БПФ степени двойки (см. [23.13] — опять же, будьте осторожны, там используются положительные показатели степени):

```
Vector<complex<double> > FFTRealEven(Vector<double> const& x)
{
    int n = x.getSize(), n2 = n/2;
    assert(n % 2 == 0);
    typedef complex<double> C;
    typedef Vector<C> VC;
    Vector<double> xOdd(n2), xEven(n2);
    for(int i = 0; i < n; ++i) (i % 2 ? xOdd[i/2] : xEven[i/2]) = x[i];
    pair<VC, VC> xSplitF = FFTReal2Seq(xEven, xOdd);
    VC xF(n);
    C wn = unityRootHelper(-2, n), wi(1, 0);
    for(int i = 0; i < n2; ++i, wi *= wn)
    {
        xF[i] = xSplitF.first[i] + wi * xSplitF.second[i];
        xF[n2 + i] = xSplitF.first[i] - wi * xSplitF.second[i];
    }
    return xF;
}
```

Многие другие полезные преобразования сводятся к вычислению БПФ. Частным случаем является *дискретное косинусное преобразование* (ДКП) *типа I*, которое определяется как  $\text{DCTI}(x)_k = \sum_{0 \leq j \leq n} x_j \cos(jk\pi / n)$ . Двойной штрих у знака суммы обозначает, что первый и последний члены делятся пополам (см. [23.14]). Например, это преобра-

зование используется для вычисления полиномиальной интерполяции Чебышева (обсуждается далее в этой главе). Вычислим его путем приведения к БПФ (см. [23.14]):

1. Пусть  $n$  = размер входных данных  $- 1$ .
2. Создать новый массив размера  $2n$ .
3. Установить  $y_j = x_j$ , где  $0 \leq j \leq n$  и  $y_{2n-j} = x_j$ , где  $1 \leq j \leq n$ .
4. Вернуть действительную часть первых  $n + 1$  элементов БПФ( $y$ )/2.

Для эффективности используем действительное БПФ последовательности четной длины:

```
Vector<double> DCTI(Vector<double> const& x)
{
    int n = x.getSize() - 1;
    assert(n > 0);
    Vector<double> y(2 * n), result(n + 1);
    for(int i = 0; i <= n; ++i) y[i] = x[i];
    for(int i = 1; i < n; ++i) y[2 * n - i] = x[i];
    Vector<complex<double>> yf = FFTRealEven(y);
    for(int i = 0; i <= n; ++i) result[i] = yf[i].real()/2;
    return result;
}
```

Для ДКП типа I обратное значение = ДКП  $\times 2/n$  (см. [23.69]):

```
Vector<double> IDCTI(Vector<double> const& x)
{ return DCTI(x) * (2.0/(x.getSize() - 1)); }
```

## 23.3. Интерполяция: общие идеи

Пусть нам нужно приблизить функцию  $f$  другой функцией, которая:

- ◆ обладает полезными аналитическими свойствами;
- ◆ дает достаточно хорошее приближение.

Обычно рассматривается только конечный интервал  $[a, b]$ . Эффективным с вычислительной точки зрения методом аппроксимации является интерполяция, при которой значения в оцененных точках  $f$  (часто называемых *узлами*) используются для оценки других. Исходя из информационной сложности, для любого набора узлов можно определить другую функцию, которая соответствует  $f$  в узловых точках, но в других местах ведет себя произвольно. Чтобы предотвратить проблемы, достаточно, чтобы  $f$  была липшицевой, что заставляет все возможные  $f$  вести себя одинаково.

Чтобы уменьшить ошибку аппроксимации, интерполянт должен:

- ◆ сходиться, когда  $f$  липшицева;
- ◆ сходиться с порядком, который соответствует числу непрерывных производных  $f$ ;
- ◆ не иметь больших колебаний между узлами, даже когда  $f$  не липшицева.

Узлы могут быть выбраны методом интерполяции или заданы заранее. Обычно в качестве интерполянтов используются полиномы:

- ◆ они обладают множеством известных аналитических свойств;
- ◆ основные операции выполняются быстро.
- ◆ качество приближения легко оценить.

*Теорема Вейерштрасса:* если функция  $f$  непрерывна на интервале  $[a, b]$ , то она может быть сколь угодно хорошо представлена полиномом. Так что задайтесь вопросом, можно ли интерполировать функцию, используя полиномы и увеличивая их степень вдвое, пока не будет достигнута сходимость. Это не работает только для непрерывных  $f$ , т. к. алгоритм, работающий с «черным ящиком»  $f$ , должен иметь некоторую стратегию выбора фиксированного узла, т. е. независимую от  $f$ , что не предполагается теоремой. *Теорема Эрдоса — Вертези:* для любой стратегии набора фиксированных узлов существует непрерывная  $f$  такая, что  $\limsup_{n \rightarrow \infty} (\text{интерполянт}(f, \text{набор узлов}(n))) = \infty$  выполняется почти везде (см. [23.19]). Выполняется «почти», потому что в узлах некоторые значения  $n$  лучше других, в зависимости от  $x$ .

Таким образом, теорема Вейерштрасса не нарушает требования Липшица, что позволяет предотвратить наихудшие случаи. Общая ошибка  $\rightarrow 0$ , даже несмотря на то, что постоянная Лебега медленно растет с ростом  $n$  (обсуждается далее в этой главе). Могут возникнуть проблемы, когда для соседних узлов значения  $f$  произвольно различны, а условие Липшица предотвращает большие наклоны. Если  $f$  имеет достаточно непрерывных производных, можно определить некоторые границы ошибки. *Теорема Лагранжа об ошибке интерполяции (LIE)* (см. [23.1 и 23.57]): при заданном наборе  $k + 1$  точек  $x_i$  в отсортированном порядке,  $f$  с  $k + 1$  непрерывными производными  $\in [x_0, x_k]$  и  $p$  — многочлен степени  $k$ , полученный путем сопоставления  $f$  в:

$$\{x_i\}, |p(x) - f(x)| \leq LIE(x, \{x_i\}, \|f^{(k+1)}\|_{\infty}) = \frac{\|f^{(k+1)}\|_{\infty} \prod (x - x_i)}{(k+1)!} \lim_{x \rightarrow \infty}.$$

Предположим, что нормы и результаты сохраняются на  $[x_0, x_k]$ , хотя в различных теоремах это не упоминается явно — например,  $\|f^{(k+1)}\|_{\infty}$  находится на  $[x_0, x_k]$  и может быть больше снаружи, что не имеет значения.

Пусть  $\frac{\|f^{(k+1)}\|_{\infty}}{(k+1)!} \leq$  некоторой константы  $M$ . Тогда  $\|p - f\|_{\infty} \leq M(x_k - x_0)^{k+1}$ , и для

$(x_k - x_0) \rightarrow 0$ , как часто бывает у численных методов, которые итеративно интерполируют сокращающиеся интервалы, получаем сходимость порядка  $k + 1$ . На практике не удастся найти границу дальней производной, поэтому формула показывает только сходимость и ее приблизительную скорость.

Границы точности зависят от предположений. Для LIE предположение состоит в том, что  $f$  имеет достаточно непрерывных производных. Например, производная (точнее, субградиент) функции абсолютного значения является ступенчатой функцией, а ее собственная производная является дельта-функцией, поэтому использование даже квадратичного многочлена в любой области, содержащей 0, кажется проблематичным.

Для более общих оценок нужны некоторые инструменты из теории приближения:

- ◆ Теорема (см. [23.52]): Если аппроксимант  $X$  является линейной проекцией, то  $\|f - X(f)\| \leq (1 + \|X\|)E(f)$ , где  $E(f)$  — ошибка наилучшего интерполянта из того же нормированного пространства, что и  $X$ . В частности, для многочленов,

$\|f - p\|_\infty \leq (1 + \Lambda(x_i)) E_n(f)$ , где  $\Lambda$  — постоянная Лебега узлов. Для произвольных узлов  $\Lambda(\{x_i\})$  есть функция интерполяции (см. [23.61]). Неравенство вытекает из логики, которая предполагает, что функция  $g$  близка как к  $f$ , так и к  $p$ , и выводит оценки на основе ее свойств (доказательство в работе [23.53] делает эту связь ясной). Обычно  $g$  — наилучшее приближение к  $f$  полиномом степени  $n$ .

♦ Пусть  $h_{\min} = \min_i (x_{i+1} - x_i)$  и  $h_{\max} = \max_i (x_{i+1} - x_i)$ . Тогда  $\Lambda(\{x_i\}) \leq \left(\frac{2h_{\max}}{h_{\min}}\right)^n$  (см. [23.53]). Для границ это неизвестная константа, не зависящая от масштабирования от другого интервала до  $[-1, 1]$ .

♦ Теорема Джексона (см. [23.52]: Пусть  $E_n(f) = \|f - g\|_\infty$  на  $[-1, 1]$ . Если  $f^{(k)}$  непрерывна, то  $E_n(f) \leq \frac{(n-k)! (\pi/2)^k}{n!} \|f^{(k)}\|_\infty$ .

♦ Теорема Джексона для липшицевой  $f$  (см. [23.52]:  $E_n(f) \leq \frac{L\pi}{2(n+1)}$ .

♦ Для  $(x_k - x_0) \rightarrow 0$  масштабируйте  $f$  до  $[-1, 1]$ , что масштабирует  $f^{(k)}$  к  $\left(\frac{x_k - x_0}{2}\right)^k$ .

Интуитивно, когда  $f$ -значения остаются неизменными, но расширяется к  $[-1, 1]$ ,  $f$  меняется быстрее. Для согласованности с производной прямой разности

$$f' = \frac{f(x+h) - f(x)}{h} \quad h \text{ увеличивается, а значения } f \text{ — нет. Константа Липшица } L$$

также масштабируется к  $\frac{x_k - x_0}{2}$ .

Обобщая сказанное, получаем, что для  $f$  с  $k$  производными для интерполяции с  $n$  точками на  $[-1, 1]$ :

$$|p(x) - f(x)| \leq \left(1 + \left(\frac{2h_{\max}}{h_{\min}}\right)^n\right) \frac{(n-k)! (\pi/2)^k}{n!} \|f^{(k)}\|_\infty.$$

В работе [23.53] даются немного другие константы из-за отличий в значениях границ для  $En(f)$ . Поскольку масштабирование является единственным переменным фактором, для фиксированного  $n$  и  $(x_k - x_0) \rightarrow 0$  имеем:

- ♦ сходимость липшицевых  $f$ ;
- ♦ сходимость порядка  $k$  для  $f$  с  $k$  непрерывными производными, с трудно определяемыми константами  $\left(\frac{2h_{\max}}{h_{\min}}\right)^n$ , минимизируется для равноотстоящих узлов, которые кажутся естественным выбором для интерполяции, хотя это и не так;
- ♦ для равноотстоящих узлов асимптотически  $\Lambda(\{x_i\}) \approx \frac{2^{n+1}}{en \log(n)}$ , что является одной из наихудших возможных конфигураций (см. [23.61]);

♦ для экстремальных узлов Чебышева (обсуждается в этой главе позже) на диапазоне  $[-1]$  получаем  $\Lambda(\{x_i\}) \leq \frac{2}{n} \log(n+1) + 1$ , что является асимптотически оптимальным, поскольку для любого фиксированного множества узлов  $\Lambda(x_i) \geq O(\lg(n))$  (см. [23.61]).

$\Lambda$  служит в качестве числа условий интерполяции — теорема (см. [23.14]):

$$\|p\|_{\infty} \leq \Lambda(\{x_i\}) \|f\|_{\infty}.$$

Из-за быстрораствующего  $\Lambda$  интерполяция в равноотстоящих узлах за пределами некоторой малой степени (обычно  $< 5$ ) является плохой идеей. На самом деле возникает расходимость за пределами малых  $n$ , потому что LIE может возрастать со степенью, если  $\|f^{(k+1)}\|_{\infty}$  очень быстро растет с ростом  $k$ . Типичным примером, где это происходит, является функция Рунге  $f(x) = \frac{1}{1+25x^2}$  с  $[a, b] = [-5, 5]$ . Таким образом, нет гарантированной сходимости для липшицевой  $f$  при  $n \rightarrow \infty$ .

Для узлов Чебышева из здесь сказанного следует, что для липшицевых  $f$  при  $n \rightarrow \infty$  имеет место сходимость с ошибкой  $O(\lg(n)/n)$  на любом фиксированном интервале. Может показаться проблематичным, что  $\lg(n) \rightarrow \infty$  при  $n \rightarrow \infty$ , но это не имеет значения, потому что, если ошибка наилучшего приближения  $\rightarrow 0$ , некоторое ее кратное значение все равно  $\approx 0$ .

## 23.4. Полиномиальная интерполяция из существующих данных

Полиномы равноотдаленных узлов малой степени бывают полезны. К сожалению, знакомое мономиальное базисное представление имеет большое число обусловленности (см. [23.61]), поэтому старайтесь избегать явного вычисления мономиальных коэффициентов. В частности, можно неявно создать и оценить интерполяционный полином на основе степени, увеличенной на 1.

Самый устойчивый способ выполнить такую интерполяцию — использовать *формулу барицентрической интерполяции* (см. [23.61]):

$$p(x) = \frac{\sum y_i w_i / (x - x_i)}{\sum w_i / (x - x_i)}, \text{ где } w_i = \frac{1}{\prod_{j \neq i} (x_i - x_j)}.$$

Равенство  $x = x_i$  приводит к  $p(x) = y_i$ , а проверка на деление на 0 достаточна для обеспечения устойчивости при  $x \approx x_i$ :

```
class BarycentricInterpolation
{
    Vector<pair<double, double> > xy;
    Vector<double> w;
public:
    BarycentricInterpolation(Vector<pair<double, double> >const& thexy)
    { // O(n^2)
        for(int i = 0; i < thexy.getSize(); ++i)
            addPoint(thexy[i].first, thexy[i].second);
    }
}
```

```

void addPoint(double x, double y)
{
    double wProduct = 1;
    for(int i = 0; i < xy.getSize(); ++i)
    {
        wProduct *= (x - xy[i].first);
        w[i] /= xy[i].first - x; // нужно обновить предыдущие значения w[i]
        assert(isfinite(w[i])); // для проверки повторной точки или переполнения
    }
    w.append(1/wProduct);
    assert(isfinite(w.lastItem()));
    xy.append(make_pair(x, y));
}

double operator() (double x) const
{
    assert(isfinite(x));
    double numSum = 0, denomSum = 0;
    for(int i = 0; i < xy.getSize(); ++i)
    {
        double factorI = w[i]/(x - xy[i].first);
        if(!isfinite(factorI)) return xy[i].second; // inf, если x лежит в xy
        numSum += factorI * xy[i].second;
        denomSum += factorI;
    }
    return numSum/denomSum;
}
};

```

У многочленов большая степень может привести к переполнению или потере значимости, но эта общая формула полезна только для полиномов малой степени, где это не проблема.

Можно удалить точки, изменив операцию прибавления:

```

void removePoint(int i)
{
    int n = xy.getSize();
    assert(i >= 0 && i < n);
    for(int j = 0; j < n; ++j)
        if(j != i) w[j] *= (xy[j].first - xy[i].first);
    for(int j = i + 1; j < n; ++j)
    { // универсальный вектор удаляет функции, не являющиеся членами?
        xy[j - 1] = xy[j];
        xy.removeLast();
        w[j - 1] = w[j];
        w.removeLast();
    }
}

```

Вы можете вычислять производные напрямую (см. [23.38]):

♦ в общем случае  $p'(x) = \frac{\sum t_i w_i / (x - x_i)}{\sum w_i / (x - x_i)}$  для  $t_i = \frac{p(x) - y_i}{x - x_i}$ ;



♦ в узле  $i$   $p'(x_i) = \frac{\sum_{j \neq i} (y_j - y_i) w_j / (x_i - x_j)}{w_i}$ . Это следует из предыдущей формулы

$$\text{для случая } x \text{ стремящегося к } x_i, \text{ потому что } p'(x_i) \rightarrow t_i = \frac{\sum (y_j - y_i) w_j / (x - x_j)}{(x - x_i) / \sum w_i / (x - x_j)} \rightarrow$$

$$\rightarrow \frac{\sum_{j \neq i} (y_j - y_i) w_j / (x_i - x_j)}{w_i}.$$

По данным моих экспериментов, общая формула неустойчива и достигает лишь оди-  
нарной точности, вероятно, из-за отмены, как в случае прямой разницы. В работе [23.7]  
предлагается другой подход, но не дается для него явной формулы.

Возможность оценить  $p'(x_i)$  достаточна для выполнения стабильной и эффективной  
оценки: используйте эти значения вместо  $p(x_i)$  в барицентрической формуле для интер-  
поляций — узлы и веса остаются прежними. Можно оценивать эти точки по одной, но  
лучше использовать матрицу дифференцирования  $D$ , элементы которой определяются  
прямым соответствием формулам (см. [23.38]):

♦ для  $j \neq i, D[i, j] = \frac{w_j / (x_i - x_j)}{w_i};$

♦  $D[i, j] = -\sum_{j \neq i} D[i, j].$

Matrix<double> diffMatrix() const

```
{ // дублирование точек невозможно из-за фильтрации весов
    int n = xy.getSize();
    assert(n > 1); // чтобы это имело смысл, нужны хотя бы две точки
    Matrix<double> diff(n, n);
    for(int r = 0; r < n; ++r) for(int c = 0; c < n; ++c) if(r != c)
    {
        diff(r, c) = w[c]/w[r]/(xy[r].first - xy[c].first);
        diff(r, r) -= diff(r, c);
    }
    return diff;
}
```

Vector<double> getY() const

```
{
    int n = xy.getSize();
    Vector<double> y(n);
    for(int i = 0; i < n; ++i) y[i] = xy[i].second;
    return y;
}
```

BarycentricInterpolation overwriteY(Vector<double> const& y) const

```
{
    int n = y.getSize();
    assert(xy.getSize() == n);
    BarycentricInterpolation result = *this;
    for(int i = 0; i < n; ++i) result.xy[i].second = y[i];
    return result;
}
```

```
BarycentricInterpolation deriver() const
{return overwriteY(diffMatrix() * getY());}
```

Производные оценки от интерполяционных полиномов также имеют оценки ошибок. Теорема (см. [23.53]):

$$\|f^{(j)} - p^{(j)}\|_{\infty} \leq \|f^{(n+1)}\|_{\infty} \frac{n! h_{\max}^{n+1-j}}{(j-1)!(n+1-j)!}.$$

Эта теорема (с измененным выражением ошибки с учетом  $(b-a)^{n-j}$ ) в работах по численному анализу встречается редко. По-видимому, впервые она упоминалась в работе [23.39]. Авторы работы [23.35] придерживаются того же мнения.

Когда у функции  $f$  нет  $n+1$  непрерывных производных, должны существовать известные из литературы границы, однако я не смог их найти. Но интерполяция с последующим выводом также является линейным оператором, поэтому можно расширить основанную на аппроксимации теорему для значений функции на  $[-1, 1]$ , используя  $f^{(j)}$  в качестве функции, чтобы получить:

$$\|p^{(j)} - f^{(j)}\| \leq (1 + \|p^{(j)}\|) \frac{(n-k)!(\pi/2)^k}{n!} \|f^{(k-j)}\|_{\infty}.$$

Константа Лебега больше не ограничена, но может применяться *неравенство братьев Марковых*: для  $[-1, 1]$   $\|p''\| \leq n^2 \|p\|_{\infty}$ , поэтому все производные ограничены (см. [23.69]).

Так  $\|p^{(j)}\| \leq (n^2)^j \left( \frac{2h_{\max}}{h_{\min}} \right)^n$ . Для приведенной далее теоремы нужно только, чтобы значение было константой относительно изменения интервала от  $[-1, 1]$  к  $[a, b]$ , что связано с тем, что отношение значений  $h$  не имеет масштаба.

Доказательство теоремы: Если  $p$  интерполирует  $f$  в точке  $\{x_i\}$  и  $f$  имеет  $k$  непрерывных производных  $\in [a = x_0, b = x_n]$  при  $n \geq k$ , то при  $i < k$   $p^{(i)}$  аппроксимирует  $f^{(i)}$  со сходимостью порядка  $k-i$  для  $(b-a) \rightarrow 0$ . Интуитивно ясно, что порядок = минимум (степень формулы, число производных) – число производных.

Для уже заданных или случайных точек постоянная Лебега зависит от барицентрических весовых значений. В этом случае могут быть полезны полиномы низкой степени, но они редко используются из-за возможных произвольных колебаний между узлами. Например, уже для квадратичных  $x_0, x_1, x_2$ , где  $x_0 \approx x_1$ , но  $f(x_0)$  далеко от  $f(x_1)$ , квадратичный экстремум будет колебаться далеко, если  $f$  не является липшицевой.

## 23.5. Полиномы Чебышева

Для полиномиальной интерполяции равноотстоящие точки не нужны. Если минимизировать часть LIE за счет размещения узлов, можно получить корни *полиномов Чебышева*. Эти полиномы определяются на  $[-1, 1]$  рекурсивно как  $T_{k+1}(x) = \begin{cases} 1, & \text{если } k = 0 \\ x, & \text{если } k = 1 \\ 2xT_k(x) - T_{k-1}(x) \end{cases}$

(см. [23.61]) и масштабируются на другие диапазоны. Кроме того,  $T_k(x) = \cos(k \cos^{-1}(x))$ .

Некоторые свойства:

- ♦ корнями  $T_n(x)$  являются  $\cos((j - 1/2)\pi/n)$  для  $1 \leq j \leq k$ ;
- ♦ экстремумы  $T_n(x)$  равны  $\cos(j\pi/n)$  для  $0 \leq j \leq k$ .

Интерполяция в любой из этих точек приводит к аппроксимации  $f(x) = \sum_{0 \leq k \leq n} c_k T_k(x)$ , которая очень точна при достаточном количестве членов:

- ♦ ограничение постоянной Лебега означает, что интерполяция Чебышева будет не так уж плоха, даже если  $f$  не липшицева. И при  $n = 10^6$  максимальная ошибка лишь на порядок хуже, чем у оптимального полинома той же степени. Таким образом, интерполяция Чебышева работает разумно, даже если у функции есть разрывы. Действительно, для  $f =$  единичной ступенчатой функции имеет место явление Гиббса, когда ошибка вблизи разрыва  $\geq$  константа (см. [23.61]), но интерполянт во всех остальных местах имеет большие значения. Таким образом, использование полиномов Чебышева ограниченной степени не приводит к произвольным колебаниям;
- ♦ может показаться, что размещать больше точек рядом с конечными точками расточительно, но это верно для других методов интерполяции, а для полиномов асимптотически это лучший вариант;
- ♦ если поискать (*ограниченные вариации* — см. главу 21. *Вычислительная статистика*), можно найти лучшие, но менее интуитивные оценки, чем формулы, основанные на ограниченных производных (см. [23.61]).

Интерполяция предпочтительнее прямой подгонки многочленов Чебышева к  $f$  с использованием их свойств ортогональности (некоторые примеры приведены в работе [23.48]), что гораздо менее эффективно и точно.

Во многих источниках используются корни (например, здесь [23.51]). Дело в том, что с сингулярностями конечной точки это все еще будет работать, хотя качество работы сомнительно). Но экстремумы позволяют упростить применение БПФ для эффективно-го вычисления  $c_k$  и повторно использовать ранее оцененные точки с удвоением. Оба имеют одинаковые свойства аппроксимации (см. [23.61]).

В частности, (см. [23.45]),  $c_k = \frac{2}{n} \sum_{0 \leq j \leq n} f(y_j) T_k(y_j)$ , где  $y_j = \cos(j\pi/k)$ . Тогда

$f(x) = \sum c_k T_k(x)$ , что реализуется путем деления первого и последнего  $c_k$  пополам при вычислении. Вспоминая определение ДКП типа I с некоторой алгеброй:

$c = \frac{2}{n} DCT(f(y))$  (см. [23.45]). Тогда вычисление выполняется за  $O(n \lg(n))$ . Это помо-

гает с постоянными коэффициентами, когда  $n$  является степенью двойки.

На основании оценки ошибок БПФ после интерполяции можно удалить все старшие коэффициенты  $< \lg(n) \epsilon_{\text{machine}} \|c\|_\infty$ , при этом  $\infty$ -норма больше подходит для отдельных коэффициентов и дает постоянный запас прочности.

Сходимость интерполяции достигается также, когда два старших коэффициента опускаются ниже  $\epsilon_{\text{machine}}$  (см. [23.9]), что в нашем случае фактически является ошибкой округления БПФ, как указано ранее (код оценки является функцией-членом функтора Чебышева). Рассматриваются два коэффициента, а не один, потому что, например, для нечетных функций каждый второй коэффициент равен 0. Хотя этот критерий не явля-

ется полным доказательством, он надежен на практике, потому что входные данные для наихудшего случая, например, с естественными последовательностями нулевых коэффициентов, не возникают:

```
class ChebFunction
{
    Vector<double> ci;
    bool converged;
    double ciAbsE()const{return numeric_limits<double>::epsilon() *
        lgCeiling(ci.getSize()) * (1 + normInf(ci));}
    void trim()
    { // удалить, если значение ci слишком мало
        int oldN = ci.getSize();
        double cutoff = ciAbsE();
        while(ci.getSize() > 1 && abs(ci.lastItem()) < cutoff)
            ci.removeLast();
        if(oldN - ci.getSize() > 1) converged = true;
    }
    // значения f должны быть отсортированы на cos(jPi/n) для 0<=j<=n
    void ChebFunctionHelper(Vector<double> const& fValues)
    { // DCTI работает лучше всего
        ci = DCTI(fValues) * (2.0/(fValues.getSize() - 1));
        ci[0] /= 2; // снижение первого и последнего вдвое
        ci.lastItem() /= 2;
        converged = false;
        trim();
    }
public:
    ChebFunction(Vector<double> const& fValues, int n)
        {ChebFunctionHelper(fValues);}
    template<typename FUNCTION> ChebFunction(FUNCTION const& f, int n)
    {
        assert(n > 0);
        Vector<double> fValues(n + 1);
        for(int i = 0; i <= n; ++i)
            fValues[i] = f(cos(PI() * i/n));
        ChebFunctionHelper(fValues);
    }
    bool hasConverged()const{return converged;}
};
```

Эвристическая оценка ошибки тоже может основываться на этом тесте на сходимость:

```
double error()const{return converged ? ciAbsE() : abs(ci.lastItem());}
```

Метод кажется достаточно точным, если интерполяция сходилась или пыталась сойтись с не слишком малым  $n$ . Имейте в виду, что если для получения нового функтора Чебышева выполнить такие операции, как формальное интегрирование или дифференцирование, эта оценка потеряет смысл.

Можно оценить полином Чебышева, эффективно используя алгоритм Кленшоу (см. [23.51 и 23.45]), который является обобщением метода Хорнера для правильных многочленов:

1.  $d_{n+2} = 0, d_{n+1} = 0$ .
2. Для  $i = n$  до 0,  $d_n = (i == 0 ? 1 : 2) \times d_{i+1} - d_{i-2} + c_i$ .
3. Возврат  $d_0$ .

```
double operator()(double x) const
{
    assert(x >= -1 && x <= 1);
    double d = 0, dd = 0;
    for(int i = ci.getSize() - 1; i >= 0; --i)
    {
        double temp = d;
        d = (i == 0 ? 1 : 2) * x * d - dd + ci[i];
        dd = temp;
    }
    return d;
}
```

В общем случае, если надо решить какую-то задачу с многочленами, нужно выяснить, как это сделать в базисе Чебышева. Может напрямую интегрировать представление в базе Чебышева:  $\int \sum c_k T_k(x) = \sum_{0 \leq k \leq n+1} C_k T_k(x)$ , где для  $k > 0$   $C_k = \frac{c_{k-1} - c_{k+1}}{2k}$ , неявно  $c_{n+1} = c_{n+2} = 0$ , а  $C_0$  определяется из константы интегрирования (см. [23.45]).

Поскольку  $T_0(x) = 1$ ,  $\frac{C_0}{2} = F(-1) - \sum_{1 \leq k \leq n+1} C_k T_k(-1)$ .

```
ChebFunction integral(double FM1 = 0)
{ // особый случай для нулевого полинома
    if(ci.getSize() == 1 && ci.lastItem() == 0) return *this;
    Vector<double> result;
    result.append(FM1);
    for(int i = 1; i - 1 < ci.getSize(); ++i)
        result.append(((i - 1 > 0 ? 1 : 2) * ci[i - 1] -
            (i + 1 > ci.getSize() - 1 ? 0 : ci[i + 1]))/2/i);
    ChebFunction cf(*this);
    cf.ci = result;
    cf.ci[0] -= cf(-1);
    return cf;
}
```

Это позволяет оценить  $\int_{[-1,x]} f(x)$ , но обычно требуется оценивать только  $x = 1$ . В этом случае используется прямая формула:

$$\int_{-1 \leq x \leq 1} \sum_{0 \leq k \leq n} c_k T_k(x) = 2 \sum_{k \text{ even}, 0 \leq k \leq n} \frac{c_k}{1 - k^2} \quad (\text{см. [23.61]}).$$

Чебышевская интерполяция с этой формулой приводит к *квадратуре Кленшоу — Кер-тиса*, которая является одним из лучших алгоритмов интегрирования. Если ошибка интерполяции  $|f - p| \leq \varepsilon$ , это означает, что ошибка интегрирования также ограничена:  $|\int f - \int p| \leq \int |f - p| \leq (x - a) \varepsilon$ , поэтому на этом можно основывать оценку эвристической ошибки:

```
pair<double, double> integrate() const
{
    double result = 0;
    for(int i = 0; i < ci.getSize(); i += 2)
        result += 2 * ci[i]/(1 - i * i);
    return make_pair(result, 2 * error());
}
```

Мои тесты говорят, что эта оценка немного пессимистична, когда интерполяция сходится, что хорошо, и слишком пессимистична, когда это не так, что приемлемо. Ошибка удвоения (обсуждается далее в этой главе) работает аналогично, но она во всех случаях слишком оптимистична. Поскольку для этого также требуется дополнительный код, он далее не рассматривается. В работе [23.45] рекомендуется использовать более сложную оценку.

Теоретически для  $f$  с  $k$  непрерывными производными имеем сходимость порядка  $k$ . Эта оценка пессимистична, но полезна. Немного лучшая оценка с простыми постоянными факторами приведена в работе [23.17].

Вычисление производных работает по аналогичной логике, полученной в результате обращения интегрального процесса:  $c_{k-1} = c_{k+1} + 2kC_k$ , где неявно  $c_{n+1} = c_{n+2} = 0$  (см. [23.45]):

```
ChebFunction derivative() const
{
    int n = ci.getSize() - 1;
    ChebFunction result(*this);
    if(n == 0) result.ci[0] = 0;
    else
    {
        result.ci = Vector<double>(n, 0);
        for(int i = n; i > 0; --i) result.ci[i - 1] =
            (i + 1 > n - 1 ? 0 : result.ci[i + 1]) + 2 * i * ci[i];
        result.ci[0] /= 2;
    }
    return result;
}
```

В отличие от интегрирования, точность производной не следует из точности интерполанта. Например, для шумной функции производная не существует даже при незначительном уровне шума. Нужна сильная корреляция ошибок.

Чтобы найти все корни  $f$ , создайте *матрицу коллег* из полиномиальных коэффициентов, собственные значения которых являются корнями (см. [23.61]):

1. Пусть  $n$  — наибольшее число такое, что  $c_n \neq 0$ .
2. Сформировать матрицу  $C$  размера  $n \times n$  такую, что все элементы равны 0, кроме:
3.  $C[0, 1] = 1$ .
4. Для  $1 \leq r \leq n - 1$   $C[r, r - 1] = C[r, r + 1] = 0.5$ .
5. Для  $0 \leq c < n$  добавить  $-c_c/(2c_n)$  к  $C[n - 1, c]$ .

Настоящие корни суммы многочленов соответствуют корням  $f$  (но не комплексным):

```
Vector<double> findAllRealRoots(double complexAbsE = highPrecEps) const
{
    int n = ci.getSize() - 1;
    if(n == 0) return Vector<double>(1, 0); // если все нули
    else if(n == 1) return Vector<double>(); // нет корней полинома
    // создание матрицы коллег
    Matrix<double> colleague(n, n);
    colleague(0, 1) = 1;
    for(int r = 1; r < n; ++r)
    {
        colleague(r, r - 1) = 0.5;
        if(r + 1 < n) colleague(r, r + 1) = 0.5;
        if(r == n - 1) for(int c = 0; c < n; ++c)
            colleague(r, c) -= ci[c]/(2 * ci[n]);
    } // решение и сохранение вещественных корней
    Vector<complex<double> > croots = QREigenHessenberg(colleague);
    Vector<double> result; // удаление комплексных и экстраполированных корней
    for(int i = 0; i < croots.getSize(); ++i) if(abs(croots[i].imag()) <
        complexAbsE && -1 <= croots[i].real() && croots[i].real() <= 1)
        result.append(croots[i].real());
    return result;
}
```

В работе [23.9] рекомендуется сбалансировать матрицу (см. [23.25]), но здесь мы этого не делали. В моих тестах с  $f(x) = \sin(x)$  на разных диапазонах корни, как правило, не очень точны, и появляются некоторые искусственные корни, но с ними легко справиться с помощью метода хорд (обсуждается далее в этой главе).

Чтобы решить, сколько точек использовать для получения достаточной точности, выполняйте удвоение, пока решение не сойдется:

```
template<typename FUNCTION>
Vector<double> reuseChebEvalPoints(FUNCTION const& f, Vector<double> const& fx)
{
    int n = 2 * (fx.getSize() - 1);
    assert(isPowerOfTwo(n));
    Vector<double> result;
    for(int i = 0; i <= n; ++i)
        result.append(i % 2 ? f(cos(PI() * i/n)) : fx[i/2]);
    return result;
}

template<typename FUNCTION> ChebFunction adaptiveChebEstimate(
    FUNCTION const& f, int maxEvals = 10000, int minEvals = 17)
{
    int n = minEvals - 1;
    assert(minEvals <= maxEvals && isPowerOfTwo(n));
    Vector<double> fx(n + 1);
    for(int i = 0; i <= n; ++i) fx[i] = f(cos(PI() * i/n));
    ChebFunction che(fx, n);
    while(maxEvals >= fx.getSize() + n && !che.hasConverged())
    {
        fx = reuseChebEvalPoints(f, fx);
    }
}
```

```

    che = ChebFunction(fx, n *= 2);
}
return che;
}

```

Разрешите масштабирование на произвольный диапазон  $[a, b]$ . После этого можно интерполировать  $f$ , как если бы она была определена на  $[-1, 1]$ , и оценить значение интерполянта и другие свойства:

- ◆ интерполяция — использовать значения  $f(x)$ , где  $x = u$  масштабируется от  $[-1, 1]$  до  $[a, b]$ ;
- ◆ оценка — возвращает  $\text{interpolant}(u)$  для  $u = x$  в масштабе от  $[a, b]$  до  $[-1, 1]$ ;
- ◆ интеграция — умножить результат на  $(b - a) / 2$ ;
- ◆ производная — разделить результат на  $(b - a) / 2$  (интуитивно это должно согласовываться с конечной разностной оценкой);
- ◆ корни — измените масштаб каждого из  $[-1, 1]$  на  $[a, b]$ .

```

static double xToU(double x, double a, double b)
{
    assert(a <= x && x <= b);
    double u = (2 * x - a - b) / (b - a);
    return u;
}

static double uToX(double u, double a, double b)
{
    assert(-1 <= u && u <= 1);
    return ((b - a) * u + a + b) / 2;
}

template<typename FUNCTION> class ScaledFunctionM11
{ // чтобы функции с любыми диапазонами приводились к [-1;1]
    FUNCTION f;
    double a, b;
public:
    ScaledFunctionM11(double theA, double theB, FUNCTION const& theF =
        FUNCTION()): f(theF), a(theA), b(theB) {assert(a < b);}
    double operator()(double u) const {return f(ChebFunction::uToX(u, a, b));}
};

struct ScaledChebAB
{ // для оценки функции Чебышева на любом диапазоне
    ChebFunction f;
    double a, b;
public:
    template<typename FUNCTION> ScaledChebAB(FUNCTION const& theF, int n,
        double theA, double theB): a(theA), b(theB),
        f(ScaledFunctionM11<FUNCTION>(theA, theB, theF), n) {assert(a < b);}
    ScaledChebAB(Vector<double> const& fValues, double theA, double theB):
        f(fValues, 0), a(theA), b(theB) {assert(a < b);}
    ScaledChebAB(ChebFunction const& theF, double theA, double theB):
        f(theF), a(theA), b(theB) {assert(a < b);}
    double operator()(double x) const {return f(ChebFunction::xToU(x, a, b));}
}

```



```

pair<double, double> integrate()const
{
    pair<double, double> result = f.integrate();
    result.first *= (b - a)/2;
    result.second *= (b - a)/2;
    return result;
}

double evalDeriv(double x)const
{
    return 2/(b - a) * f.derivative()(ChebFunction::xToU(x, a, b));
}

Vector<double> findAllRealRoots()const
{
    // значения по умолчанию вполне подходят
    Vector<double> roots = f.findAllRealRoots();
    for(int i = 0; i < roots.getSize(); ++i)
        roots[i] = ChebFunction::uToX(roots[i], a, b);
    return roots;
}
};

```

## 23.6. Кусочная интерполяция

*Линейная интерполяция* создает линию между двумя ближайшими точками, охватывающими входные  $x$ , и вычисляет значение  $y$ , как будто оно находится на этой линии (рис. 23.1).

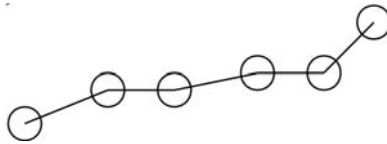


Рис. 23.1. Набор точек, соединенных линиями

Если  $f$  является липшицевой, линейная интерполяция сходится при  $\max \Delta x \rightarrow 0$ , потому что  $|f - LL| \leq O(\max \Delta x)$ . Согласно LIE при достаточном количестве производных ошибка в каждой части составляет  $O(\Delta x^2)$ . Несмотря на отсутствие плавности, линейная интерполяция часто является первым методом, используемым для заданных точек данных, потому что интерполированные значения функции всегда находятся внутри диапазона концов используемого сегмента, даже если функция  $f$  не является непрерывной.

Реализация обрабатывает точки динамически, используя динамически отсортированную последовательность. Чтобы интерполировать в точке  $x$ , найдите ее предшественника и преемника и используйте их значения. Это занимает время  $O(\lg(n))$ .

Из-за значений NaN и возможности сравнения значения в регистре со значениями в структуре данных и последующего округления перед сохранением возможен случай, когда сохраняются равные при округлении значения. Таким образом, технически декартово дерево не определяется с ключом `double` и требует особого внимания. Отношение «<» не сохраняется при округлении (а «≤» сохраняется).

Следует определять диапазон и проверять значения, используя некоторые  $\varepsilon$ . Это также предотвращает дублирование точек, что часто бывает полезно. Операция включающего

предшественника, в которой используется сравнение « $\leq$ », дает правильный результат (то же самое для включающего преемника) в том смысле, что возвращаемый диапазон гарантированно содержит  $x$ , пока он находится внутри конечных точек и  $\neq \text{NaN}$ :

```
template<typename DATA> class PiecewiseData
{
    typedef Treap<double, DATA> POINTS;
    mutable POINTS values;
    double eRelAbs;
public:
    int getSize() const { return values.getSize(); }
    PiecewiseData(double theERelAbs = numeric_limits<double>::epsilon()):
        eRelAbs(theERelAbs) {}
    typedef typename POINTS::NodeType NODE;
    pair<NODE*, NODE*> findPiece(double x) const
    {
        assert(!isnan(x));
        NODE* left = values.inclusivePredecessor(x), *right = 0;
        if(!left) right = values.findMin();
        else
        {
            typename POINTS::Iterator i(left);
            ++i;
            right = i != values.end() ? &*i : 0;
        }
        return make_pair(left, right);
    }
    NODE* eFind(double x) const
    {
        assert(!isnan(x));
        pair<NODE*, NODE*> piece = findPiece(x);
        if(piece.first && isEEqual(x, piece.first->key, eRelAbs))
            return piece.first;
        if(piece.second && isEEqual(x, piece.second->key, eRelAbs))
            return piece.second;
        return 0;
    }
    NODE* findMin() const
    {
        assert(values.getSize() > 0);
        return values.findMin();
    }
    NODE* findMax() const
    {
        assert(values.getSize() > 0);
        return values.findMax();
    }
    bool isInERange(double x) const
    {
        return getSize() > 1 && findMin()->key <= x && x <= findMax()->key;
    }
    void insert(double x, DATA const& y)
    {
        NODE* node = eFind(x);
```

```

        if(node) node->value = y;
        else values.insert(x, y);
    }
    void eRemove(double x)
    {
        NODE* node = eFind(x);
        if(node) values.removeFound(node);
    }
    Vector<pair<double, DATA> > getPieces() const
    {
        Vector<pair<double, DATA> > result;
        for(typename POINTS::Iterator iter = values.begin();
            iter != values.end(); ++iter)
            result.append(make_pair(iter->key, iter->value));
        return result;
    }
};

```

Реализуем линейную интерполяцию, используя приведенное только что:

```

class DynamicLinearInterpolation
{
    PiecewiseData<double> pd;
public:
    DynamicLinearInterpolation(){}
    DynamicLinearInterpolation(Vector<pair<double, double> > const& xy)
    { // фильтрация точек для достижения точности emachine для расстояния x
        assert(isSorted(xy.getArray(), 0, xy.getSize() - 1,
            PairFirstComparator<double, double>()));
        for(int i = 0, lastGood = 0; i < xy.getSize(); ++i)
            if(i == 0 || isELess(xy[lastGood].first, xy[i].first))
            {
                insert(xy[i].first, xy[i].second);
                lastGood = i;
            }
    }
    double operator() (double x) const
    {
        if(!pd.isInERange(x)) return numeric_limits<double>::quiet_NaN();
        typedef typename PiecewiseData<double>::NODE NODE;
        pair<NODE*, NODE*> segment = pd.findPiece(x);
        assert(segment.first && segment.second); // проверка разумности значения
        double ly = segment.first->value, lx = segment.first->key;
        return ly + (segment.second->value - ly) * (x - lx) /
            (segment.second->key - lx);
    }
    void eRemove(double x) {pd.eRemove(x);}
    bool eContains(double x) {return pd.eFind(x) != 0;}
    void insert(double x, double y) {pd.insert(x, y);}
};

```

Операция удаления не распространена, но полезна, если оцененные точки устаревают.

Следующий шаг — кусочные полиномы более высокой степени. Для функции  $f$  с достаточным числом производных ошибка аппроксимации уменьшается. Чтобы использовать кусочные полиномы степени  $d$ , разделите  $n = dm + 1$  точек на соединенные сегменты по  $d$  точек в каждом при условии, что левая точка данных сегмента содержит полином. Настройка выполняется за  $O(nd)$ , а оценка —  $O(\lg(n/d) + d)$ .

Поскольку полиномы Чебышева стабильнее, используйте их в основе кусочной интерполяции:

```
template<typename INTERPOLANT> class GenericPiecewiseInterpolation
{
    PiecewiseData<INTERPOLANT> pd;
public:
    GenericPiecewiseInterpolation(
        PiecewiseData<INTERPOLANT> const& thePd): pd(thePd){}
    double operator() (double x) const
    {
        if(!pd.isInRange(x)) return numeric_limits<double>::quiet_NaN();
        typename PiecewiseData<INTERPOLANT>::NODE* segment =
            pd.findPiece(x).first;
        assert(segment); // проверка разумности
        return segment->value(x);
    }
    Vector<pair<pair<double, double>, INTERPOLANT> > getPieces() const
    {
        Vector<pair<double, INTERPOLANT> > pieces = pd.getPieces();
        assert(pieces.getSize() > 1); // одно реальное значение, одно подложное
        Vector<pair<pair<double, double>, INTERPOLANT> > result;
        for(int i = 0; i < pieces.getSize(); ++i)
        {
            if(result.getSize() > 0) // установка правой границы предыдущего сегмента
                result.lastItem().first.second = pieces[i].first;
            result.append(make_pair(make_pair(pieces[i].first, 0),
                pieces[i].second));
        }
        result.removeLast(); // последний сегмент подложный
        return result;
    }
    GenericPiecewiseInterpolation<INTERPOLANT> derivier() const
    {
        Vector<pair<double, INTERPOLANT> > pieces = pd.getPieces();
        PiecewiseData<INTERPOLANT> result;
        for(int i = 0; i < pieces.getSize(); ++i) // последний подложный
            result.insert(pieces[i].first, i < pieces.getSize() - 1 ?
                pieces[i].second.deriver() : pieces[i].second);
        return GenericPiecewiseInterpolation<INTERPOLANT>(result);
    }
    double integrate() const
    {
        double sum = 0;
        Vector<pair<pair<double, double>, INTERPOLANT> > pieces = getPieces();
```

```

    for(int i = 0; i < pieces.getSize(); ++i)
        sum += pieces[i].second.integrate();
    return sum;
}
};

```

Трудно решить, сколько точек использовать, поэтому можно применить глобальную адаптивную стратегию, как и для адаптивной интеграции (обсуждается далее в этой главе). Для оценки ошибки на интервале задействуйте максимальную ошибку среди новых точек, которые будут использоваться для разделения интервала. Идея проста, хотя код немного сложен:

```

template<typename ADAPTIVE_INTERVAL> struct AdaptiveIntervalComparator
{
    double deltaLength;
    bool operator()(ADAPTIVE_INTERVAL const& lhs,
        ADAPTIVE_INTERVAL const& rhs) const
    {
        return (lhs.length() > deltaLength || rhs.length() > deltaLength) ?
            lhs.length() > rhs.length() : lhs.error() > rhs.error();
    }
};

template<typename INTERPOLATION_INTERVAL, typename FUNCTION>
pair<GenericPiecewiseInterpolation<
    typename INTERPOLATION_INTERVAL::INTERPOLANT>,
    double> interpolateAdaptiveHeap(FUNCTION const& f, double a, double b,
    double param, double eRelAbs = highPrecEps, int maxEvals = 1000000,
    int minEvals = -1)
{
    typedef INTERPOLATION_INTERVAL II;
    typedef typename II::INTERPOLANT I;
    if(minEvals == -1) minEvals = sqrt(maxEvals);
    assert(a < b && maxEvals >= minEvals && minEvals >= II::initEvals(param));
    INTERPOLATION_INTERVAL i0(f, a, b, param);
    double scale = i0.scaleEstimate();
    AdaptiveIntervalComparator<II> ic = {(b - a)/minEvals};
    Heap<II, AdaptiveIntervalComparator<II> > h(ic);
    h.insert(i0);
    for(int usedEvals = II::initEvals(param);
        usedEvals < minEvals || (
            usedEvals + II::splitEvals(param) <= maxEvals &&
            !isEEqual(scale, scale + h.getMin().error(), eRelAbs));
        usedEvals += II::splitEvals(param))
    {
        II next = h.deleteMin();
        Vector<II> division = next.split(f);
        for(int i = 0; i < division.getSize(); ++i) h.insert(division[i]);
    } // обработка интервалов кучи
    PiecewiseData<I> pd;
    double error = h.getMin().error(),
        right = -numeric_limits<double>::infinity();

```

```

while(!h.isEmpty())
{
    II next = h.deleteMin();
    Vector<pair<double, I> > interpolants = next.getInterpolants();
    for(int i = 0; i < interpolants.getSize(); ++i)
        pd.insert(interpolants[i].first, interpolants[i].second);
} // нужен интерполятор с подложными правыми конечными точками
pd.insert(b, i0.getInterpolants().lastItem().second);
return make_pair(GenericPiecewiseInterpolation<I>(pd), error);
}

```

Вам может также потребоваться защита от слишком маленьких подынтервалов, хотя в моих тестах такой необходимости не возникало.

Для произведений Чебышева, если удвоение не приводит к предполагаемой сходимости при достижении некоторого порога  $k$ , например 64, нужно рекурсивно разделить заданный интервал на два и повторить процедуру, хотя и без возможности повторного использования точек. Когда  $f$  не липшицева или сходимость удвоения линейна, этот метод имеет много преимуществ:

- ◆ постоянная Лебега ограничена;
- ◆ вычисления более эффективны — на вычисление требуется время  $O(k \lg(n/k))$ ;
- ◆ на некоторых частях  $f$  может быть липшицевой или допускать сходимость высокого порядка, и на них не будут влиять другие части. Таким образом, ошибка  $L_2$  должна быть лучше.

Но, как и в случае с правильными кусочными полиномами, производные не являются непрерывными. В отличие от рекурсивной реализации, глобальное подразделение кучи обеспечивает лучший контроль:

```

class IntervalCheb
{
    ScaledChebAB cf;
    pair<double, double> ab;
    int maxEvals;
public:
    typedef ScaledChebAB INTERPOLANT;
    template<typename FUNCTION> IntervalCheb(FUNCTION const& f, double a,
        double b, int theMaxEvals = 64): ab(a, b), cf(f, theMaxEvals, a, b),
        maxEvals(theMaxEvals){}
    Vector<pair<double, ScaledChebAB> > getInterpolants() const
    {
        return Vector<pair<double, ScaledChebAB> >(1, make_pair(ab.first, cf));
    }
    double scaleEstimate() const {return 1;}
    double length() const {return 0;}
    double error() const // сначала большие значения
    {return cf.f.hasConverged() ? 0 : ab.second - ab.first;}
    template<typename FUNCTION>
    Vector<IntervalCheb> split(FUNCTION const& f) const
    {
        Vector<IntervalCheb> result;
        double middle = (ab.first + ab.second)/2;
        result.append(IntervalCheb(f, ab.first, middle, maxEvals));
    }
}

```

```

    result.append(IntervalCheb(f, middle, ab.second, maxEvals));
    return result;
}
static int initEvals(int maxEvals){return maxEvals;}
static int splitEvals(int maxEvals){return 2 * maxEvals;}
};

```

## 23.7. Сплайны — для работы с существующими данными

Проблема кусочных многочленов заключается в том, что производные не являются непрерывными в узлах, и в некоторых приложениях требуется гладкость первой и, возможно, второй производной. Также неудобно подгонять к существующим данным что-либо, кроме линейной кусочной интерполяции. *Кубические сплайны* гарантируют непрерывные производные, обеспечивая для сплайна  $s$  непрерывность  $s'$  и  $s''$  в точках интерполяции. Пусть  $M_i = s''(x_i)$ . Тогда для  $x \in [x_{i-1}, x_i]$   $s''(x) = \frac{M_{i-1}(x_i - x) + M_i(x - x_{i-1})}{h_i}$ ,

где  $h_i = x_i - x_{i-1}$ . После двух интегрирований получаем  $s_{i-1}(x) = \frac{M_{i-1}(x_i - x) + M_i(x - x_{i-1})}{6h_i} + b_{i-1}(x - x_{i-1}) + c_{i-1}$ . Сопоставление значений конечных

точек сегментов дает  $b_{i-1} = f_{i-1} - M_{i-1} \frac{h_i^2}{6}$  и  $c_{i-1} = \Delta f_i - (M_i - M_{i-1}) \frac{h_i}{6}$ , где  $\Delta f_i = \frac{f_i - f_{i-1}}{h_i}$ .

Обеспечение непрерывности  $f'$  приводит к системе уравнений  $\mu_i M_{i-1} + 2M_i + \lambda_i M_{i+1} = d_i$  для  $1 \leq i \leq n-2$ , где  $\mu_i = \frac{h_i}{h_i + h_{i+1}}$ ,  $\lambda_i = \frac{h_{i+1}}{h_i + h_{i+1}}$  и  $d_i = \frac{6(\Delta f_{i+1} - \Delta f_i)}{h_i + h_{i+1}}$ . Нужны граничные условия вида  $2M_i + \lambda_i M_{i+1} = d_i$  для  $i = 0$  и  $\mu_i M_{i-1} + 2M_i = d_i$  для  $i = n-1$ .

Самая простая альтернатива — это *естественный кубический сплайн*, который становится линией за пределами диапазона данных (см. [23.20]), определяемой как  $2M_0 = 2M_{n-1} = 0$ . Его точность на компактном интервале внутри конечных точек составляет  $O(h^4)$  при достаточном количестве непрерывных производных и  $h \rightarrow 0$  (см. [23.14]). Но предельная точность вблизи конечных точек составляет всего  $O(h^2)$ , поэтому лучше использовать *сплайн без узлов*, который поддерживает точность  $O(h^4)$  внутри конечных точек (см. [23.5 и 23.14]). Точность неузловых точек и некоторых других сплайнов трудно проанализировать, а некоторые основные результаты не были известны до работы [23.4]).

Идея неузловых точек заключается в том, чтобы обеспечить непрерывность  $s^{(3)}$  в точках  $x_1$  и  $x_{n-2}$ , т. е. первые два и два последних сегмента становятся одним и тем же многочленом, «отбрасывая» эти точки и используя их только для граничных условий. Это не меняет  $h$  в конечных точках границ, потому что  $s$  по-прежнему проходит через эти точки. Дифференцирование  $s''(x)$ , дает:

$$s^{(3)}(x_1) = \frac{M_1 - M_0}{h_1} = \frac{M_2 - M_1}{h_2},$$

откуда  $2h_2M_0 - 2(h_1 + h_2)M_1 + 2h_1M_2 = 0$ . При подстановке значений  $\mu_1$  и  $\lambda_1$  в неграничные уравнения, получим

$$h_1M_0 + 2(h_1 + h_2)M_1 + h_2M_2 = d_1(h_1 + h_2).$$

Это позволяет исключить  $M_1$ , так что в системе без узлов получаем граничные условия:

$$2M_0 + 2M_2 \frac{2h_1 + h_2}{2h_2 + h_1} = 2d_1 \frac{h_1 + h_2}{2h_2 + h_1}.$$

Хотя в  $M_1$  недостает  $x_1$ , это значение используется для вычисления  $h_1$ ,  $h_2$  и  $d_1$ .

Аналогично

$$s^{(3)}(x_{n-2}) = \frac{M_{n-2} - M_{n-3}}{h_{n-2}} = \frac{M_{n-1} - M_{n-2}}{h_{n-1}},$$

что дает

$$2h_{n-1}M_{n-3} - 2(h_{n-2} + h_{n-1})M_{n-2} + 2h_{n-2}M_{n-1} = 0.$$

При подстановке значений  $\mu_{n-2}$  и  $\lambda_{n-2}$  в неграничные уравнения получим

$$h_{n-2}M_{n-3} + 2(h_{n-2} + h_{n-1})M_{n-2} + h_{n-1}M_{n-1} = d_{n-2}(h_{n-2} + h_{n-1}).$$

После исключения  $M_{n-2}$  получим

$$2M_{n-3} \frac{2h_{n-1} + h_{n-2}}{2h_{n-2} + h_{n-1}} + 2M_{n-1} = 2d_{n-2} \frac{h_{n-2} + h_{n-1}}{2h_{n-2} + h_{n-1}}.$$

Результирующая трехдиагональная система не доминирует по диагонали из-за первой и последней строк, но алгоритм Гаусса — Томаса все еще работает. Наихудший случай для первого граничного уравнения имеет место, когда  $\frac{h_2}{h_1} \rightarrow 0$ , что дает

$2M_0 + 4M_2 = R_0$ . Тогда, поскольку  $h_1 + h_2 \rightarrow h_1$ , второе ограничивающее уравнение системы без  $M_1$  становится равным

$$M_0 \frac{h_1}{h_1 + h_3} + 2M_2 + M_3 \frac{h_3}{h_1 + h_3} = R_1.$$

Пока  $\frac{h_3}{h_1} \rightarrow 0$ , и учитывая, что  $\frac{h_1}{h_1 + h_3} \approx 1 - \frac{h_3}{h_1}$ , после устранения  $M_0$  путем вычитания

граничного уравнения  $\times \frac{1}{2} \left( 1 - \frac{h_3}{h_1} \right)$ , второе уравнение равно

$$2 \frac{h_3}{h_1} M_2 + \frac{h_3}{h_1} M_3 = R_1 - \left( 1 - \frac{h_3}{h_1} \right) \frac{R_0}{2},$$

а это диагональная доминанта.

Симметрично, если бы процесс исключения был выполнен в обратном порядке, уравнение, предшествующее последнему, осталось бы диагонально доминирующим. Таким образом, когда обычное исключение, при котором все уравнения, кроме последнего, остаются доминантными по диагонали, достигает последнего уравнения, уравнение, предшествующее последнему, сохраняет достаточное диагональное доминирование



(и обычно получает больше), так что отмена не произойдет. Чтобы этот процесс заработал, необходимо хотя бы одно неграничное уравнение, что имеет место при  $n = 5$ . Решение состоит из двух сплайнов, что является минимумом, учитывая, что один сплайн невозможен при не-узле:

```
class NotAKnotCubicSplineInterpolation
{ // книга M равна a, книга d равна R
    struct Data{double a, b, c;};
    PiecewiseData<Data> pd;
public:
    NotAKnotCubicSplineInterpolation(Vector<pair<double, double> > xy,
        double eRelAbs = numeric_limits<double>::epsilon()): pd(eRelAbs)
    { // фильтрация точек, чтобы обеспечить расстояние eps x
        int n = xy.getSize(), skip = 0;
        assert(isSorted(xy.getArray(), 0, n - 1,
            PairFirstComparator<double, double>()));
        for(int i = 1; i + skip < n; ++i)
        {
            if(!isELess(xy[i - 1].first, xy[i + skip].first, eRelAbs)) ++skip;
            if(i + skip < n) xy[i] = xy[i + skip];
        }
        while(skip--) xy.removeLast();
        n = xy.getSize();
        assert(n > 3); // нужны 4 точки, чтобы сформировать кубический сплайн
        // особая логика для конечных точек
        double h1 = xy[1].first - xy[0].first, h2 = xy[2].first - xy[1].first,
            t1 = (xy[1].second - xy[0].second)/h1,
            t2 = (xy[2].second - xy[1].second)/h2,
            hnm2 = xy[n - 2].first - xy[n - 3].first,
            hnm1 = xy[n - 1].first - xy[n - 2].first,
            tnm2 = (xy[n - 2].second - xy[n - 3].second)/hnm2,
            tnm1 = (xy[n - 1].second - xy[n - 2].second)/hnm1;
        // извлечение точек 1 и n - 2
        for(int i = 1; i < n - 3; ++i) xy[i] = xy[i + 1];
        xy[n - 3] = xy[n - 1];
        xy.removeLast();
        xy.removeLast();
        n = xy.getSize();
        // настройка и решение тридиагональной системы
        TridiagonalMatrix<double> T(
            2 * TridiagonalMatrix<double>::identity(n));
        Vector<double> R(n);
        // граничные условия
        double D0Factor = 2/(2 * h2 + h1), Dnm1Factor = 2/(2 * hnm2 + hnm1);
        T(0, 1) = (2 * h1 + h2) * D0Factor;
        R[0] = 6 * (t2 - t1) * D0Factor;
        T(n - 1, n - 2) = (2 * hnm1 + hnm2) * Dnm1Factor;
        R[n - 1] = 6 * (tnm1 - tnm2) * Dnm1Factor;
        for(int i = 1; i < n - 1; ++i)
        {
            double hk = xy[i].first - xy[i - 1].first,
                tk = (xy[i].second - xy[i - 1].second)/hk,
```

```

        hkpl = xy[i + 1].first - xy[i].first, hSum = hk + hkpl,
        tkpl = (xy[i + 1].second - xy[i].second)/hkpl;
    R[i] = 6 * (tkpl - tk)/hSum;
    T(i, i + 1) = hkpl/hSum;
    T(i, i - 1) = hk/hSum;
}
Vector<double> a = solveTridiag(T, R);
// вычисление b и c
for(int i = 1; i < n + 1; ++i)
{
    double bi = 0, ci = 0;
    if(i < n)
    {
        double hi = xy[i].first - xy[i - 1].first;
        bi = (xy[i].second - xy[i - 1].second)/hi -
            (a[i] - a[i - 1]) * hi/6;
        ci = xy[i - 1].second - a[i - 1] * hi * hi/6;
    }
    Data datai = {a[i - 1], bi, ci};
    pd.insert(xy[i - 1].first, datai);
}
}

double operator()(double x, int deriv = 0) const
{
    assert(deriv >= 0 && deriv <= 2); // наличие двух непрерывных производных
    if(!pd.isInERange(x)) return numeric_limits<double>::quiet_NaN();
    typedef typename PiecewiseData<Data>::NODE NODE;
    pair<NODE*, NODE*> segment = pd.findPiece(x);
    assert(segment.first && segment.second); // проверка разумности значения
    double dxl = x - segment.first->key, dxr = segment.second->key - x,
        aiml = segment.first->value.a, ai = segment.second->value.a,
        bi = segment.first->value.b, ci = segment.first->value.c,
        hi = dxr + dxl;
    if(deriv == 2) return (aiml * dxr + ai * dxl)/hi;
    if(deriv == 1) return (-aiml * dxr * dxr + ai * dxl * dxl)/(hi * 2) + bi;

    return (aiml * dxr * dxr * dxr + ai * dxl * dxl * dxl)/(hi * 6) + bi * dxl + ci;
}
};

```

Зачем использовать кубические сплайны, а не квадратичные или более высокого порядка? Несколько причин использовать именно их как компромисс:

- ◆ вычисление  $O(n)$ ;
- ◆ с двумя непрерывными производными получается интерпретация: и скорость, и ускорение непрерывны, поэтому резких движений нет;
- ◆ квадратичные сплайны имеют только одну непрерывную производную, а этого недостаточно для некоторых приложений;
- ◆ высокая степень гладкости может быть необходима только в редких случаях, специфичных для конкретных приложений. Но вывод становится утомительным, результирующая матрица больше не является тридиагональной (но в некоторых

случаях малодиагональной), и по соображениям стабильности вычисления обычно выполняются с использованием *B-сплайнов*, которые здесь не обсуждаются (см. раздел комментариев).

Трудно сравнивать ошибки кусочно-кубических и кубических сплайнов. Оба имеют ошибку  $O(h^4)$  при достаточно дифференцируемом  $f$ , и мои простые эксперименты с несколькими функциями показывают, что одна из них может превосходить другую на небольшой постоянный коэффициент. *Полный кубический сплайн*, предполагающий знание  $f$  в конечных точках, имеет лучшие постоянные коэффициенты ошибки по сравнению с кусочно-кубическим сплайном, а не-узловые точки аппроксимируются с крошечной ошибкой (см. [23.14]), поэтому он вроде чуть лучше. В работе [23.4] даются границы погрешности для случаев, когда  $f$  имеет мало непрерывных производных, однако константы трудно сравнить с кусочными. Но поскольку кусочные полиномы обладают известными простыми аналитическими свойствами, в большинстве случаев их и используют.

Для выбранных точек сплайны не имеют смысла. Выбрать значение  $n$  трудно, поэтому можно просто удваивать его (рис. 23.2).

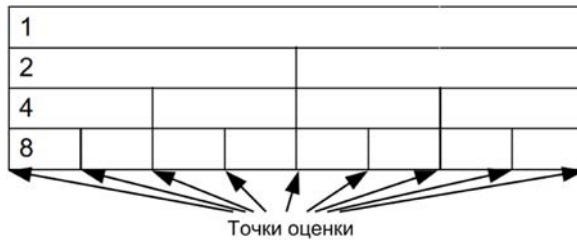


Рис. 23.2. Повторное использование оцениваемых точек с удвоением количества интервалов

Проблема в том, что другие методы гораздо лучше справляются с выбором нужных точек для адаптивного получения требуемой точности.

Для сплайнов константа Лебега у равноотстоящих точек постоянна (см. [23.62]). Сплайн-интерполяция превосходит линейную интерполяцию для уже заданных данных и сглаженной  $f$ , но используется редко, особенно потому, что интерполяция редко является целью сама по себе.

## 23.8. Сравнение методов интерполяции

Для кусочных многочленов фиксированной степени, как обсуждалось ранее, постоянная Лебега ограничена константой, которая зависит от длины участков. В частности, она также ограничена константой для равностоящих точек.

Но в точке разрыва ничего не помогает. Никакие аппроксимации не дают нулевую ошибку — они сходятся только в близких точках. Учет цели интерполяции упрощает принятие решений. Некоторые типичные варианты использования:

1. Требуется оценить  $f(x)$  эффективнее, чем методом прямой оценки. Если  $f$  — элементарная функция, такая как  $\cos$  или любая другая, непрерывная со многими производными, интерполяция Чебышева некоторой фиксированной степени, выбранной экспериментально, работает. Но аппроксимация элементарных функций — это са-

мостоятельная область со многими полезными методами, которые будут обсуждаться далее в этой главе. Обычно ее невозможно выполнить для  $f$ -«черного ящика», а в случае конкретной  $f$  нужно тщательно рассмотреть ее свойства. Излишне говорить, что не стоит пытаться оптимизировать библиотечную функцию, такую как  $e^x$ , т. к. вы потеряете надежность.

2. Аналогично случаю (1), но  $f$  является результатом дорогостоящего эксперимента. Истинная выходная функция, скорее всего, не будет непрерывной. Линейная интерполяция — это опция по умолчанию, но можно использовать кусочно-квадратичную или кубическую, если вы ожидаете некоторой плавности. Если ожидать шума, регрессия окажется более приблизительной (см. главу 27. *Машинное обучение: регрессия*).
3. Если нужно интерполировать из таблицы значений, используйте линейную интерполяцию из-за ограниченной точности табличных значений. Но вместо этого в большинстве случаев специальные библиотечные функции могут достаточно дешево и точно вычислить требуемое значение.
4. Смоделируйте такую задачу как интегрирование, где полезны аналитические свойства полиномиальных частей. Интерполяция в этом случае будет использовать зависимости от задачи эвристики, такие как оценки ошибок и адаптивные стратегии. Например, все интерполянты плохо справляются со ступенчатой функцией в отношении ошибки максимальной нормы и числа вычислений функции (рис. 23.3).

Функция	Cheb адаптивная		Cheb61 адаптивная		DynLin	
	Ошибка	Оценка	Ошибка	Оценка	Ошибка	Оценка
Ступенчатая	0.00	4097	-15.65	6175	-0.13	1000
Модуль	-3.84	4097	-15.65	1105	-2.96	1000
Линейная	-15.65	17	-15.65	1105	-15.65	1000
Квадратичная	-15.65	17	-15.65	1105	-4.22	1000
Кубическая	-15.65	17	-15.65	1105	-4.11	1000
4-й степени	-15.65	17	-15.65	1105	-3.74	1000
Экспонента	-14.77	17	-15.17	1105	-4.52	1000
SqrtAbs	-1.94	4097	-14.59	12155	-1.86	1000
Рунге	-13.40	1025	-14.44	1105	-2.47	1000
Логарифм	-15.30	33	-14.68	1105	-5.37	1000
XSinXM1	-1.60	4097	-5.32	1015625	-1.23	1000
Синус	-15.65	17	-14.87	1105	-4.74	1000
AbsIntegral	-7.65	4097	-15.65	1105	-4.60	1000
Тангенс	-15.00	33	-15.15	1105	-4.84	1000
Нормальная PDF	-14.83	129	-14.76	1105	-3.19	1000
Дельта PDF	-2.54	4097	-15.65	10595	-1.33	1000
F575	0.83	4097	-14.67	6825	0.00	1000
Средние ранги	1.7	1.6	1.2	2.9	2.8	1.5

**Рис. 23.3.** Эксперименты на некоторых  $f$ . Ошибки даны в десятичных разрядах относительной абсолютной ошибки по отношению к известному ответу. То же самое для предполагаемых ошибок, где это применимо. Для оценки максимальной ошибки использовалось  $10^5$  равномерно случайных точек в диапазоне функции (указанном в тестовом файле). Во всех случаях все интерполянты плохо справляются с  $f$  со скачками, такими как ступенчатая функция, поэтому такие случаи далее обсуждаться не будут

Некоторые грубые общие закономерности:

- ◆ для адаптивной интерполяции:
  - когда функция  $f$  имеет много непрерывных производных, удвоение интерполяции Чебышева дает значительный выигрыш и часто оценивает  $f$  с машинной точностью менее чем за 100 оценок (хотя для функции Рунге требуется  $\approx 1000$ ). Но если  $f$  недифференцируема, она исчерпывает предел вычислений и, по-видимому, дает очень грубую ошибку  $O(1/n)$  для  $n$  вычислений;
  - использование кусочной функции Чебышева максимальной степени 64 задействует  $\approx 1000$  оценок во всех случаях (из-за предела безопасности в адаптивной процедуре), но всегда обращается к  $\epsilon_{\text{machine}}$ , за исключением таких функций, как  $x \sin(1/x)$  с известным континуумом проблем;
- ◆ для 1000 случайно выбранных оценочных точек:
  - линейная интерполяция работает хорошо с очень грубой ошибкой  $O(1/n)$  для недифференцируемой  $f$  и  $O(1/n^2)$  для дифференцируемой  $f$ , как и предсказывает теория;
  - сплайны дают немного большие ошибки в первых случаях, но почти достигают  $\epsilon_{\text{machine}}$  во втором случае.

Получается, выбираемый метод интерполяции зависит от конечной цели. Некоторые общие выводы:

- ◆ используйте полиномы Чебышева для любых аналитических  $f$ ;
- ◆ для интерполяции промежуточных результатов проще всего работать с кусочными полиномами;
- ◆ для построения графиков линейная интерполяция набора данных является опцией по умолчанию в MATLAB, потому что другие опции, такие как кубические сплайны, могут привести к колебаниям (см. [23.46]). Ее «некрасивость» — всего лишь обман человеческого глаза.

Сплайны обычно бесполезны, но выигрывают при работе с существующими данными в некоторых случаях:

- ◆ графические данные с линейной интерполяцией слишком грубы;
- ◆ оценка первой и второй производных, где должна обеспечиваться непрерывность оценок.

## 23.9. Интегрирование

Определенный интеграл  $f$  в диапазоне = объем диапазона  $\times$  среднее значение  $f$  в диапазоне. Его нужно найти, имея только возможность оценить  $f$  в любой точке диапазона. Это предполагает четко определенный интеграл, т. е. отсутствие бесконечностей, NaN, бесконечных диапазонов и т. д., хотя многие из них можно обрабатывать с помощью аналитической предварительной обработки. Например, рассмотрим  $\int_{0 \leq x \leq 1} e^x$ . Аналитически ответ равен  $e$ , но мы хотим получить его численно.

В случае одного измерения самый простой способ получить ответ — *правило трапеций*. Линейно интерполируем функцию через равные промежутки времени и интегрируем интерполяцию. Поэтому используем  $n + 1$  точек, чтобы

$$\int_{a \leq x \leq b} f(x) dx \approx \sum_{0 \leq i < n} (x_{i+1} - x_i) \frac{f(x_{i+1}) + f(x_i)}{2}.$$

Из-за информационной сложности это в наихудшем случае оптимально для липшицевой  $f$  (см. [23.60]).

Правило трапеций легко использовать для уже оцененных значений  $f$ , в частности для случайных значений. Предполагается, что диапазон интегрирования определяется наименьшим и наибольшим  $x_i$ . Повторяющиеся значения не вызывают проблем. Единственная хитрость заключается в оценке ошибки, для чего используется *ошибка удвоения*, равная [интегралу по всем значениям — интеграл по половине значений]. Эта оценка является примером сходимости по Коши и может отличаться от реальной ошибки на порядки, но обычно ненамного:

```
pair<double, double> integrateFromData(Vector<pair<double, double> > xyPairs)
{
    assert(xyPairs.getSize() >= 3);
    quickSort(xyPairs.getArray(), 0, xyPairs.getSize() - 1,
        PairFirstComparator<double, double>());
    double result[2] = {0, 0}, last = 0;
    for(int j = 2; j >= 1; --j)
        for(int i = 0; i + j < xyPairs.getSize(); i += j) result[j - 1] +=
            last = (xyPairs[i + j].first - xyPairs[i].first) *
                (xyPairs[i + j].second + xyPairs[i].second)/2;
    if(xyPairs.getSize() % 2 == 0) result[1] += last;
    return make_pair(result[0], abs(result[0] - result[1]));
}
```

Можно сделать то же самое с кусочными многочленами с фиксированным порядком, но это неуклюже. Например, барицентрическая форма не допускает интегрирования (см. комментарии). Интеграция из данных вызывает сомнения как потенциально неправильный подход, но использование правила трапеций кажется хорошим вариантом.

Ошибка удвоения является общей оценкой, применимой к любому алгоритму. Это усовершенствование общей стратегии удвоения до «счастливого значения». Но «счастливое» — это логическое значение, поэтому имеет смысл прекратить удваивать, когда ошибка достаточно мала, потому что достигается убывающая отдача. В качестве оценки погрешности это, возможно, оправдано тем, что использование удвоенного количества информационных точек предположительно приводит к совершенно другому алгоритму, а разница в их результатах является некоторым показателем того, насколько ошибочен один из них.

Правило трапеций сходится, если  $\max(x_{i+1} - x_i) \rightarrow 0$ , а  $f$  является липшицевой с константой  $C$ . Если для простоты предположить интервалы одинаковой длины, наихудшая ошибка в таком интервале равна  $O(\Delta x^2)$ , заданная треугольной формой с вершиной высота ограничена  $C\Delta x/2$ , а площадь —  $C\Delta x^2/2$ , поэтому общая ошибка равна  $O(\Delta x)$ . Эта логика применима ко всем правилам типа  $\sum \Delta x f_{\text{ave}}$ , таким как основные математические суммы Римана, Симпсона и т. д., где  $f_{\text{ave}}$  оценивается как взвешенное среднее нескольких оценок внутри интервала  $\Delta x$ .

Для  $f$  с 2 непрерывными производными и интервалами равной длины:

$$\left| \int_{a \leq x \leq b} f(x) - \Delta x \sum_{0 \leq i < n} f(x_i) \right| \leq \|f''\| \Delta x^2 \frac{b-a}{12} \quad (\text{см. [23.1 и 23.57]},$$

т. е. порядок сходимости равен  $O(n)$ .

Интерполяция Чебышева приводит к эффективной и точной *квадратуре Кленшоу — Кертиса*. Чтобы применить ее, сперва нужно преобразовать диапазон интегрирования в  $[-1, 1]$ . Точность не указана, есть только ограничение на количество оценок, потому что нужно попытаться получить точность  $\varepsilon_{\text{machine}}$ .

```
template<typename FUNCTION> pair<double, double> integrateCC(FUNCTION const& f,
    double a, double b, int maxEvals = 5000, int minEvals = 17)
{
    ScaledChebAB c(adaptiveChebEstimate(ScaledFunctionM1<FUNCTION>(a, b, f),
        maxEvals, minEvals), a, b);
    return c.integrate();
}
```

Для аналитической  $f$  квадратура Кленшоу — Кертиса сходится экспоненциально быстро по  $n$ . В противном случае сходимость полиномиальна по  $n$  и экспоненциальна по количеству непрерывных производных (см. [23.61]).

*Правило Симпсона* использует кусочно-квадратичную интерполяцию, где каждый сегмент оценивается слева (0), посередине (1) и справа (2). Предполагая, что каждый интервал занимает один сегмент:

$$\int_{a \leq x \leq b} f(x) \approx (b-a) \frac{f_0 + 4f_1 + f_2}{6} \quad (\text{см. [23.51]}).$$

Распространение этого на множество сегментов дает ошибку  $O(f^{(4)} \Delta x^4)$ . В работах [23.1 или 23.57] приведено точное выражение. Во многих случаях, например в правилах трапеций и алгоритме Симпсона, ошибка удвоения имеет известное аналитическое выражение, если  $f$  имеет достаточное количество непрерывных производных. Например, у Симпсона деление пополам дает ошибку = разнице/15 (см. [23.1]). Но полагаться на такие оценки ненадежно, в то время как обоснование независимой разницы удвоения является надежным.

*Адаптивное интегрирование* выделяет больше  $f$  оценок субрегионам высшей вариации. Стратегия кучи затем выбирает интервал с наибольшим потенциалом для уменьшения ошибок (см. [23.26]), такие как поиск в пространстве состояний  $A^*$  и алгоритмы ближайших соседей многомерных деревьев. В некотором смысле это самый эффективный способ уменьшить неопределенность в отношении  $[a, b]$  с учетом текущих оценочных баллов. Адаптивное интегрирование также дает более надежную оценку ошибки, чем стратегия удвоения.

Для сходимости требуется, чтобы  $\max \Delta x \rightarrow 0$ , поэтому  $> O(1)$  оценок должно быть дано конкретному интервалу, и можно выбрать значение, равное  $\sqrt{\text{максимальной оценки}}$ . Предпочтительно использовать интервалы с большими ошибками, но у достаточно больших интервалов используется предпочтение по размеру, чтобы избежать обманчивых случаев, когда большой интервал случайно сообщает о небольшой ошибке. Разработка такого компаратора немного сложна. Например, использование относительных размеров интервалов, а не заранее определенной минимальной длины, приводит к не-

транзитивному оператору «<» (приоритетная разработка с несколькими переменными затруднена, поскольку сложно объединить их в один приоритет):

```
template<typename INTEGRATION_INTERVAL, typename FUNCTION> pair<double, double>
    integrateAdaptiveHeap(FUNCTION const& f, double a, double b, double
        eRelAbs = highPrecEps, int maxEvals = 1000000, int minEvals = -1)
{
    typedef INTEGRATION_INTERVAL II;
    if(minEvals == -1) minEvals = sqrt(maxEvals);
    assert(a < b && maxEvals >= minEvals && minEvals >= II::initEvals());
    II i0(f, a, b);
    double result = i0.integrate(), totalError = i0.error();
    AdaptiveIntervalComparator<II> ic = {(b - a)/minEvals};
    Heap<II, AdaptiveIntervalComparator<II> > h(ic);
    h.insert(i0);
    for(int usedEvals = II::initEvals();
        usedEvals < minEvals || (usedEvals + II::splitEvals() <= maxEvals &&
            !isEEqual(result, result + totalError, eRelAbs));
        usedEvals += II::splitEvals())
    {
        II next = h.deleteMin();
        Vector<II> division = next.split(f);
        result -= next.integrate();
        totalError -= next.error();
        for(int i = 0; i < division.getSize(); ++i)
        {
            h.insert(division[i]);
            result += division[i].integrate();
            totalError += division[i].error();
        }
    }
    return make_pair(result, totalError);
}
```

Использование интервалов Симпсона дает хороший результат (реализация не представлена), но предпочтительно использовать формулы более высокого порядка, например *правило Гаусса — Лобатто — Кронрода* (см. [23.22]). Оно основано на полиномиальной интерполяции с ортогональными полиномами на  $[-1, 1]$ , но вместо полиномов Чебышева используются производные полиномов Лежандра. Они обладают рядом важных свойств:

- ◆ точны для полиномов высокой степени при заданном количестве узлов оценки;
- ◆ узлы оценки включают в себя конечные точки  $-1$  и  $1$ , которые нужны для повторного использования оценок в подынтервале;
- ◆ *расширение Кронрода* позволяет построить формулу более высокого порядка при повторном использовании предыдущих оценок. В частности, из 4-точечного правила можно создать 7-точечное правило и разделить интервал на 6 частей, если нужно увеличить глубину.

В работе [23.22] можно найти более подробную информацию. Приведенная далее реализация намного проще в том смысле, что мера ошибки равна интегралу Кронрода по 7 точкам минус менее точный 4-точечный интеграл Гаусса — Лобатто:



```

class IntervalGaussLobattoKronrod
{
    double a, b, y[7];
    void generateX(double x[7]) const
    {
        x[0] = -1; x[1] = -sqrt(2.0/3); x[2] = -1/sqrt(5); x[3] = 0;
        x[4] = -x[2]; x[5] = -x[1]; x[6] = -x[0];
    }
    template<typename FUNCTION> void initHelper(FUNCTION const& f, double fa,
        double fb)
    {
        assert(a < b);
        y[0] = fa;
        y[6] = fb;
        double x[7];
        generateX(x);
        ScaledFunctionM11<FUNCTION> fM11(a, b, f);
        for(int i = 1; i < 6; ++i) y[i] = fM11(x[i]);
    }
    double integrateGL() const
    {
        double w[7] = {1.0/6, 0, 5.0/6, 0};
        w[4] = w[2]; w[5] = w[1]; w[6] = w[0];
        double result = 0;
        for(int i = 0; i < 7; ++i) result += w[i] * y[i];
        return result * (b - a)/2;
    }
    template<typename FUNCTION> IntervalGaussLobattoKronrod(FUNCTION const& f,
        double theA, double theB, double fa, double fb): a(theA), b(theB)
    {initHelper(f, fa, fb);}
public:
    template<typename FUNCTION> IntervalGaussLobattoKronrod(FUNCTION const& f,
        double theA, double theB): a(theA), b(theB)
    {initHelper(f, f(a), f(b));}
    double integrate() const
    {
        double w[7] = {11.0/210, 72.0/245, 125.0/294, 16.0/35};
        w[4] = w[2]; w[5] = w[1]; w[6] = w[0];
        double result = 0;
        for(int i = 0; i < 7; ++i) result += w[i] * y[i];
        return result * (b - a)/2; // нужно обратное масштабирование
    }
    double length() const{return b - a;}
    double error() const{return abs(integrate() - integrateGL());}
    template<typename FUNCTION>
    Vector<IntervalGaussLobattoKronrod> split(FUNCTION const& f) const
    {
        Vector<IntervalGaussLobattoKronrod> result;
        double x[7];
        generateX(x);
        for(int i = 0; i < 7; ++i) x[i] = a + (x[i] - -1) * (b - a)/2;
    }

```

```

    for(int i = 0; i < 6; ++i) result.append(
        IntervalGaussLobattoKronrod(f, x[i], x[i + 1], y[i], y[i + 1]));
    return result;
}
static int initEvals(){return 7;}
static int splitEvals(){return 30;}
};

```

*Правило Гаусса — Лобатто* по  $n$  точкам работает точно для многочленов степени  $2n - 1$  (см. [23.1, 23.57]), поэтому при  $n = 4$  интегрируются ровно полиномы степени 7. Расширение Кронрода по 7 точкам интегрирует ровно полиномы степени 10 (см. [23.14]). Вместо этого использование квадратуры Кленшоу — Кертиса с 4 и 8 точками требует разбиения на 7 интервалов, что хуже, чем на 6. Повторное использование точек не может иметь одновременно адаптивности и высокого порядка.

Учитывая, что Кленшоу — Кертис сходится к  $\varepsilon_{\text{machine}}$  приблизительно за 1000 оценок для очень гладкой  $f$ , а часто даже за 100, ее комбинируют с адаптивным интегрированием для негладких  $f$  (адаптивность чрезвычайно эффективна, если у функции есть разрывы) или очень гладких  $f$ , соединенных разрывом. Выполните  $\min(\max \text{ evals}/2, 1000)$  оценок на нем, а остальные, если необходимо, передайте адаптивному интегратору. Адаптивное интегрирование не может повторно использовать оценки Кленшоу — Кертиса в том виде, в каком они реализованы, но это не страшно с учетом преимущества в простоте:

```

template<typename FUNCTION> pair<double, double> integrateHybrid(
    FUNCTION const& f, double a, double b, double eRelAbs = highPrecEps,
    int maxEvals = 1000000, int minEvals = -1)
{
    int CCEvals = min(maxEvals/2, 1000);
    pair<double, double> resultCC = integrateCC(f, a, b, CCEvals);
    if(isEEqual(resultCC.first, resultCC.first + resultCC.second, eRelAbs))
        return resultCC;
    pair<double, double> resultGLK = integrateAdaptiveHeap<
        IntervalGaussLobattoKronrod>(f, a, b, eRelAbs, maxEvals - CCEvals,
        minEvals);
    return resultCC.second < resultGLK.second ? resultCC : resultGLK;
}

```

Из-за информационной сложности невозможно интегрировать детерминированные  $f$  в точности (если  $f$  не липшицева) (см. [23.12]). Чтобы построить проблемную функцию  $g$ , запустите любой алгоритм интегрирования на  $f = 0$  и запишите оценочные точки  $x_i$ . Затем определите  $g(x) = c \prod (x - x_i)$  для константы  $c$ . Алгоритм скажет, что  $\int g = 0$ , а изменение  $c$  заставляет интеграл принимать любое значение. Но чтобы добиться цели с помощью адаптивного разбиения, достаточно сделать самое первое разбиение случайным образом, потому что после этого последующие обычные точки разбиения также будут случайны (предположим, что  $f$  не сможет запомнить эту точку). Здесь это не реализовано, потому что состязательные  $f$  на практике не встречаются.

Оценки погрешности ненадежны. Критерий Кленшоу — Кертиса является эвристическим, а оценка интегрирования по методу Монте-Карло (обсуждаемая в следующем разделе) имеет достоверность  $< 100\%$ . Для «черного ящика»  $f$  метод Монте-Карло почти всегда является наименее точным, гораздо более безопасным теоретически, потому

что  $f$  может не быть липшицевой. Обычно лучше знать, какой тип  $f$  интегрируется и какие алгоритмы с ним работают. В частности, алгоритм Кленшоу — Кертиса кажется предпочтительным для типичного аналитического  $f$ . Но ни один алгоритм не может точно интегрировать  $f$  с концентрированным пиком, потому что интеграл  $= \infty$  (рис. 23.4).

Функция	CC			CCDoubling			AS			AGLK			Гибрид			MC			TrapData		
	Ош.	Приб.	Оц-ка	Ош.	Приб.	Оц-ка	Ош.	Приб.	Оц-ка	Ош.	Приб.	Оц-ка	Ош.	Приб.	Оц-ка	Ош.	Приб.	Оц-ка	Ош.	Приб.	Оц-ка
Ступенчатая	-3.4	-3.6	4097	-3.4	-3.4	4097	-13.7	-13.7	4229	-14.4	-14.0	17407	-14.4	-14.0	17920	-3.6	-3.2	10000000	-7.8	-6.7	10000000
Модуль	-7.0	-6.7	4097	-7.0	-6.5	4097	-13.7	-13.7	4361	-15.3	-15.7	1027	-15.3	-15.7	1540	-3.3	-3.4	10000000	-14.4	-14.8	10000000
Линейная	-15.7	-15.1	17	-15.7	-15.7	33	-13.7	-13.7	4361	-15.7	-15.7	1027	-15.7	-15.1	17	-3.9	-3.1	10000000	-15.7	-15.7	10000000
Квадратичная	-15.7	-14.9	17	-15.7	-15.7	33	-13.7	-13.7	4353	-15.3	-15.7	1027	-15.7	-14.9	17	-3.5	-3.4	10000000	-13.1	-13.1	10000000
Кубическая	-15.7	-14.8	17	-15.7	-15.7	33	-13.7	-13.7	4345	-15.7	-15.7	1027	-15.7	-14.8	17	-3.9	-3.3	10000000	-15.6	-15.5	10000000
4-й степени	-15.7	-14.7	17	-15.7	-15.7	33	-13.7	-13.7	8329	-15.7	-15.7	1027	-15.7	-14.7	17	-4.0	-3.5	10000000	-12.8	-12.8	10000000
Экспонента	-15.7	-14.8	17	-15.7	-15.7	33	-13.9	-13.8	4353	-15.4	-15.7	1027	-15.7	-14.8	17	-4.2	-3.5	10000000	-13.7	-13.7	10000000
SqrtAbs	-5.2	-5.1	4097	-5.2	-4.9	4097	-14.4	-13.7	15189	-14.8	-13.7	18487	-14.8	-13.7	19000	-5.2	-3.7	10000000	-10.3	-11.3	10000000
Рунге	-15.1	-13.6	1025	-15.0	-15.0	1025	-15.0	-13.7	25345	-15.2	-13.7	17467	-15.2	-13.7	17980	-3.2	-3.0	10000000	-13.8	-13.3	10000000
Логарифм	-15.7	-14.8	33	-15.7	-15.7	33	-13.7	-13.7	4341	-15.5	-15.7	1027	-15.7	-14.8	33	-4.1	-3.9	10000000	-14.6	-14.6	10000000
XSinXM1	-4.3	-5.6	4097	-4.3	-5.4	4097	-10.3	-11.3	999997	-11.8	-11.0	999997	-11.8	-11.0	999490	-4.0	-3.3	10000000	-9.2	-8.3	10000000
Синус	-15.7	-14.5	17	-15.7	-15.7	33	-13.9	-13.7	4365	-15.7	-15.7	1027	-15.7	-14.5	17	-3.3	-3.2	10000000	-15.7	-15.7	10000000
AbsIntegral	-15.7	-13.4	4097	-15.7	-15.7	33	-13.7	-13.7	4345	-15.7	-15.7	1027	-15.7	-15.7	1540	-4.5	-3.5	10000000	-15.7	-15.7	10000000
Тангенс	-15.7	-14.4	33	-15.7	-15.7	33	-15.7	-13.7	4733	-15.7	-14.4	1027	-15.7	-14.4	33	-3.4	-3.2	10000000	-15.7	-15.7	10000000
Нормальная PDF	-15.7	-13.5	129	-15.4	-15.7	257	-15.1	-13.7	19677	-15.2	-13.7	4657	-15.2	-13.7	4786	-3.6	-2.9	10000000	-15.4	-14.7	10000000
Дельта PDF	-4.4	-3.9	4097	-4.4	-3.7	4097	-14.0	-13.7	4541	-15.7	-15.7	17437	-15.7	-15.7	17950	-3.0	-2.5	10000000	-11.5	-11.0	10000000
F575	-1.4	1.3	4097	-1.4	-1.7	4097	-14.8	-13.7	130457	-14.9	-13.7	38767	-14.9	-13.7	39280	-2.2	-0.9	10000000	-2.0	-1.4	10000000
Средние ранги	2.8	4.5	1.3	2.9	2.6	1.9	4.4	4.5	4.6	1.7	1.4	3.5	1.2	2.5	2.5	6.7	6.9	6.0	3.8	3.6	6

Рис. 23.4. Сравнение производительности на некоторых  $f$ .

Гибридный метод дает лучшие результаты. Удвоение Чебышева проигрывает общей интерполяции. Адаптивный Симпсон не так хорош, как адаптивный GLK. Монте-Карло проигрывает 1D

Простой способ выполнить распараллеливание или обойти ограничения памяти адаптивных методов с большим бюджетом вычислений  $n$  состоит в том, чтобы разбить интервал  $[a, b]$  на  $k$  равных частей и интегрировать каждую по отдельности. Каждое произведение получает бюджет  $n/k$ . Допуск к ошибкам труднее распределить, но ошибки от сложения интегралов компенсируются, поэтому при исходной погрешности  $\epsilon$  имеет

смысл использовать  $\min\left(\frac{\epsilon}{\sqrt{k}}, \epsilon_{machine}\right)$ . Для оценки ошибки используется консерватив-

ная сумма оценок ошибок. Небольшая модификация — стратегия менее эффективных рекурсивных интеграторов, которые не используют кучу, а решатели ОДУ задействуют ее по необходимости (обсуждается далее в этой главе), но в рассматриваемом случае она работает хорошо.

Простая эвристика для сингулярностей — установка их равными 0. Часто этот метод работает эффективно, но, как правило, хорошо работает только с адаптивными интеграторами, которые могут адаптироваться к разрыву между 0 и его соседями:

```
template<typename FUNCTION> struct SingularityWrapper
{
    FUNCTION f;
    mutable int sCount;
    SingularityWrapper(FUNCTION const& theF = FUNCTION()): f(theF), sCount(0){}
    double operator()(double x) const
    {
        double y = f(x);
        if(isfinite(y)) return y;
```

```

else
{
    ++sCount;
    return 0;
}
};

```

В моих тестах с этой эвристикой интегрирование Чебышева дает сбой, а адаптивные методы (включая гибридную стратегию) работают хорошо.

Еще одно простое расширение выглядит как  $\int_{a \leq x < \infty} f$ : найти  $\min b$  такое, что при  $x > b$   $f(x)$  оно слишком мало в некотором относительном смысле, и выполнить интегрирование по  $[a, b]$ . «Слишком маленькое» значение обычно равно  $\int_{b \leq x \leq \infty} f \leq \varepsilon \int_{a \leq x \leq b} f$  для некоторого  $\varepsilon$ ; вместо  $\infty$  используйте  $2b$  на стратегию удвоения. Начиная с некоторого начального  $b$  (выбор которого требует аналитических знаний), на каждом новом шаге удваивайте  $b$  и добавляйте  $\int_{b \leq x \leq 2b} f$ , пока следующее изменение не станет слишком маленьким.

Многие полуаналитические методы обсуждаются в работе [23.40], и в ней приведено множество примеров. Там также есть несколько тестовых наборов с известными решениями.

Интегрирование хорошо обусловлено абсолютно, но не относительно (см. [23.12, 23.32]). Даже Монте-Карло вычисляет ответ в диапазоне абсолютной неопределенности. Когда ответ  $\approx 0$ , в лучшем случае можно вычислить больше абсолютных десятичных разрядов и не получить большей относительной точности, когда ответ по-прежнему равен 0. Случаи, когда возникает такая проблема, называются *колебательными интегралами* и обычно обрабатываются с помощью аналитических предварительных вычислений (см. работы [23.12 и 23.40], где описан ряд методов). Некоторые алгоритмы появились сравнительно недавно (см. [23.23]).

Интегрирование плохо обусловлено по отношению к изменениям в диапазоне. Если большая часть интегрального значения сосредоточена вблизи границы, оно может иметь плохую обусловленность в отношении изменения границы (см. [23.63, 23.32]).

## 23.10. Многомерное интегрирование

Простой способ вычислить многомерный интеграл — свести его к одномерному интегралу с помощью *теоремы Фубини* (см. [23.67]): пусть  $X, Y$  — области интегрирования, удовлетворяющие условию о том, что использование  $|f|$  вместо этого приводит к конечному ответу и некоторым второстепенным техническим условиям (т. е. мы работаем с  $\sigma$ -конечными пространствами). Если  $\int_{X \times Y} f$  существует и  $< \infty$ , то  $\int_{X \times Y} f = \int_X \left( \int_Y f \right) = \int_Y \left( \int_X f \right)$ .

Это позволяет вычислять многомерные интегралы рекурсивно, используя любое правило одномерного интегрирования. Количество разрешенных оценок лучше всего делить поровну, поэтому для одномерного интегратора используется алгоритм Кленшоу — Кертиса, поскольку для высокой точности требуется мало оценок, что хорошо работает с проклятием размерности большого  $D$ .

Получить хорошую оценку ошибки немного сложно. Вы можете выполнять анализ в точной арифметике на основе обратного анализа рекурсии. Предположим, что:

- ◆ рекурсивное одномерное интегрирование по переменной  $i$  дает среднюю ошибку  $e_i$ ;
- ◆ базовый вариант интеграции  $1D$  лежит в диапазоне от  $a_0$  до  $b_0$ .

Тогда общая ошибка равна  $\frac{\text{Vol}(X+Y) \sum e_i}{\prod_{0 \leq j \leq i} (b_j - a_j)}$ . Если, кроме того:

- ◆ все  $e_i$  имеют одинаковую величину;
- ◆ любое  $e_i$  значительно меньше любой стороны гиперкуба,

тогда общая ошибка приблизительно равна  $\frac{\text{Vol}(X+Y)e_0}{b_0 - a_0}$ . Таким образом, ошибки на

более низких уровнях рекурсии имеют большее значение. Их легко вычислить и дать хорошую эвристическую оценку. Некоторые хитрости реализации:

- ◆ сохраняйте среднюю ошибку для каждого уровня рекурсии;
- ◆ в качестве базового случая используйте оценочную ошибку оценки  $f$  при правильном округлении;
- ◆ индекс цикла  $i$  для удобства идет в обратном направлении.

```
struct Cheb1DIntegrator
{ // гарантированное получение оценки ошибки
    template<typename FUNCTION> pair<double, double> operator()(
        FUNCTION const& f, double a, double b, int maxEvals)
    {return integrateCC(f, a, b, maxEvals, min(17, (maxEvals - 1)/2 + 1));}
};

template<typename FUNCTION, typename INTEGRATOR1D = Cheb1DIntegrator>
class RecursiveIntegralFunction
{
    FUNCTION f;
    mutable Vector<double> xBound;
    typedef Vector<pair<double, double> > BOX;
    BOX box;
    mutable Vector<pair<double, long long> >* errors; // защита от копирования
    int maxEvalsPerDim; // значение по умолчанию для малых D степени 2+1
    pair<double, double> integrateHelper() const
    {
        INTEGRATOR1D i1D;
        int i = xBound.getSize();
        return i1D(*this, box[i].first, box[i].second, maxEvalsPerDim);
    }
public:
    RecursiveIntegralFunction(BOX const& theBox, int theMaxEvalsPerDim = 33,
        FUNCTION const& theF = FUNCTION()): f(theF), box(theBox), errors(0),
        maxEvalsPerDim(theMaxEvalsPerDim){}
    pair<double, double> integrate() const // основная функция
    {
        errors = new Vector<pair<double, long long> >(box.getSize());
        pair<double, double> result = integrateHelper();
```

```

    double error = 0;
    for(int i = box.getSize() - 1; i >= 0; --i) error = (box[i].second -
        box[i].first) * (error + (*errors)[i].first/(*errors)[i].second);
    result.second += error;
    delete errors;
    return result;
}

double operator() (double x) const // вызывается со стороны INTEGRATOR1D
{
    xBound.append(x);
    bool recurse = xBound.getSize() < box.getSize();
    pair<double, double> result = recurse ? integrateHelper() :
        make_pair(f(xBound), numeric_limits<double>::epsilon());
    if(!recurse) result.second *= max(1.0, abs(result.first));
    xBound.removeLast();
    int i = xBound.getSize();
    (*errors)[i].first += result.second;
    ++(*errors)[i].second;
    return result.first;
}
};

```

Этот код полезен для гладких  $f$  с  $D < 8$  или около того. Как «черный ящик», он не может работать с непрямоугольными доменами, потому что, например, установка  $f = 0$  за пределами желаемого пользовательского домена внутри фиктивного прямоугольника вызовет разрывы и воспрепятствует быстрой сходимости одномерной интеграции для очень гладких  $f$ . Но если вызывающая сторона предоставляет какой-либо неявный граничный решатель, то заданные текущие связанные переменные могут определять границы текущей переменной. Это работает для произвольных выпуклых областей, а невыпуклые области должны быть разбиты вызывающей стороной на выпуклые (здесь это не реализовано).

Границы аналитической погрешности получить трудно, потому что нет границы LIE для  $D > 1$ . В этом случае требуются другие подходы. Этот вопрос не исследуется далее, но для повторного интегрирования в точной арифметике, если каждый одномерный интеграл выполняется точно для любой комбинации связанных переменных, из индукции следует, что результат также оказывается точен.

Для  $D > 1$  детерминированный метод имеет сходимость  $O(1/n^{1/D})$ , и диапазон интегрирования не обязательно должен быть простым. Согласно CLT *интегрирование Монте-Карло* имеет сходимость  $O(1/\sqrt{n})$ . Оно также может обрабатывать непрямоугольные домены, используя выборку «принять-отклонить». Поддерживайте инкрементную статистику по интегралу, чтобы показать ошибку. Чтобы вычислить интеграл с учетом тестера принадлежности, нужно:

1. Найти ограничивающий гиперпрямоугольник и вычислить его объем  $v$ .
2. Создать  $n$  случайных точек, равномерно распределенных внутри гиперпрямоугольника.
3. Пусть  $s = \sum$  значений функции точек в пределах диапазона.
4. Вернуть  $sv/n$ .

Со значением  $f=1$  можно вычислить объем диапазона. Для  $m$  точек в пределах диапазона среднее значение функции  $= s/m$ , а объем  $= vm/n$ . Оценка ошибки = 2 стандартные ошибки:

```
double boxVolume(Vector<pair<double, double> > const& box)
{
    double result = 1;
    for(int i = 0; i < box.getSize(); ++i)
        result *= box[i].second - box[i].first;
    return result;
}

struct InsideTrue{
    bool operator()(Vector<double> const& dummy) const{return true;};};
template<typename TEST, typename FUNCTION> pair<double, double>
MonteCarloIntegrate(Vector<pair<double, double> > const& box, int n,
    TEST const& isInside = TEST(), FUNCTION const& f = FUNCTION())
{
    IncrementalStatistics s;
    for(int i = 0; i < n; ++i)
    {
        Vector<double> point(box.getSize());
        for(int j = 0; j < box.getSize(); ++j)
            point[j] = GlobalRNG().uniform(box[j].first, box[j].second);
        if(isInside(point)) s.addValue(f(point));
    }
    double regionVolume = boxVolume(box) * s.n/n;
    return make_pair(regionVolume * s.getMean(), regionVolume * s.error95());
}
```

Хотя ошибка интегрирования по методу Монте-Карло обычно определяется количественно с точки зрения CLT, если  $|f(x)|$  ограничена константой, неравенство Хёффдинга приводит к конечно-выборочным границам. В противном случае, хотя дисперсия постоянна по  $n$ , она может быстро расти с ростом  $D$ , так что алгоритм бесполезен даже при  $n \approx 10^9$ , хотя на практике такие случаи редки.

Другая проблема заключается в том, что принятие-отклонение на основе произвольной формы не масштабируется с ростом  $D$ . Основная проблема заключается в том, что отношение объема гиперболы к объему ограничивающей гиперболы стремится к нулю (гиперсфера) при  $D \rightarrow \infty$ . Для конкретных предметных областей требуются пользовательские генераторы.

Использование выборки гиперкубов из последовательности Соболя (в главе 21. *Вычислительная статистика* приведено описание и анализ, которые идеально переносятся на интеграцию, поскольку вычисление выполняется в среднем) обычно приводит к гораздо меньшей ошибке. Оценка ошибки 95% является эвристической из-за неслучайной выборки, но работает хорошо (см. [23.28]):

```
template<typename TEST, typename FUNCTION> pair<double, double> SobolIntegrate(
    Vector<pair<double, double> > const& box, int n,
    TEST const& isInside = TEST(), FUNCTION const& f = FUNCTION())
{
    IncrementalStatistics s;
    ScrambledSobolHybrid so(box);
```

```
for(int i = 0; i < n; ++i)
{
    Vector<double> point = so();
    if(isInside(point)) s.addValue(f(point));
}
double regionVolume = boxVolume(box) * s.n/n;
return make_pair(regionVolume * s.getMean(), regionVolume * s.error95());
}
```

## 23.11. Оценка функции

Аппаратное обеспечение может выполнять лишь определенное подмножество операций, таких как  $+$ ,  $-$ ,  $\times$  и  $/$ . Остальные операции реализуются на их основе в программном или аппаратном обеспечении. Специальные функции, такие как  $\sin$  и  $\cos$ , обычно оцениваются следующими способами (см. [23.48]):

1. Разделение домена на подходящие регионы.
2. Создание области для конкретной аппроксимационной модели, обычно основанной на минимаксных полиномах (см. раздел комментариев), решении уравнений или специальных методах.
3. Оценка  $f(x)$ , поиск ее области и использование модели области.

У элементарных функций нужно обращать внимание на каждый бит, потому что в библиотечной реализации ошибки на  $\geq 2$  бита обычно недопустимы. Для проверки часто используются автоматизированные помощники, которые в основном работают с линейными программами (т. е. без циклов). Оценка производительности обычно зависит от  $x$ , но фактически равна  $O(1)$  с большой константой. Алгоритмы весьма специализированы, и большинство из них не связаны с алгоритмами общего численного анализа. Например, обычно получают приближения в высокоточной арифметике и вносят некоторые коррективы для достижения стандартной точности. Можно разбить  $x$  на показатель степени и мантиссу и работать с каждым из них отдельно. Подробнее — в работах [23.48 и 23.6]. Лучше использовать хорошие библиотеки вместо создания собственных реализаций.

Вычисление *элементарных функций* вроде косинуса библиотечными подпрограммами обычно дает правильно округленный ответ с полной точностью. Для общих  $f$  это невозможно из-за *дилеммы создателя таблицы*: чтобы решить, в какую сторону выполнять округление, может потребоваться оценка с произвольной точностью. Поэтому в некоторых случаях гарантируется только *точное округление*, т. е. одно из двух ближайших представлений точного результата. Не забывайте, что вычисление может быть плохо обусловлено — точное округление только контролирует обратную ошибку.

Некоторые пользовательские функции оцениваются путем моделирования, которое может выполняться часами. Оценка функции  $D$  занимает время  $O(D + F(D))$  для некоторого  $F$ .

## 23.12. Оценка производных

Предполагая, что  $f$  дифференцируема, обычно легко вычислить  $f'$  аналитически из представления синтаксического дерева. Но это невозможно, если  $f$  является «черным



ящиком». Итак, вы хотите оценить  $f'$  из оценок  $f$ . Для этого  $f$  должна быть хорошо обусловлена. Например,  $f(x) = \begin{cases} 0, & \text{если } x = 0 \\ x^2 \sin\left(\frac{1}{x}\right) \end{cases}$  дифференцируема всюду, но не непрерывна в нуле и имеет большие колебания поблизости. Фактически линейное число условий  $f''(x)$  управляет чувствительностью  $f'(x)$ .

Основные формулы можно вывести из ряда Тейлора (см. [23.1]): при некотором малом  $h > 0$  и предположении, что  $f$  имеет достаточное количество непрерывных производных, имеем:

- ♦ прямую разность (FD):  $f'(x) = \frac{f(x+h) - f(x)}{h} \pm h \max_{x \leq \xi \leq x+h} \frac{|f''|}{2}$ ;
- ♦ обратную разность (BD):  $f'(x) = \frac{f(x) - f(x-h)}{h} \pm h \max_{x-h \leq \xi \leq x} \frac{|f''|}{2}$ ;
- ♦ центральную разность (CD):  $f'(x) = \frac{f(x+h) - f(x-h)}{2h} \pm h^2 \max_{x-h \leq \xi \leq x+h} \frac{|f^{(3)}|}{6}$ ;
- ♦ пятиточечный шаблон (FPS):  

$$f'(x) = \frac{f(x-2h) + 8f(x+h) - 8f(x-h) - f(x+2h)}{12h} \pm h^4 \max_{x-h \leq \xi \leq x+h} \frac{|f^{(5)}|}{30}.$$

Эти значения также можно получить из полиномиальной интерполяции в наборе точек, разделенных  $h$ , путем вычисления интерполяционного полинома, его аналитического дифференцирования и оценки результата. При заданных неравномерных точках используют барицентрическую интерполяцию.

Малые числа  $h$  дают ошибку оценки во всех формулах из-за сокращений вычитания. Предположим, что  $f(x)$  и все  $f(x+ih)$  имеют ту же ошибку. В частности, пусть обратная ошибка  $= |x|\varepsilon$ . Тогда прямая ошибка  $\leq f'(x)|x|\varepsilon$ . Когда  $h = |x|a$ , ошибка отмены  $\leq \frac{c|f'(x)|\varepsilon}{a}$ , где  $c$  — небольшая константа, так что  $a$  можно считать долей битов, на которую оценка неверна.

Требуется минимизировать как аппроксимацию, так и границы ошибки оценки, а это происходит, когда они равны. Сворачивание некоторых констант в  $c$  для формулы, в которой ошибка аппроксимации зависит от  $k$ -й производной, дает  $a^*(k) = \varepsilon_{\text{factor}}^{1/k}$ , где

$$\varepsilon_{\text{factor}} = \frac{c|f'(x)|\varepsilon}{|x|^{k-1}||f^{(k)}||}. \text{ Для многочлена } f \text{ значения } x \text{ сокращаются, и } \varepsilon_{\text{factor}} \approx \frac{c\varepsilon}{d^{k-1}}, \text{ где } d —$$

это степень. При отсутствии других знаний и ввиду того, что  $\varepsilon$  может быть намного больше, чем  $\varepsilon_{\text{machine}}$ , разумно предположить, что  $\varepsilon_{\text{factor}} = \varepsilon_{\text{machine}}$ . Нужно получить абсолютную погрешность для малых  $x$ , поэтому используйте  $h = \max(1, |x|)\varepsilon_{\text{machine}}^{1/k}$ .

Обратная разность обладает теми же свойствами, что и более естественная передняя разница, и пятиточечный шаблон обычно не лучше, чем центральная разница (обсуждается позже). Это также показывает, что бессмысленно искать лучшую ошибку ап-

проксимации при использовании локальной интерполяции Чебышева в  $[-h, h]$ . Таким образом, прямая и центральная разница являются основными подходами:

```
template<typename FUNCTION> double estimateDerivativeFD(FUNCTION const& f,
    double x, double fx, double fEFactor = numeric_limits<double>::epsilon())
{
    double h = sqrt(fEFactor) * max(1.0, abs(x));
    return (f(x + h) - fx)/h;
}

template<typename FUNCTION> double estimateDerivativeCD(FUNCTION const& f,
    double x, double fEFactor = numeric_limits<double>::epsilon())
{
    double h = pow(fEFactor, 1.0/3) * max(1.0, abs(x));
    return (f(x + h) - f(x - h))/(2 * h);
}
```

Что может пойти не так:

- ◆ ошибка аппроксимации намного больше, чем ожидалось, особенно когда  $f$  не имеет  $k$  производных, а значение  $h$  велико;
- ◆ ошибка оценки намного больше, чем ожидалось, а  $h$  слишком мало;
- ◆ функция  $f$  недифференцируема, и в лучшем случае на выходе получится мусор, например для  $f(x) = |x|$  вычисление вблизи  $x = 0$  даст субградиент для  $x$  и  $h \in [-1, 1]$  в зависимости от формулы;
- ◆ те или иные методы контролируют абсолютную, а не относительную ошибку. Таким образом, вблизи  $x = 0$  возникает большая высокая относительная ошибка, в частности оценка  $\text{sign}(f'(x))$  будет плохо обусловлена. Это может быть важно в некоторых приложениях — например, решении уравнений методом секущих (обсуждается далее в этой главе).

Конкретным примером отказа является вычисление  $\sin(x)$  для больших  $x$ . Из-за уменьшения диапазона старшие биты  $x$  теряются, и только младшие биты определяют результат, поэтому абсолютная обратная ошибка оказывается велика. Аналогично относительное линейное число обусловленности велико из-за большого  $x$ .

Иногда при выборе производных вблизи границ домена требуется осторожность, если  $f$  за их пределами не определена. Возможно, вы захотите использовать гибридный алгоритм, который сначала пробует центральную разность и переключается на перемотку вперед или назад, если в результате получается NaN, но здесь из соображений простоты это не делается.

Также вы можете использовать глобальные оценки на основе интерполяции. Они дают гораздо меньшие ошибки оценки из-за эффективного использования гораздо больших  $h$  и контролируют ошибку аппроксимации с помощью специальных критериев сходимости. В частности, рассмотрим полиномы Чебышева и кусочно-чебышевские многочлены (рис. 23.5).

Некоторые экспериментальные и теоретические выводы говорят, что центральная разность в общем случае превосходит другие методы:

- ◆ в ней используются только две функции оценки;
- ◆ ошибка аппроксимации со значением  $h$  по умолчанию не слишком велика в случае, если у функции менее 3 производных;

- ♦ когда  $f$  не рассчитывается с полной точностью, значение  $h$  не оказывается слишком малым;
- ♦ кубический корень дает некоторую защиту от неправильной оценки  $\epsilon_{\text{factor}}$  на несколько порядков.

Функция	FD		CD		FPS		Cheb Doubling		Cheb64 адаптивная	
	Ошибка	Оценка	Ошибка	Оценка	Ошибка	Оценка	Ошибка	Оценка	Ошибка	Оценка
Ступенчатая	-15.7	2	4.9	2	2.9	4	3.2	4097	-15.7	6175
Модуль	-15.7	2	-0.2	2	0.0	4	0.0	4097	-15.4	1105
Линейная	-9.9	2	-11.3	2	-12.9	4	-15.7	17	-15.4	1105
Квадратичная	-7.7	2	-10.9	2	-12.6	4	-15.7	17	-14.9	1105
Кубическая	-7.6	2	-10.4	2	-12.9	4	-15.3	17	-14.6	1105
4-й степени	-7.1	2	-9.8	2	-12.4	4	-15.1	17	-14.2	1105
Экспонента	-7.8	2	-10.7	2	-12.6	4	-13.1	17	-12.9	1105
Логарифм	-7.8	2	-10.5	2	-12.4	4	-11.9	33	-11.9	1105
XSinXM1	4.0	2	2.5	2	1.6	4	1.8	4097	5.2	1015625
Синус	-7.9	2	-10.8	2	-12.7	4	-13.9	17	-12.0	1105
AbsIntegral	-8.0	2	-5.6	2	-3.6	4	-3.8	4097	-15.1	1105
Тангенс	-8.0	2	-10.8	2	-12.6	4	-12.3	33	-12.0	1105
Нормальная PDF	-8.3	2	-11.0	2	-12.7	4	-12.2	129	-12.2	1105
Дельта PDF	-15.7	2	-11.1	2	-4.88	4	0.0	4097	-12.7	10335
Средние ранги	0.0	1.0	3.8	1.0	2.8	3.0	2.2	4.1	2.3	4.9

Рис. 23.5. Некоторые сравнения производительности методов дифференцирования. Для оценки ошибок используется тот же алгоритм, что и для интерполяции

Среди глобальных методов кусочная чебышевская интерполяция превосходит остальные, несмотря на то, что она не гарантирует ожидание непрерывных производных, хотя для известных аналитических  $f$  лучше всего подходят полиномы Чебышева малой степени.

Для заданных точек лучше всего использовать барицентрическую интерполяцию до пятой степени и, вероятно, основанную на ближайших точках к точке запроса, если не выполнять экстраполяцию (в случае которой барицентрическая формула нестабильна).

Производная оценка через конечные разности бесконечно плохо обусловлена (см. [23.12]), что делает задачу некорректной. Из-за этого ни один такой алгоритм не может оценить производные с высокой точностью в худшем случае. Невозможно выбрать правильное значение  $h$ . Например, когда  $f$  вычисляется с ошибкой  $\approx h$  и не с предполагаемой полной точностью, вся точность теряется.

Чтобы оценить градиент функции  $D$ -переменной, нужно оценить все частные производные. Это занимает время  $O(D^2 + DF(D))$ . Вычисление аналитического градиента занимает время  $O(D + G(D))$  и  $G(D) = O(D)$ , если частная производная зависит от  $O(1)$  других переменных. В формуле центральной разности используются двумерные оценки.

Чтобы разрешить повторное использование кода, удобно создавать масштабированные функции направления, которые можно передать в центральную разность или любой другой алгоритм. Простой способ масштабирования в единичном направлении  $u$  состоит в том, чтобы привести масштаб в соответствие со значениями градиента  $h_i$ , когда  $u$  является направлением оси, и использовать норму  $L_2$  для их интерполяции в других направлениях:

```
double findDirectionScale(Vector<double> const& x, Vector<double> u)
{
    for(int i = 0; i < x.getSize(); ++i) u[i] *= max(1.0, abs(x[i]));
    return norm(u);
}
```

Фокус в том, что функция масштабированного направления должна начинаться со значения шага  $s = s_{\text{scaled}}$ , а не 0:

```
template<typename FUNCTION> class ScaledDirectionFunction
{
    FUNCTION f;
    Vector<double> x, d;
    double scale;
public:
    ScaledDirectionFunction(FUNCTION const& theF, Vector<double> const& theX,
        Vector<double> const& theD): f(theF), x(theX), d(theD),
        scale(findDirectionScale(x, d * (1/norm(d)))){assert(norm(d) > 0);}
    double getS0()const{return scale;} // возвращает x[i], если
        // x - это вектор оси
    double operator()(double s)const{return f(x + d * (s - getS0()));}
};
```

Затем для расчета градиента необходимо расположить векторы осей:

```
template<typename FUNCTION> Vector<double> estimateGradientCD(
    Vector<double> const& x, FUNCTION const& f,
    double fEFactor = numeric_limits<double>::epsilon())
{
    int D = x.getSize();
    Vector<double> result(D), d(D);
    for(int i = 0; i < D; ++i)
    {
        d[i] = 1;
        ScaledDirectionFunction<FUNCTION> df(f, x, d);
        result[i] = estimateDerivativeCD(df, df.getS0(), fEFactor);
        d[i] = 0;
    }
    return result;
}
```

Таким же образом можно оценить якобиан, но, поскольку это функция с векторным значением, обертка не используется, т. к. ее нельзя будет применить повторно:

```
template<typename FUNCTION> Matrix<double> estimateJacobianCD(FUNCTION const&
    f, Vector<double> x, double fEFactor = numeric_limits<double>::epsilon())
{
    int n = x.getSize();
    Matrix<double> J(n, n);
    Vector<double> dx(n);
    double temp = pow(fEFactor, 1.0/3);
    for(int c = 0; c < n; ++c)
    {
        double xc = x[c], h = max(1.0, abs(xc)) * temp;
```

```

    x[c] += h;
    Vector<double> df = f(x);
    x[c] = xc - h;
    df -= f(x);
    x[c] = xc;
    for(int r = 0; r < n; ++r) J(r, c) = df[r]/(2 * h);
}
return J;
}

```

Оценка производных по направлению в единичном направлении  $u$  выполняется еще проще. Реализация вычисляет  $\nabla f d$ , где  $d$  может не иметь единичной нормы:

```

template<typename FUNCTION> double estimateDirectionalDerivativeCD(
    Vector<double> const& x, FUNCTION const& f, Vector<double> const& d,
    double fEFactor = numeric_limits<double>::epsilon())
{ // оценка grad*d, если d не является единичной
    ScaledDirectionFunction<FUNCTION> df(f, x, d);
    return estimateDerivativeCD(df, df.getS0(), fEFactor);
}

```

Насчет использования для удвоения вычислений функции прямой разности часто возникают споры, но даже когда низкая точность вполне подходит, гораздо надежнее применить более точный метод (т. к. получать менее точный ответ быстрее — не вариант). Я попробовал обе системы для решения систем нелинейных уравнений (обсуждаемых далее в этой главе), и хотя прямое различие приводит к меньшему количеству оценок многих  $f$ , во многих других случаях использование якобиана центральной разности повышает точность и уменьшает общее количество оценок  $f$ .

Более сложной задачей является оценка производных высших порядков. Для второй производной можно использовать сочетание нескольких рядов Тейлора вокруг  $x$ . Это дает *формулу центральной разности второго порядка*:

$$f''(x) = \frac{f(x+h) - f(x) + f(x-h)}{h^2} \pm h^2 \max_{x-h \leq \xi \leq x+h} \frac{|f^{(4)}(\xi)|}{12} \quad (\text{см. [23.1]}).$$

Одна из серьезных проблем заключается в том, что ошибка отмены теперь равна  $O(1/h^2)$ . В оптимальном случае  $h = \max(1, |x|) \epsilon_{\text{machine}}^{1/4}$ :

```

template<typename FUNCTION> double estimate2ndDerivativeCD(FUNCTION const& f,
    double x, double fx = numeric_limits<double>::quiet_NaN(),
    double fEFactor = numeric_limits<double>::epsilon())
{
    if(!isfinite(fx)) fx = f(x);
    double h = pow(fEFactor, 1.0/4) * max(1.0, abs(x));
    return (f(x+h) - 2 * fx + f(x-h))/(h * h);
}

```

Чтобы оценить гессиан, можно использовать соотношение *гессиан* = *якобиан*( $\nabla f$ )<sup>T</sup> (см. [23.71]). Из-за численных ошибок это может привести к потере симметрии, поэтому восстановление производится путем усреднения симметричных записей (см. [23.49]).

```
template<typename GRADIENT> Matrix<double> estimateHessianFromGradientCD(
    Vector<double> x, GRADIENT const& g,
    double fEFactor = numeric_limits<double>::epsilon())
{
    Matrix<double> HT = estimateJacobianCD(g, x, fEFactor);
    return 0.5 * (HT + HT.transpose()); // обеспечение симметрии
}
```

При использовании числовых градиентов  $\varepsilon_{\text{factor}} = \varepsilon_{\text{machine}}^{2/3}$ , что предполагает их расчет по центральной разности. Но здесь метод будет выполнять больше оценок, чем необходимо, по сравнению с прямой конечной разностью 2-го порядка (в работе [23.51] приведены формулы). Тем не менее можно получить простое универсальное решение, которое работает в обоих случаях.

## 23.13. Решение нелинейных уравнений и систем

В точной арифметике уравнение или система уравнений в наиболее общем виде задаются некоторой функцией, и требуется найти  $x^*$  такое, что  $f(x^*)$  равно 0. В арифметике с ограниченной точностью приходится довольствоваться меньшим, чтобы корни были проверяемы:

- ◆ для  $D$ -мерных  $f$  и  $x$  требуется найти  $x^*$  и  $\varepsilon$  с минимальным  $\| \varepsilon \|_\infty$  такие, что  $\forall i \in [0, D - 1] f_i(x^*) < 0$  и  $f_i(x^* + \varepsilon) > 0$ . Тогда  $\text{root} \in \text{box}[x^*, x^* + \varepsilon]$ , но это практически только для  $D = 1$ ;
- ◆ найти такой  $\varepsilon$ -корень, что  $\|f(x^*)\|_\infty < \varepsilon$ . Это обратная ошибка, которая не гарантирует, что  $x^*$  близка к корню, но полезна на практике, если  $\varepsilon <$  некоторого небольшого допуска;
- ◆ если дан точный корень  $x^*$ , измерить ошибку в ответе  $x$  как  $\|x - x^*\|_\infty$ .

Предположим, что значения  $f$  и  $x$  имеют одинаковое  $D$ . Решение линейных уравнений имеет смысл только для квадратных матриц. То же самое справедливо для нелинейных систем, потому что перепараметризованные или недопараметризованные системы, скорее всего, либо не имеют решений, либо имеют слишком много решений. Тогда, как и в случае с линейными уравнениями, вместо этого обычно решают задачу минимизации методом наименьших квадратов.

Может показаться, что простейшее решение — свести задачу к оптимизации, решив  $\min \|f(x)\|_2$  с помощью методов оптимизации, описанных в следующей главе. Если решение равно 0, это и есть корень. Но преобразование в минимизацию может создать искусственные локальные минимумы, которые не соответствуют корням, поэтому обычно этот способ используется только в крайнем случае, когда специализированные методы не работают.

Если  $f$  — «черный ящик», возможны следующие варианты:

- ◆ нет решений;
- ◆ одно решение;
- ◆ конечное множество решений;
- ◆ бесконечное множество решений в ограниченной или неограниченной области;
- ◆ некоторое количество непрерывных областей решений.

Таким образом, в худшем случае, даже если у задачи есть решение, она может оказаться сколь угодно плохо обусловленной, потому что даже незначительное отклонение от похожей задачи может привести к тому, что решение появится.

В частности, в одном измерении и с одним корнем плохая обусловленность имеет место в случае, если  $f'$  очень мала вблизи корня  $x^*$ . Дело в том, что поиск корня на самом

деле вычисляет  $f^{-1}$  и  $(f^{-1})' = \frac{1}{f'}$ . Интуитивно понятно, что  $x$  вблизи  $x^*$  имеет почти

одинаковое значение  $f$ , и такое  $x$  можно обоснованно выбрать в качестве ответа. Например, рассмотрим  $f(x) = x^{10}$ . Для  $D > 1$  производные задаются как  $J(f(x))$ , где  $J$  — якобиан, а наличие маленькой записи в столбце  $c$  строки  $r$  означает, что нахождение  $x^*c[r]$  плохо обусловлено.

В одном измерении, если есть интервал, где  $f$ -значения в конечных точках имеют разные знаки, а  $f$  является непрерывной, можно надежно использовать *метод деления пополам*. Пока текущий интервал  $<$  предыдущего, алгоритм заменяет правое/левое значение на среднее, если  $f(\text{среднее})$  и  $f(\text{левое})$  имеют разные/одинаковые знаки:

```
bool haveDifferentSign(double a, double b){return (a < 0) != (b < 0);}
template<typename FUNCTION> pair<double, double> solveFor0(FUNCTION const& f,
    double xLeft, double xRight, double relAbsXPrecision = highPrecEps)
{
    double yLeft = f(xLeft), xMiddle = xLeft;
    assert(xRight > xLeft && haveDifferentSign(yLeft, f(xRight)));
    while(isELess(xLeft, xRight, relAbsXPrecision))
    { // приведенная далее формула надежнее простого усреднения
        xMiddle = xLeft + (xRight - xLeft)/2;
        double yMiddle = f(xMiddle);
        if(haveDifferentSign(yLeft, yMiddle)) xRight = xMiddle;
        else
        {
            xLeft = xMiddle;
            yLeft = yMiddle;
        }
    } // ошибка в лучшем и худшем случае
    return make_pair(xLeft + (xRight - xLeft)/2, xRight - xLeft);
}
```

Сходимость является линейной с  $O(\lg(1/\text{абсолютная точность}))$ , и алгоритм завершает работу во всех случаях. Но для стохастических  $f$  начальные конечные точки должны с чрезвычайно высокой вероятностью иметь разные знаки, чтобы избежать ошибок сравнения. У непрерывных  $f$  корень существует по теореме о среднем значении. Без непрерывности функция может сходиться к значению, которое не является корнем, например к 0 для  $f(x) = 1/x$  с начальным интервалом  $[-1, 1]$ .

Не пытайтесь получить слишком высокую точность — ошибка оценки  $f$  является пределом, требует дополнительных оценок потерь и дает нереалистичную оценку ошибки, основанную на найденном интервале, который может даже не иметь истинных корней из-за наличия шума. Реализация очень надежна со всеми допустимыми входными данными, потому что, например,  $xLeft + xRight$  может переполниться, но эту проблему нужно решать выбором хорошей формулы.

Простой метод, который работает без изменений в любом  $D$  — это *итерация с фиксированной точкой*:

1. Сформировать  $g(x) = f(x) + x$ .
2. Пока алгоритм не сойдется или не будет превышен бюджет оценки:
3.  $x = g(x)$ .
4. Вернуть  $x$  как корень  $f$ , если сошлось.

Алгоритм полезен лишь концептуально, потому что условие сходимости очень ограничено (см. [23.57]): условие  $\|J(x)\|_\infty < 1$  должно выполняться на всей области действия. Так, добавление к  $f$  константы, например 100, может нарушить работу этого метода.

*Метод Ньютона* дает лучшую итерацию. В одном измерении идея состоит в том, чтобы двигаться в направлении  $-f'(x)$  точно до  $f = 0$ , т. е. выполнять шаг  $\Delta x = \frac{-f(x)}{f'(x)}$ .

Алгоритм повторяется до сходимости, т. е. снова делается корректирующий шаг в направлении уменьшения или увеличения.

Для  $D > 1$  разница лишь в том, что используется функция  $J(x)$ , которая должна быть несингулярной. Затем  $\Delta x = -J(x)^{-1} f(x)$ . Обоснование также аналогично — если  $f$  дифференцируема, с помощью многомерного ряда Тейлора близкие значения задаются по линейной модели, т. е.  $f(x + \Delta x) = f(x) + J(x)\Delta x + O(\|\Delta x\|^2)$ , в которой есть решение в нуле 0. Таким образом, получается следующий алгоритм (в упрощенном виде):

1. Начать с пользовательского предположения  $x = x_0$ .
2. До схождения:
3.  $x \pm -J(x)^{-1} f(x)$ .

Алгоритм сходится квадратично, пока  $J$ (корень) не является сингулярной величиной и выполняются некоторые другие условия (см. [23.57]). Но у алгоритма есть и проблемы:

- ◆ когда  $J(x)$  является сингулярной величиной (например, у одномерной  $f(x) = x^3 + 1$  (с корнем  $x = -1$ ) при  $x = 0$ , где  $f'(x) = 0$ ), метод не работает;
- ◆ вызывающая сторона должна предоставить функцию  $J$ , что нельзя сделать в случае «черного ящика»  $f$  и неудобно, когда  $f$  известна;
- ◆ для решения  $\Delta x$  требуется  $O(D^3)$  времени на итерацию без масштабирования;
- ◆ в одномерном случае, когда корень кратен, имеет место только линейная сходимость (см. [23.14]). Это связано с плохой обусловленностью:  $f'(\text{множество корней}) = 0$ .

По существу метод Ньютона полезен только для конкретных задач, в которых сходимость доказана, а формулы функции и ее производной упрощены.

В одномерном случае, если  $f$  дифференцируема, то, если  $f'(x) = 0$ ,  $f$  локально линейна и  $-sf'(x)$  является *направлением спуска*, т. е. для достаточно малых  $s \leq 1$ ,  $\Delta x = -sf'(x)$  приводит к уменьшению  $f$ -значения. В случае  $D > 1$  и единственной  $f' - \nabla(x)$  является направлением спуска, но оно не уникально, поскольку значение  $-d$  такое, что  $d\nabla(x) < 0$ , также дает спуск. Для решения уравнения нужно выполнить спуск до нуля, что за один шаг обычно невозможно даже методом Ньютона.



Для многомерной  $f$  каждый компонент принимает  $x$  в разных направлениях. Но если  $J(x)$  не сингулярна, ньютоновское направление является спуском в том смысле, что  $\|f\|$  убывает в любой норме — при достаточно малых значениях  $s$ :

$$f(x + \Delta x) \approx f(x) + J(x)\Delta x = f(x) + J(x)(-sJ(x)^{-1}f(x)) = (1 - s)f(x).$$

Для вычислительной эффективности примем первое значение  $s$ , которое дает коэффициент уменьшения и называется *возвратом* (или *демпфированием*). Начнем с  $s = 1$  и будем уменьшать его вдвое, пока в выбранной норме (здесь  $\infty$ )  $\|f(x + s\Delta x)\| \leq (1 - as)\|f(x)\|$  для малого  $a$  (обычно  $10^{-4}$ ). Если конечное  $s$  ограничено и выше 0, что должно иметь место, если  $J$  далеко не сингулярна в домене решения, имеет место *достаточно точный спуск*, т. е. фиксированная величина спуска за шаг. В этом случае получаем равномерную сходимость к 0 в норме, используемой при возврате:

```
double normInf(double x){return abs(x);}
template<typename FUNCTION, typename X> bool equationBacktrack(
    FUNCTION const& f, X& x, X& fx, int& maxEvals, X const& dx, double xEps)
{
    bool failed = true;
    for(double s = 1;
        maxEvals > 0 && normInf(dx) * s > xEps * (1 + normInf(x)); s /= 2)
    {
        X fNew = f(x + dx * s);
        --maxEvals;
        if(normInf(fNew) <= (1 - 0.0001 * s) * normInf(fx))
        {
            failed = false;
            x += dx * s;
            fx = fNew;
            break;
        }
    }
    return failed;
}
```

В применяемых на практике алгоритмах производные не нужны, т. к. они аппроксимируются. В одном измерении *метод секущих* является аналогом метода Ньютона, за исключением оценки  $f'$  по конечной разности, основанной на двух последних оценках. Для первого шага используется центральная разность. Для чистого метода секущих без возврата есть локальная теорема сходимости (в работе [23.14] авторы также выводят константу): Если в некоторой окрестности корня  $x^*$  функция  $f$  имеет две непрерывные производные,  $f'(x^*) \neq 0$ , а  $x_0$  и  $x_{-1}$  достаточно близки к  $x^*$ , алгоритм сходится с порядком  $(1 + \sqrt{5})/2$ . В работе [23.58] приведен подробный анализ условий и сходимости многих методов решения одномерных уравнений. В работе [23.59] также делается то же самое для  $D > 1$ .

В случае  $D > 1$  простым расширением является оценка  $J$  с использованием конечных разностей на каждом шаге. Но хочется добиться большей эффективности и выполнять оценку с помощью существующих оценок функций. Частным обобщением является *метод Бройдена*. Идея его состоит в том, что после обновления  $J$  должно быть таким,

чтобы  $J\Delta x = \Delta f$ . Среди таких  $J$  нужно выбрать ту, которая минимизирует  $\Delta J$  по норме Фробениуса, обладающей здесь некоторыми хорошими свойствами (см. [23.18]).

Это приводит к понижению ранга  $J += u \otimes v$ , где  $u = \frac{\Delta f - J\Delta x}{\|\Delta x\|}$  и  $v = \frac{\Delta x}{\|\Delta x\|}$ . Начальное  $J$

оценивается с помощью конечных разностей и обновляется с помощью QR-разложения.

Когда  $J$  является сингулярной (или почти сингулярной), проблема заключается в том, что не существует направления спуска, которое работало бы для всех функциональных компонентов. Разумная эвристика, применимая как к Бройдену, так и к методу секущих, состоит в том, чтобы делать случайный шаг, взятый из нормального (0, размер последнего успешного полного шага) распределения. Это небольшая подстройка для защиты от сингулярностей, которая в литературе не относится к числу стандартных:

```
template<typename X> struct BroydenSecant
{ // для метода секущих
    static int getD(double dummy){return 1;}
    static double generateUnitStep(double dummy, double infNormSize)
    {return GlobalRNG().normal(0, infNormSize);}
    class InverseOperator
    {
        double b;
    public:
        template<typename FUNCTION> InverseOperator(FUNCTION const& f,
            double x): b(estimateDerivativeCD(f, x)){}
        void addUpdate(double df, double dx){b = df/dx;}
        double operator*(double fx) const{return fx/b;}
    };
};

template<> struct BroydenSecant<Vector<double>>
{ // для Бройдена
    typedef Vector<double> X;
    static int getD(X const& x){return x.getSize();}
    static X generateUnitStep(X const& x, double infNormSize)
    {
        X dx(getD(x));
        for(int j = 0; j < dx.getSize(); ++j)
            dx[j] = GlobalRNG().normal(0, infNormSize);
        return dx;
    }
    class InverseOperator
    {
        QRDecomposition qr;
    public:
        template<typename FUNCTION> InverseOperator(FUNCTION const& f,
            X const& x): qr(estimateJacobianCD(f, x)){}
        void addUpdate(X const& df, X dx)
        {
            double ndx2 = norm(dx);
            dx *= (1/ndx2);
            qr.rank1Update(df * (1/ndx2) - qr * dx, dx);
        }
    };
};
```

```

        X operator*(X const& fx) const { return qr.solve(fx); }
    };
};

```

Основной алгоритм одинаков как для Бройдена, так и для секущих, а в реализации вводятся некоторые дополнительные хитрости:

- ◆ текущее значение  $J$  может быть достаточно неточным и не давать спуска. Поэтому, когда возврат не удастся, нужно выполнить повторную инициализацию, используя конечные разности;
- ◆ если повторная инициализация не дает спуска, попробуйте случайные шаги. Если случайный шаг дает спуск, нужно выполнить повторную инициализацию в надежде, что функция не станет сингулярной.

Разумный способ оценить ошибку результата, по-видимому, заключается в использовании размера последнего успешного полного шага, т. е. до применения обратного отслеживания. Эта оценка несколько консервативна, особенно когда большой шаг достигает почти 0, но кажется устойчивой к слишком маленьким успешным шагам и исчерпанию оценок и т. д.:

```

template<typename FUNCTION, typename X> bool equationTryRandomStep(
    FUNCTION const& f, X& x, X& fx, double stepNorm)
{
    X dx = BroydenSecant<X>::generateUnitStep(x, stepNorm), fNew = f(x + dx);
    bool improved = normInf(fNew) < normInf(fx);
    if(improved)
    {
        x += dx;
        fx = fNew;
    }
    return !improved;
}

template<typename FUNCTION, typename X> pair<X, double> solveBroyden(
    FUNCTION const& f, X const& x0, double xEps = highPrecEps,
    int maxEvals = 1000)
{
    int D = BroydenSecant<X>::getD(x0), failCount = 0;
    X x = x0, fx = f(x);
    assert(D == BroydenSecant<X>::getD(fx) && maxEvals >= 2 * D + 1);
    typedef typename BroydenSecant<X>::InverseOperator BIO;
    BIO B(f, x);
    maxEvals -= 2 * D + 1;
    double lastGoodNorm = 1, xError = numeric_limits<double>::infinity();
    if(!isfinite(normInf(fx))) return make_pair(x, xError);
    while(maxEvals > 0)
    {
        if(failCount > 1)
        { // после второй ошибки используется случайный шаг
            --maxEvals;
            if(!equationTryRandomStep(f, x, fx, lastGoodNorm))
            {
                assert(normInf(f(x)) <= normInf(f(x0)));
            }
        }
    }
}

```

```

        if(maxEvals >= 2 * D + 1) // для следующего шага нужно
                                // достаточное количество оценок
        {
            B = BIO(f, x);
            maxEvals -= 2 * D;
            failCount = 0; // возврат к нормальному распределению
        } // иначе продолжаем делать случайные шаги
        continue;
    }
}
X dx = B * -fx, oldFx = fx, oldX = x;
double ndx = normInf(dx);
if(!isfinite(ndx))
{ // возможен сингуляр в результате неудачного
  // обновления или оценки
  ++failCount;
  continue;
}
if(ndx < xEps * (1 + normInf(x))) break; // полный шаг слишком мал
if(!equationBacktrack(f, x, fx, maxEvals, dx, xEps))
{
    assert(normInf(f(x)) <= normInf(f(x0)));
    xError = lastGoodNorm = ndx; // последний успешный полный шаг
    failCount = 0;
}
else ++failCount;
if(failCount == 1) // повторная оценка J после первой неудачи
    if(maxEvals >= 2 * D + 1) // для следующего шага нужно достаточно оценок
    {
        B = BIO(f, x);
        maxEvals -= 2 * D;
    }
    else ++failCount; // если шаги невозможны
else B.addUpdate(fx - oldFx, x - oldX);
}
return make_pair(x, xError);
} // для типобезопасности вроде int x0 используется обертка
template<typename FUNCTION> pair<double, double> solveSecant(FUNCTION const&
    f, double const& x0, double xEps = highPrecEps, int maxEvals = 1000)
{return solveBroyden(f, x0, xEps, maxEvals);}

```

В точной арифметике базовый метод Бройдена (без возврата, переоценки или случайных шагов) при некоторых условиях сходится сверхлинейно (см. [23.49]), но, в отличие от одномерного метода секущих, скорость сходимости неизвестна (см. [23.18]). Некоторые дополнительные аспекты метода Бройдена рассмотрены в работе [23.44].

Для больших значений  $D$  время выполнения  $O(D^3)$  для первой итерации и занимаемое пространство  $O(D^2)$  не масштабируются. Поэтому нужна для оптимизации реализация с ограниченной памятью, такая как L-BFGS (см. главу 24. Численная оптимизация). Чтобы сделать формулу Бройдена удобной для вычислений, примените формулу Шермана — Моррисона к  $B = J^{-1}$ , чтобы получить ранг-1 обновление с  $B_{i+1} = B_i + u_i \otimes v_i$ ,

где  $u_i = \frac{\Delta x_i - B_i \Delta f_i}{\Delta x_i B_i \Delta f_i}$  и  $v_i = \Delta x_i B_i$  (см. [23.18]), можно выбрать значение  $B_0 = I$ . Оно не

имеет времени выполнения  $O(D^3)$  для первой итерации, поэтому вы можете задаться вопросом, почему реализация QR вообще используется. Проблема в том, что использование  $B_0 = I$  — плохой выбор при возврате, потому что результирующее направление не обязательно должно быть спуском. На самом деле, любая перестановка, принятая с равной вероятностью, будет лучшим приближением.

Использование какой-либо формы предварительной обработки имеет свои ограничения. Например, использование  $m$  итераций для  $m < D$ , в каждой из которых вычисляется прямая разность нового случайного компонента  $x$ . Для  $m < D$  это почти эквивалентно оценке  $J$ , за исключением того, что «априорное»  $B_0 = I$  вряд ли является агностическим и влияет на результат. Кажется, этот метод работает для реализации с ограниченной памятью как для инициализации, так и для повторной оценки.

Итак,  $B_{i+1} = B_0 + \sum u_i \otimes v_i$ . Как и в случае с L-BFGS, реализация с ограниченной памятью сохраняет  $m$  самых последних  $\Delta x_j$  и  $\Delta f_j$  для  $0 \leq j < m$  и формирует из них  $B_m$ . Хитрость заключается в вычислении  $B_m x$  без явного формирования  $B_m$ . Учитывая, что  $B_m = I + \sum u_j \otimes v_j$ , следует использовать динамическое программирование. Рассмотрим также более неуклюжую рекурсию — интуитивно понятно, что последняя сохраняет тип, неявно работая с матрицей  $B$ , а первая работает с компонентами  $x$  напрямую. Сохраняйте не только  $Bx$ , но и  $B_{ij} \Delta f_{ij}$  и  $\Delta x_j B_j$  для вычисления  $u_j$  и  $v_j$ . Это занимает  $O(m^2 D)$  времени и  $O(mD)$  пространства. Поэтому  $B_m$  можно представить как оператор, а не как матрицу:

```
class BroydenLMInverseOperator
{
    typedef Vector<double> X;
    int m;
    Queue<pair<X, X> > updates;
    struct Result
    {
        X Bfx;
        Vector<X> Bdfs, dxBs;
    };
public:
    BroydenLMInverseOperator(int theM): m(theM){}
    void addUpdate(X const& df, X const& dx)
    {
        if (updates.getSize() == m) updates.pop();
        updates.push(make_pair(df, dx));
    }
    X operator*(X const& fx) const
    {
        int n = updates.getSize();
        X Bfx = fx; // признак плохого случая
        Vector<X> Bdfs(n), dxBs(n);
        for (int i = 0; i < n; ++i)
        {
            Bdfs[i] = updates[i].first;
```

```

        dxBs[i] = updates[i].second;
    }
    for(int i = 0; i < n; ++i)
    {
        X u = (updates[i].second - Bdfs[i]) *
            (1/dotProduct(updates[i].second, Bdfs[i]));
        if(!isfinite(normInf(u))) continue; // защита от деления на 0
        Bfx += outerProductMultLeft(u, dxBs[i], fx);
        for(int j = i + 1; j < n; ++j)
        {
            Bdfs[j] += outerProductMultLeft(u, dxBs[i], updates[j].first);
            dxBs[j] += outerProductMultRight(u, dxBs[i],
                updates[j].second);
        }
    }
    return Bfx;
}
};

```

Приближение с ограниченной памятью не отвечает предположению о направлении спуска, в нем нет возможности выполнить повторную оценку, и возврат может не сработать. Но обновления, которые дают плохую оценку якобиана, будут удалены через  $m$  шагов, так что нет необходимости и в случайных шагах.

Может показаться, что для обеспечения монотонного уменьшения  $\|f(x)\|_\infty$  лучше всего использовать поиск с возвратом и случайные шаги, но, судя по моим экспериментам, это плохой вариант. Лучше постараться предотвратить одиночные плохие шаги. Кажется, удобнее всего использовать возврат с уменьшением шага в 10 раз, чтобы  $\|f(x)\|_\infty$  не увеличивалось на превышающий это значение коэффициент. Такой метод лучше, чем никакого. Несколько идей в его поддержку:

- ◆ плохие шаги легко восстановить, потому что вклад производной от плохого шага, вероятно, приведет к обращению большей части этого шага. Так что не нужно тратить оценки на то, чтобы исправить то, что исправится само. Но следует сохранить лучшее решение, найденное до сих пор, на случай, если расхождение будет близко к решению;
- ◆ этот метод, по-видимому, необходимо изучить, чтобы он был эффективен для больших  $D$ . Уменьшение шагов на коэффициент 10 дешево с точки зрения оценки и, по-видимому, дает больше возможностей для исследования, чем, возможно, использование  $\Delta x$ , которое было бы подходящим для конечной разностной оценки производной по направлению.

```

template<typename FUNCTION> pair<Vector<double>, double> solveLMBroyden(
    FUNCTION const& f, Vector<double> const& x0, double xEps = highPrecEps,
    int maxEvals = 1000, int m = 30)
{
    int D = x0.getSize();
    BroydenLMInverseOperator B(m);
    Vector<double> x = x0, fx = f(x);
    double s = 1, xError = numeric_limits<double>::infinity();

```

```

while(maxEvals-- > 0)
{
    Vector<double> dx = B * -fx * s;
    double ndx = normInf(dx)/s; // норма полного шага
    // какие-то проблемы со слишком малым шагом
    if(!isfinite(ndx) || ndx < xEps * (1 + normInf(x))) break;
    Vector<double> fNew = f(x + dx);
    if(normInf(fNew) > 10 * normInf(fx)) s /= 10;
    else
    {
        B.addUpdate(fNew - fx, dx);
        fx = fNew;
        x += dx;
        xError = ndx; // используется последний полный шаг
        s = 1; // возврат к нормальному шагу
    }
}
return make_pair(x, xError);
}

template<typename FUNCTION> pair<Vector<double>, double> solveLMBroyden(
    FUNCTION const& f, Vector<double> const& x0, double xEps = highPrecEps,
    int maxEvals = 1000, int m = 30)
{
    int D = x0.getSize();
    BroydenLMInverseOperator B(m);
    Vector<double> x = x0, fx = f(x), xBest = x, fBest = fx;
    double s = 1, xError = numeric_limits<double>::infinity(), eBest = xError;
    while(maxEvals-- > 0)
    {
        Vector<double> dx = B * -fx * s;
        double ndx = normInf(dx)/s; // норма полного шага
        // какие-то проблемы со слишком малым шагом
        if(!isfinite(ndx) || ndx < xEps * (1 + normInf(x))) break;
        Vector<double> fNew = f(x + dx);
        if(normInf(fNew) > 10 * normInf(fx)) s /= 10;
        else
        {
            B.addUpdate(fNew - fx, dx);
            fx = fNew;
            x += dx;
            xError = ndx; // используется последний полный шаг
            s = 1; // возврат к нормальному шагу
            if(normInf(fx) < normInf(fBest))
            {
                xBest = x;
                fBest = fx;
                eBest = xError;
            }
        }
    }
    return make_pair(xBest, eBest);
}

```

Эксперименты, описанные в работе [23.64], предполагают, что большие значения  $m$  работают лучше, во всех случаях требуется  $m \geq 3$ , и значение 20 кажется безопасным, но мы на всякий пожарный используем 30. Алгоритм с ограниченной памятью хотя и полезен, но имеет недостатки:

- ◆ нет доказуемой сходимости;
- ◆ нет четких указаний по выбору  $m$ , т. к., в отличие от L-BFGS,  $m = 3$  недостаточно для многих задач;
- ◆ потенциальная линейная зависимость значения  $m$  от  $\Delta x_j$  ставит под сомнение выбор базиса. Несколько алгоритмов из [23.64] работают на разных базисах, но, чтобы выбрать лучший, придется попотеть;
- ◆ в моих тестах алгоритм часто не сходится, в отличие от версии с полной памятью, — нахождение направления спуска не гарантируется.

Таким образом, алгоритм с ограниченной памятью имеет смысл только как эвристическая часть гибридного глобального решателя (обсуждается далее в этой главе). Здесь мы выбираем между QR и этим алгоритмом для  $D < 200$ . Для локального решения следует использовать только QR или более дорогой гибрид:

```
template<typename FUNCTION> pair<Vector<double>, double> solveBroydenHybrid(
    FUNCTION const& f, Vector<double> const& x0, double xEps = highPrecEps,
    int maxEvals = 1000, int changeD = 200)
{
    return x0.getSize() > changeD ? solveLMBroyden(f, x0, xEps, maxEvals) :
        solveBroyden(f, x0, xEps, maxEvals);
}
```

Для  $D > 1$  наличие единственной функции с градиентом 0 приведет к сингулярности  $J$ . В этом заключается фундаментальный недостаток представленных локальных алгоритмов, который можно «исправить», попытавшись перейти к решению с минимальной нормой 2. Но это равносильно сведению к задаче минимизации.

Одно уравнение может породить множество других проблем, таких как искажение норм из-за отсутствия масштаба или NaN. Алгоритмы типа «черный ящик»  $f$  не могут справиться с такими проблемами и предполагают, что их не бывает. Переменные должны иметь примерно одинаковый масштаб, иначе  $\|x\|$  бессмысленно. Некоторые реализации (см. [23.18]) масштабируются неявно, но здесь масштабирование ложится на самого пользователя. Жесткие решатели ОДУ (обсуждаемые далее в этой главе) игнорируют это, поскольку предполагается, что входные данные уже хорошо масштабированы.

Наилучшая достижимая точность в конечном счете ограничена точностью  $f$ . Для любых  $x$  и  $\Delta x$  на уровне  $\epsilon_{\text{machine}}$  значения  $f(x + \Delta x)$  не обязательно должны сохранять аналитические свойства  $f$ , такие как выпуклость или монотонность. Таким образом, методы, которые достигают высокой точности и пробных шагов малой величины, фактически делают случайные шаги, поэтому они должны быть осторожны с критериями сходимости и не требовать точности, которая может зря растратить предел оценки.

Чтобы найти сложные корни уравнения с одной переменной, можно преобразовать задачу в двумерную (см. [23.57]). Учитывая, что  $z_{\text{output}} = f(z)$  для  $z = x + iy$ , создайте  $g$  таким образом, что  $(x_{\text{output}}, y_{\text{output}}) = g(x, y)$  оценивается через  $f$ .



До сих пор представленные методы предполагали наличие информации об  $f$ , т. е. либо наличие интервала смены знака и непрерывности, либо неособое  $J$ , что давало достаточный спуск. Но если  $f$  — это «черный ящик», для запуска алгоритма нужна эвристическая выборка.

Часто нужно получить границы с некоторой случайной точки в одном измерении. *Экспоненциальный поиск*, также называемый *брекетингом*, шагает от точки предположения на некоторое расстояние  $d$ , удваивая  $d$  после каждого шага, пока не будет достигнуто некоторое условие завершения. Вы можете двигаться в направлении  $+$ , в направлении  $-$  или в обоих направлениях. Обычно делают и то и другое, потому что догадка предполагается верной — сначала шаг на  $d$ , затем на  $-d$ , затем на  $2d$  и т. д. Односторонние варианты хороши при наличии конкретных ограничений на уровне предметной области. Теорема: двусторонний и односторонний экспоненциальный поиск сходятся, если область, в которой  $f$  имеет тот же знак, что и  $f(x_0)$ , ограничена в любом направлении поиска. В частности, двусторонний экспоненциальный поиск сходится, если  $g = \text{sign}(f)$  монотонна. Доказательство: ограниченная область в конечном итоге будет проанализирована. Для монотонных  $g$  требуется соблюдение условия ограниченной области  $\forall x_0$  и  $d$ . Это бывает, например, в случаях численного обращения функции CDF для генерации случайных величин обратным методом (см. главу 6. *Генерация случайных чисел*). Но алгоритм не должен завершаться слишком рано, а это возможно в любой реализации.

Если условия не выполняются, возможно, вы все равно сможете случайно найти изменение знака, поэтому следует использовать маленькое масштабированное  $d = 0,001\max(1, |x_0|)$  по умолчанию и допускать до 30 удвоений:

```
template<typename FUNCTION> pair<double, double> find1SidedInterval0(
    FUNCTION const& f, double x0 = 0, double d = 0.001, int maxEvals = 30)
{
    assert(maxEvals >= 2 && isfinite(x0 + d) && d != 0 &&
        !isEEqual(x0, x0 + d));
    for (double xLast = x0, f0 = f(x0); --maxEvals > 0; d *= 2)
    {
        double xNext = x0 + d, fNext = f(xNext);
        if (!isfinite(xNext) || isnan(fNext)) break;
        if (haveDifferentSign(f0, fNext))
            return d < 0 ? make_pair(xNext, xLast) : make_pair(xLast, xNext);
        xLast = xNext;
    }
    return make_pair(numeric_limits<double>::quiet_NaN(),
        numeric_limits<double>::quiet_NaN());
}

template<typename FUNCTION> pair<double, double> findInterval0(
    FUNCTION const& f, double x0, double d, int maxEvals)
{
    assert(maxEvals >= 2 && isfinite(x0 + d) && isELess(x0, x0 + d));
    for (double xLast = x0, f0 = f(x0); --maxEvals > 0; d = -d)
    {
        double xNext = x0 + d, fNext = f(xNext);
        if (!isfinite(xNext) || isnan(fNext)) break;
        if (haveDifferentSign(f0, fNext)) return d < 0 ?
            make_pair(xNext, x0 - (xLast - x0)) : make_pair(xLast, xNext);
    }
}
```

```

    if(d < 0)
    {
        xLast = x0 - d;
        d *= 2;
    }
}

return make_pair(numeric_limits<double>::quiet_NaN(),
    numeric_limits<double>::quiet_NaN());
}

template<typename FUNCTION> pair<double, double> exponentialSearch(
    FUNCTION const& f, double x0 = 0, double step = 0.001,
    double xERelAbs = highPrecEps, int maxExpEvals = 60)
{
    pair<double, double> i0 = findInterval0(f, x0, step * max(1.0, abs(x0)),
        maxExpEvals);
    return !isnan(i0.first) ? solveFor0(f, i0.first, i0.second, xERelAbs) : i0;
}

template<typename FUNCTION> pair<double, double> exponentialSearch1Sided(
    FUNCTION const& f, double x0 = 0, double step = 0.001, // отрицательное значение
                                                           // для движения влево
    double xERelAbs = highPrecEps, int maxExpEvals = 60)
{
    pair<double, double> i0 = find1SidedInterval0(f, x0,
        step * max(1.0, abs(x0)), maxExpEvals);
    return !isnan(i0.first) ? solveFor0(f, i0.first, i0.second, xERelAbs) : i0;
}

```

В этой реализации односторонний и двусторонний брекетинг гарантированно находят допустимый интервал или сигнализируют об ошибке с точной арифметикой и детерминированным  $f$ . Таким образом, при тех же условиях экспоненциальный поиск гарантированно не вызовет бинарный поиск с неправильным интервалом (что приводит к ошибке проверки), за исключением случаев, когда  $f$  рандомизирована (и тогда ничего нельзя сделать). Но с неточной арифметикой могут возникнуть проблемы — например,  $f$  может при заключении в скобки иметь один знак, а при бинарном поиске — другой, из-за плохо обусловленного характера оценки знака вблизи 0. Причины тому те же, что и те, по которым повторное вычисление  $f$  не обязательно дает то же значение. Неясно, как лучше всего решить эту проблему — возможно, стоит ориентироваться только на определение знака с некоторым буфером  $\varepsilon$  и рассматривать более близкое к нему значение как решение. Но в этом случае нет способа выбрать безопасное значение  $\varepsilon$ , однако такая проблема на практике возникает редко.

В отличие от бинарного поиска, не гарантируется, что брекетинг будет работать для непрерывной  $f$ , поскольку возможен перелет через область, где  $f$  дважды меняет знак или обращается в NaN для предметной области. Поиск интервала смены знака, где  $f$  непрерывна, гарантирует, что найдется решение. Например,  $f(x) = x^2$  имеет корень, но не имеет интервалов смены знака, а  $f(x) = x^2 - \varepsilon$  имеет очень малый интервал изменения знака, который вряд ли будет найден. Можно выполнять случайный поиск, пока не найдется точка с другим знаком, но метод секущих кажется более полезным.

Если хорошая начальная точка неизвестна, необходимо использовать некоторую глобальную стратегию. В отличие от глобальной оптимизации (см. главу 24. Численная

оптимизация), структура глобального ландшафта для решения уравнений мало говорит о том, где может быть решение, потому что  $\infty$ -норма в основном обращает внимание на одну переменную. Таким образом, случайная выборка была бы лучшей стратегией исследования, по крайней мере теоретически (рис. 23.6):

```
template<typename FUNCTION> pair<double, double> solveSecantGlobal(
    FUNCTION const& f, double xEps = highPrecEps, int nSamples = 100)
{
    assert(nSamples > 0 && xEps >= numeric_limits<double>::epsilon());
    pair<double, double> best;
    double nBestfx;
    for(int i = 0; i < nSamples; ++i)
    {
        double x = GlobalRNG().Levy() * GlobalRNG().sign();
        pair<double, double> next = solveBroyden(f, x, xEps);
        double nNextfx = normInf(f(next.first));
        if(i == 0 || nNextfx < nBestfx)
        {
            best = next;
            nBestfx = nNextfx;
        }
    }
    return best;
}
```

Функция	Эксп. поиск		Секущие		Глобалн. секущие	
	Ошибка	Оценка	Ошибка	Оценка	Ошибка	Оценка
Линейная	-13.24	73	-10.45	5	-9.77	600
Квадратичная	-15.65	60	-10.24	18	-10.83	2007
Полином	-15.65	60	-13.04	233	-13.05	23645
Sqrt2F_2_0	-13.54	71	-10.64	49	-9.85	1259
Sqrt2F_2_2	-13.54	71	-10.64	49	-12.17	1282
Средние ранги	1.0	1.8	2.6	1.2	2.4	3.0

Рис. 23.6. Эксперименты на некоторых  $f$ . Большинство функций имеют несколько корней, поэтому ошибка — это ошибка  $y$ . Все методы работали хорошо, но экспоненциальному поиску, похоже, повезло. У метода секущих было меньше всего оценок

Случайная выборка в одном измерении кажется эффективной только для метода секущих, и в ней используется распределение с толстым хвостом, такое как Леви. Оно позволяет делать масштабируемые по  $x$  шаги от 0 или текущей лучшей точки. Тем не менее, согласно моим экспериментам, оптимизация с  $\infty$ -нормой работает хорошо. Но она не будет работать для сингулярных якобианов:

```
template<typename FUNCTION> pair<Vector<double>, double> solveBroydenLevy(
    FUNCTION const& f, int D, double xEps = highPrecEps, int nSamples = 1000)
{
    assert(nSamples > 0 && xEps >= numeric_limits<double>::epsilon());
    typedef Vector<double> X;
    pair<X, double> best;
```

```

double nBestfx;
for(int i = 0; i < nSamples; ++i)
{
    X x = GlobalRNG().randomUnitVector(D) * GlobalRNG().Levy();
    pair<X, double> next = solveBroydenHybrid(f, x, xEps);
    double nNextfx = normInf(f(next.first));
    if(i == 0 || nNextfx < nBestfx)
    {
        best = next;
        nBestfx = nNextfx;
    }
}
return best;
}

```

Еще один вариант — изменить задачу на глобальную оптимизацию. В определенных задачах выполнение локального поиска с номинальной точки, такой как 0, может сэкономить время, но обычно это не имеет значения. Простая ошибка равна:

$$\max \left( \sqrt{\epsilon}, \frac{\|\nabla g(x_{found})\|_{\infty}}{\max(1, |g(x_{found})|)} \right), \text{ где } g = \|f\|_2.$$

```

template<typename FUNCTION> struct NormFunction
{
    FUNCTION f;
    NormFunction(FUNCTION const& theF): f(theF){}
    double operator() (Vector<double> const& x) const {return norm(f(x));}
};

template<typename FUNCTION> pair<Vector<double>, double> solveByOptimization(
    FUNCTION const& f, Vector<double> const& x0, int maxEvals = 1000000)
{ // в качестве оценки ошибки используется масштабированный градиент
    NormFunction<FUNCTION> nf(f);
    pair<Vector<double>, double> xy = hybridBeforeLocalMinimize(nf,
        UnboundedSampler(x0), AgnosticStepSampler(),
        makeAgnosticBox(x0.getSize()), maxEvals - 2);
    double errorEstimate = max(normInf(estimateGradientCD(xy.first, nf))/
        max(1.0, abs(xy.second)), defaultPrecEps);
    return make_pair(xy.first, errorEstimate);
}

```

Это дает разумное решение для сингулярных  $J$ . Используйте решение оптимизации в качестве отправной точки для одного запуска локального поиска. Можно гибридизировать случайную выборку с помощью оптимизации:

```

template<typename FUNCTION> pair<Vector<double>, double>
    hybridEquationSolve(FUNCTION const& f, int D,
        double xEps = highPrecEps, int maxEvals = 1000000)
{ // берется оптимум а затем для увеличения точности на ней выполняется
    // локальный поиск Бройдена
    int broydenEvals = max(1000, maxEvals/4),
        optEvals = maxEvals/2 - broydenEvals;
}

```

```

pair<Vector<double>, double> result = solveByOptimization(f,
    Vector<double>(D), optEvals), result2 =
    solveBroydenHybrid(f, result.first, xEps, broydenEvals);
// выполняется ошибка оценки, если метод Бройдена ничего не дал
if(!isfinite(result2.second)) result2.second = result.second;
// для другой половины оценок выполняется случайный поиск Бройдена
result = solveBroydenLevy(f, D, xEps, maxEvals/2/1000);
if(normInf(f(result.first)) < normInf(f(result2.first)))
    result = result2;
return result;
}

```

Это, вероятно, самый безжалостный метод для функций «черных ящиков», но для стандартного решения конкретных проблем используется быстрый локальный решатель (рис. 23.7).

Функция	Broyden QR		LMBroyden		BroydenLevy		Opt		Hybrid	
	Ошибка	Оценка	Ошибка	Оценка	Ошибка	Оценка	Ошибка	Оценка	Ошибка	Оценка
ExtendedRosenbrock2	-15.65	8	-15.65	14	-15.65	33560	-13.57	926690	-13.66	253938
ExtendedPowellSingular4	-15.65	104	-14.89	1001	-15.65	84257	-13.48	950871	-15.65	280917
Trig2	-15.65	16	-14.52	49	-15.65	150843	-13.56	900835	-13.61	293819
MultiArctan2	-15.65	23	-15.65	25	-15.65	17972	-14.16	900524	-15.65	235169
HelicalValley	-12.73	198	1.70	1001	-15.65	121340	-13.70	908270	-13.95	291246
LinearFFullRank2	-15.65	7	-15.65	3	-15.65	8000	-15.65	950328	-14.75	229484
Wood	0.62	1002	-6.62	1001	-15.65	727857	-15.65	901484	-15.65	590109
GulfRND	-1.53	1000	-2.10	1001	-15.65	909270	-13.79	925174	-15.65	901813
BiggsExp6	-0.41	1000	-0.59	1001	-15.65	982939	-2.84	966911	-2.50	981325
ExtendedRosenbrock10	-15.65	24	-15.65	14	-15.65	55483	-13.21	946776	-13.45	266917
ExtendedPowellSingular12	-15.65	136	-15.65	410	-15.65	112309	-9.84	991448	-15.65	297937
Trig10	-14.17	76	-1.35	245	-15.65	635819	-13.76	905656	-1.75	803655
MultiArctan10	-15.65	39	-15.65	89	-15.65	59640	-15.65	903046	-14.59	254857
LinearFFullRank10	-15.65	23	-15.65	3	-15.65	24000	-15.65	902984	-14.23	238113
ExtendedRosenbrock100	-14.35	204	-15.65	14	-15.65	293157	-7.52	953481	-14.05	389808
ExtendedPowellSingular100	-15.65	488	-15.37	1001	-15.65	393638	-4.75	953476	-15.65	442253
Trig100	-14.18	923	-2.31	263	-14.23	953470	-2.88	944671	-3.62	968564
MultiArctan100	-14.64	219	0.16	1001	-15.65	272096	-15.65	904848	-15.65	365887
LinearFFullRank100	-15.65	203	-15.65	3	-15.65	204000	-14.33	904289	-15.35	328986
Средние ранги	2.2	1.4	2.7	1.6	1.0	3.3	3.4	4.8	2.8	3.9

**Рис. 23.7.** Эксперименты с некоторыми  $f$ . Среди локальных методов QR гораздо надежнее. Среди неоднократных глобальных методов Бройден всегда выигрывает, хотя иногда виной тому могло быть везение

Ни один из представленных методов не проверяет, дает ли найденный корень  $f$ -значение, достаточно близкое к 0. Код вызывающего объекта может сделать это с любой погрешностью. Возможно, следует проверить попадание в корень хотя бы с одинарной точностью, но погрешность таких проверок, по-видимому, сильно зависит от конкретной задачи. Также может быть существенная зависимость от масштаба, если типичные значения  $f$  велики.

## 23.14. Поиск всех корней в одномерном случае

Нахождение всех корней нормализованного многочлена  $\sum_{0 \leq i \leq n} c_i x^i$  такого, что  $c_n = 1$ , сводится к нахождению собственных значений его матрицы-компаньона, элементы поддиагонали которой = 1, элементы последнего столбца =  $-c_i$ , а остальные = 0 (см. [23.25]). Например для  $n = 3$  получаем:

$$\begin{bmatrix} 0 & 0 & -c_0 \\ 1 & 0 & -c_1 \\ 0 & 1 & -c_2 \end{bmatrix}.$$

Матрица находится в форме Хессенберга, поэтому для ограниченного числа итераций нужно  $O(n^2)$  времени и места:

```
Vector<complex<double>> findAllRoots(Vector<double> const& lowerCoefs)
{
    int n = lowerCoefs.getSize();
    Matrix<double> companion(n, n);
    for(int r = 0; r < n; ++r)
    {
        if(r > 0) companion(r, r - 1) = 1;
        companion(r, n - 1) = -lowerCoefs[r];
    }
    return QREigenHessenberg(companion);
}
```

Специальных проверок для  $x^n = 0$  не требуется (единственное решение:  $x = 0$ ), но это может быть неверно для других реализаций решателей собственных значений, поэтому будет проверяться  $\|x\|_\infty < \varepsilon$  для некоторого  $\varepsilon$ .

Поскольку нет особого смысла находить корни полиномов очень высокой степени из-за возможной плохой обусловленности с точки зрения  $c_i$  (см. [23.12]), этого решения достаточно, и оно предпочтительно на практике из-за его надежности.

Для функций «черного ящика» и конечных диапазонов полиномы Чебышева позволяют вычислить все корни, но есть некоторые проблемы надежности. Чтобы можно было использовать единственный многочлен, функция  $f$  должна быть липшицевой и сходиться. Но поскольку неизвестно, липшицева ли она, и требуется снизить стоимость оценки собственных значений, используется адаптивная интерполяция Чебышева. Кажется разумным ограничить степень небольшим числом, и мои тесты говорят, что значения 32 вполне достаточно. В работе [23.9] автор советует использовать метод секущих для полировки найденных корней, чтобы повысить их точность и удалить ложные корни:

```
template<typename FUNCTION> Vector<double> findAllRealRootsCheb(
    FUNCTION const& f, double a, double b, int maxDegree = 32,
    double duplicateXEps = highPrecEps)
{
    Vector<pair<pair<double, double>, ScaledChebAB>> pieces =
        interpolateAdaptiveHeap<IntervalCheb>(f, a, b, maxDegree).first.
        getPieces();
    PiecewiseData<EMPTY> resultFilter(duplicateXEps);
    for(int i = 0; i < pieces.getSize(); ++i)
    {
        Vector<double> rootsI = pieces[i].second.findAllRealRoots();
```

```

for(int j = 0; j < rootsI.getSize(); ++j)
{ // проверка диапазона и конечности
    double polishedRoot =
        solveSecant(f, rootsI[j], duplicateXEps).first;
    if(isfinite(polishedRoot) && a <= polishedRoot &&
        polishedRoot <= b) resultFilter.insert(polishedRoot, EMPTY());
}
}
Vector<pair<double, EMPTY> > tempResult = resultFilter.getPieces();
Vector<double> result(tempResult.getSize());
for(int i = 0; i < tempResult.getSize(); ++i)
    result[i] = tempResult[i].first;
return result;
}

```

Некоторые аналитические преобразования иногда разрешают расширение до бесконечных диапазонов, но это мы не будем рассматривать.

Поиск всех корней в сложных, многомерных или бесконечных диапазонах проблематичен из-за отсутствия хороших алгоритмов. Некоторые из них могут не иметь смысла для такой задачи — например,  $\sin(x)$  имеет бесконечно много корней, поэтому алгоритмы вроде экспоненциального поиска бессмысленны, если только вам не нужен лишь один или вы не против сдаться после проверки достаточно большого интервала.

## 23.15. Обыкновенные дифференциальные уравнения

Предположим, что  $y(x)$  — это неизвестная функция с начальным условием  $y(x_0) = y_0$  и  $y'(x) = f(x, y)$ , где  $f$  — «черный ящик». Например, рассмотрим  $f(x, y) = x + y$ . При  $y(0) = 2$  аналитическое решение имеет вид  $y(x) = 3e^x - x - 1$  (см. [23.20]). Для сравнения, при интегрировании  $f(x, y)$  является функцией только от  $x$ , что упрощает задачу.

Нам нужно решить ОДУ численно при  $x \in [x_0, b]$  для некоторого  $b$ . Для простоты предположим, что вы хотите знать только  $y(b)$ . Если нужны промежуточные значения, можно изменить для этого код либо запускать его на нескольких  $b$ .

Требуются дополнительные предположения для определения уникального пути в пространстве  $x$ – $y$ , по которому должно следовать решение (см. [23.57]). *Теорема Пикара*: пусть  $x \in [x_0, b]$  и  $y$  таковы, что  $|y - y_0|$  ограничена, а  $f$  непрерывна по  $[x_0, b]$ , липшицева по  $y$  и ограничена при  $y = y_0$  (обратите внимание, что переменная  $y$  и функция  $y$  пока не совпадают — это надо доказать). Затем в области решения  $y(x)$  существует и единственна.

Простой алгоритм решения ОДУ — *прямой алгоритм Эйлера* (часто опускается слово «прямой»):

1. Выбрать маленький размер шага  $h$ .
2. От  $x = x_0$  до момента  $x = b$ :
3.  $y \leftarrow y + hf(x, y)$ .
4. Вернуть  $y$ .

В случае интегрирования алгоритм Эйлера эквивалентен левой сумме Римана низкого порядка из-за предположения о том, что  $\Delta y = hf(x, y)$ . Но это полезно для стохастических дифференциальных уравнений, где  $y'(x) = f(x, y) + \text{шум}$ , потому что шум сводит на нет любую гладкость, используемую методами более высокого порядка.

Методы Рунге — Кутты, к которым относится алгоритм Эйлера, кратко указаны в виде таблицы (см. [23.68]):

$c$	$A$
	$b$
	$b^*$

Здесь  $y_{i+1} = y_i + \sum b_j k_j$ , где  $k_j = hf(x_i + c_{jk}, \sum A[l, j]k_l)$ . Необязательный параметр  $b^*$  определяет встроенную оценку ошибки  $y_{i+1}(b) - y_{i+1}(b^*)$ . Методы, в которых только нижняя треугольная часть  $A \neq 0$  и  $c_0 \neq 0$  считаются *явными*, т. к. их можно вычислить напрямую. Остальные *неявны* из-за того, что зависят от значений, «вычисляемых в будущем», и определяются итерацией с фиксированной точкой. Любые 0 записей обычно не указываются в таблице, хотя они могут быть здесь для иллюстрации. Например, для алгоритма Эйлера таблица выглядит так:

0	0
	1

Убедите себя, что она алгебраически эквивалентна более простой формуле. Оценить  $f$  можно только при выполнении шагов, поэтому можно напрямую моделировать подшаги, применяя алгоритм Эйлера. В частности, применение правила Симпсона приводит к методу Рунге — Кутты 4-го порядка:

0				
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

Код, следующий далее, приведен лишь для иллюстрации того, как реализуется шаг, основанный на такой таблице. Более эффективный метод обсуждается позже.

```
template<typename TWO_VAR_FUNCTION>
double RungeKutta4Step(TWO_VAR_FUNCTION const& f, double x, double y,
    double h, double f0 = numeric_limits<double>::quiet_NaN())
{
    if(isnan(f0)) f0 = f(x, y);
    double k1 = h * f0, k2 = h * f(x + h/2, y + k1/2),
        k3 = h * f(x + h/2, y + k2/2), k4 = h * f(x + h, y + k3);
    return y + (k1 + 2 * k2 + 2 * k3 + k4)/6;
}
```



Одношаговые методы, такие как метод Эйлера и метод Рунге — Кутты, делают один шаг размером  $h$ . Ошибка усечения шага равна  $T_n = \frac{y_{i+1} - y_i}{h} - \frac{y(x_{i+1}) - y(x_i)}{h}$  (т. е. приращению по сравнению с правильным приращением). Теорема (см. [23.57]): пусть  $T = \max |T_i|$ , и предположим, что уравнение удовлетворяет условиям теоремы Пикара. Тогда глобальная ошибка равна  $O(T)$ . Постоянный фактор гораздо более пессимистичен, чем, например, фактор интегрирования Симпсона.

Применяя это к методу Рунге — Кутты, получим, что ошибка в  $y(b)$  равна  $O(h^4)$ . В случае для Эйлера она равна  $O(h)$ . Интуитивное понимание ошибки несколько отличается от интегрирования: аппроксимация отдельного шага + ошибка оценки влияют на  $y_0$ , отправляя решение на другой непрерывный путь, который, как мы надеемся, близок к исходному.

Метод *непротиворечив*, если  $T \rightarrow 0$  при  $n \rightarrow \infty$ . ОДУ является хорошо обусловленным, если небольшое возмущение  $y_0$  приводит только небольшому возмущению  $y_b$ . Условия теоремы Пикара подразумевают, что абсолютное число обусловленности конечно. Теорема (см. [23.12]): одношаговый метод сходится, если он непротиворечив и число обусловленности ОДУ конечно. Интуитивно понятно, что на каждом шаге не должно быть слишком много ошибок, а путь решения не должен значительно усугублять ошибки. Итак, учитывая, что в первую очередь должны выполняться условия теоремы Пикара, для сходимости требуется только непротиворечивость. Скорость  $O(h^4)$  на практике может быть достаточно хороша, но адаптивные шаги более эффективны (хотя коды с фиксированным  $n$  хороши для построения графиков решений, если они настроены на возврат всех промежуточных значений  $y$ ). Адаптивность на основе кучи невозможна, потому что интервалы зависимы, поэтому существует эквивалент адаптивности на основе рекурсии/стека, но где результат левого вызова используется правым вызовом. Некоторые подробности:

- ◆ в начале берется малое  $h$ ;
- ◆ отслеживайте локальные ошибки. Разные варианты могут привести к нескольким разным алгоритмам;
- ◆ пока локальная ошибка текущего шага оказывается приемлема и не достигнут некоторый предел, снижаем  $h$  вдвое;
- ◆ когда шаг делается, немного увеличиваем  $h$ , если малые  $h$  больше не нужны;
- ◆ придерживаемся  $h \in [h_{\min}, h_{\max}]$ . Первое нужно для эффективности, а второе — для сходимости, т. е. должно быть  $h_{\max} \rightarrow 0$ , а количество интервалов  $\rightarrow \infty$ . Слишком большие шаги сокращаются, а слишком маленькие автоматически принимаются и увеличиваются. По умолчанию  $h_{\max} = h_{\min}^2$ ;
- ◆ правую границу можно обработать, взяв точный шаг вблизи к ней, не перепрыгивая;
- ◆ в качестве оценки ошибки можно вернуть сумму локальных ошибок. Это эвристика, но, как правило, она полезна для прогнозирования истинной ошибки с точностью до порядка. Истинная ошибка может быть больше линейной ошибки, если константа Липшица для этой задачи велика (как это предсказывается теоремами), а предыдущие ошибки увеличиваются;

◆ при заданной погрешности  $\varepsilon$  нужно вычислить

$$\max \left( \text{некоторая высокая точность}, \varepsilon \sqrt{\frac{h}{b-x_0}} \right).$$

Пессимистично иметь сумму локальных ошибок  $\leq \varepsilon$ , потому что ошибки аннулируются при превышении или занижении оценки  $y$ . Итак, если верна теорема о центральном пределе и  $n = (b - x_0) / h$ ,  $\sqrt{n} \sum$  локальных ошибок  $\leq \varepsilon$ , т. е. достаточно, чтобы для одного интервала локальная ошибка  $\leq \varepsilon / \sqrt{n}$ . Высокая точность нужна только в качестве меры предосторожности.

Простой адаптивный оценщик ошибок сравнивает один шаг с двумя полушагами. Затем делаются полушаги, если принимается значение  $h$ . Это работает с любым пошаговым методом.

Но лучшим алгоритмом является *метод Дорманда — Принса* (см. [23.22, 23.51]). Он имеет пятый порядок и является методом по умолчанию в MATLAB (см. [23.55]). В работе [23.29] приводится следующая таблица:

0							
1/5	1/5						
3/1	3/40	9/40					
0							
4/5	44/45	-56/15	32/9				
8/9	19372/6561	-25360/2187	64448/6561	-212/729			
1	9017/3168	-355/33	46732/5247	49/176	-5103/18656		
1	35/384	0	500/1113	125/192	-2187/6784	11/84	
	35/384	0	500/1113	125/192	-2187/6784	11/84	0
	5179/57600	0	7571/16695	393/640	-92097/339200	187/2100	1/40

Контроль адаптивности здесь выполняется немного проще, потому что не требуется вычислять полушаги. Хитрость заключается в *FSAL*: последняя оценка  $f$  на последнем шаге = первая оценка  $f$  на следующем шаге:

```
template<typename TWO_VAR_FUNCTION> pair<pair<double, double>, double>
RungeKuttaDormandPrinceStep(TWO_VAR_FUNCTION const& f, double x, double y,
double h, double f0)
{
double k1 = h * f0,
k2 = h * f(x + h/5, y + k1/5),
k3 = h * f(x + h * 3/10, y + k1 * 3/40 + k2 * 9/40),
k4 = h * f(x + h * 4/5, y + k1 * 44/45 + k2 * -56/15 + k3 * 32/9),
k5 = h * f(x + h * 8/9, y + k1 * 19372/6561 + k2 * -25360/2187 +
k3 * 64448/6561 + k4 * -212/729),
k6 = h * f(x + h, y + k1 * 9017/3168 + k2 * -355/33 + k3 * 46732/5247 +
k4 * 49/176 + k5 * -5103/18656),
yNew = y + k1 * 35/384 + k3 * 500/1113 + k4 * 125/192 +
k5 * -2187/6784 + k6 * 11/84, f1 = f(x + h, yNew),
k7 = h * f1;
```

```

    return make_pair(make_pair(yNew, f1), y + k1 * 5179/57600 + k3 * 7571/16695
        + k4 * 393/640 + k5 * -92097/339200 + k6 * 187/2100 + k7/40);
}

template<typename TWO_VAR_FUNCTION> pair<double, double>
adaptiveRungeKuttaDormandPrince(TWO_VAR_FUNCTION const& f, double x0,
double xGoal, double y0, double localERelAbs = defaultPrecEps,
int maxIntervals = 100000, int minIntervals = -1, int upSkip = 5)
{
    if(minIntervals == -1) minIntervals = sqrt(maxIntervals);
    assert(xGoal > x0 && minIntervals > 0 && upSkip > 0);
    double hMax = (xGoal - x0)/minIntervals, hMin = (xGoal - x0)/maxIntervals,
        linearError = 0, h1 = hMax, y = y0, f0 = f(x0, y);
    bool last = false;
    int stepCounter = 0;
    for(double x = x0; !last;)
    {
        if(x + h1 > xGoal)
        { // последний шаг делается точным
            h1 = xGoal - x;
            last = true;
        }
        pair<pair<double, double>, double> yfye =
            RungeKuttaDormandPrinceStep(f, x, y, h1, f0);
        double h2 = h1/2, xFraction = h1/(xGoal - x0);
        if(h2 < hMin || isEEqual(yfye.first.first, yfye.second,
            max(highPrecEps, localERelAbs * sqrt(xFraction))))
        { // принимаемый шаг
            x += h1;
            y = yfye.first.first;
            f0 = yfye.first.second; // повторное использование последней оценки
            linearError += abs(y - yfye.second);
            if(++stepCounter == upSkip && h2 >= hMin)
            { // использование большего шага после нескольких принятых
                h1 = min(hMax, h1 * 2);
                stepCounter = 0;
            }
        }
        else
        { // использование половинного шага
            h1 = h2;
            last = false;
            stepCounter = 0;
        }
    }
    return make_pair(y, linearError);
}

```

Распространение алгоритма на системы дифференциальных уравнений просто. Основное отличие состоит в том, что для сходимости используется  $\|y_1 - y_2\|_\infty$  (рис. 23.8).

Функция	Euler10k			DoublingRK4			DP		
	Ошибка	Прибл.	Оценка	Ошибка	Прибл.	Оценка	Ошибка	Прибл.	Оценка
Ступенчатая	-3.699	0.000	10000	-5.823	-5.746	512927	-5.490	-6.379	243739
Модуль	-13.276	0.000	10000	-14.463	-14.347	3476	-14.477	-15.654	1897
Линейная	-3.699	0.000	10000	-15.654	-14.765	3476	-15.654	-15.654	1897
Квадратичная	-7.875	0.000	10000	-14.593	-14.353	3476	-14.574	-15.654	1897
Кубическая	-3.699	0.000	10000	-15.654	-14.710	3476	-15.605	-15.654	1897
4-й степени	-7.574	0.000	10000	-11.777	-10.601	3476	-14.622	-12.074	1897
Экспонента	-4.000	0.000	10000	-13.481	-12.282	3476	-14.946	-13.736	1897
SqrtAbs	-6.056	0.000	10000	-9.195	-7.849	446959	-8.590	-9.026	243751
Рунге	-9.973	0.000	10000	-9.213	-7.688	4105	-9.264	-8.420	2005
Логарифм	-4.460	0.000	10000	-14.251	-13.241	3476	-14.750	-14.747	1897
XSinXM1	-5.345	0.000	10000	-6.897	-5.501	300684	-7.138	-6.900	245983
Синус	-3.774	0.000	10000	-15.487	-12.319	3476	-15.654	-13.794	1897
AbsIntegral	-4.000	0.000	10000	-15.654	-14.795	3476	-15.654	-15.654	1897
Тангенс	-3.817	0.000	10000	-15.654	-11.562	3476	-15.654	-13.036	1897
Нормальная PDF	-15.654	0.000	10000	-15.176	-7.835	3476	-15.353	-9.308	1897
Дельта PDF	-15.176	0.000	10000	-8.623	-8.238	355785	-8.577	-9.830	255793
F575	-0.501	0.000	10000	-0.989	-1.624	889912	-0.975	-3.519	485425
Средние ранги	2.647	3.000	2.294	1.647	2.000	2	1.529	1.000	1

Рис. 23.8. Производительность методов на некоторых  $f$ .

Оценка ошибки Дорманда — Принса кажется менее надежной, чем ошибка удвоения.

Таким образом, предположения формулы ошибки не очень точны, но на практике это не имеет значения

## 23.16. Решение жестких ОДУ

Некоторые ОДУ — *жесткие*. Например, как говорится в работе [23.57]),  $y' = ay$  с  $y_0 = 0$  имеет решение  $y(x) = y_0 e^{ax}$ . Для  $a > 0$  решение растет неограниченно, поэтому интересен случай  $a \leq 0$ , где  $y(x) \rightarrow 0$ . Используя метод Эйлера с  $n$  шагами размером  $h = x/n$ , получаем  $y_n = y_0 (1 + ah)^n$ . Чтобы избежать экспоненциального взрыва, требуется соблюдение условия  $-2 > ha > 0$ . В жестком случае необходимо использовать гораздо меньшее  $h$ , чем то, что в противном случае потребовалось бы для желаемой точности.

Для линейной системы  $y' = Ay$ , матрица  $A$  имеет различные собственные значения  $\lambda_i$  (предположим, что выполняются технические условия), и для многих алгоритмов можно получить *области абсолютной устойчивости*, т. е. подмножества комплексной плоскости, где значения  $z = h\lambda_i$  не вызывают взрыв, упомянутый ранее. Метод является **A-устойчивым**, если его область абсолютной устойчивости содержит левую комплексную полуплоскость  $\text{Re}(z) \leq 0$ .

Функция *устойчивости*  $R(z)$  метода для одного шага такова, что  $y_1 = R(z)y_0$  (см. [23.30]). Например, для метода Эйлера  $R(z) = 1 + z$ . Для A-устойчивости должно соблюдаться условие  $|R(z)| \leq 1$  для  $\text{Re}(z) \leq 0$ . Это приводит к *L-устойчивости* (см. [23.30]): A-стабильность выполняется, и  $\lim_{\text{Re}(z) \rightarrow -\infty} R(z) = 0$ . В некоторых источниках используется альтернативное, эквивалентное определение L-устойчивости («L» значит «левый»), которое требует, чтобы  $\lim_{z \rightarrow -\infty} R(z) = 0$ . Эти определения эквивалентны, потому что  $R$  является рациональной функцией для интересующих нас случаев, и  $R(\infty) = R(-\infty)$ . На практике иногда A-устойчивости недостаточно, но достаточно L-устойчивости.

L-устойчивость приводит к конечному «кругу» устойчивости  $|R(z)| \leq 1$ . Для  $a > 0$  и некоторого  $h$  это обеспечит устойчивость, поскольку решение неограниченно растет. Обычно это не проблема, но в работе [23.41] приводятся примеры того, когда это происходит.

У жестких уравнений явные ступенчатые методы неустойчивы, поэтому нужно использовать неявные. Самый простой метод неявного шага — это *обратный метод Эйлера*, т. е. решение уравнения  $y_{i+1} = y_i + hf(x + h, y_{i+1})$ . Можно использовать метод Бройдена, где  $y_i$  будет выступать в качестве начального значения для  $y_{i+1}$ . Таблица для этого:

1	1
	1

Формула и таблица имеют тот же вид:  $y_{i+1} = y_i + k_0$ , а замена  $k_0$  в уравнениях таблицы приводит к обратному уравнению Эйлера.

Но работоспособность метода Бройдена или другого универсального решателя не гарантируется, хотя обычно они работают нормально. ОДУ гарантирует лишь то, что решение будет близко к начальному значению и существует, поэтому использование локального решателя оправданно. Но решение может быть недостаточно локальным или иметь плохие свойства. В качестве дополнительной защиты можно уменьшать  $h$  в случае сбоя решателя, но не факт, что это поможет.

Формула более высокого порядка называется *правилом трапеций*, и оно аналогично правилу интегрирования, заданному формулой  $y_{i+1} = y_i + \frac{h}{2}(f(x, y_i) + f(x + h, y_{i+1}))$ .

Это правило обладает только A-устойчивостью, а обратный метод Эйлера L-устойчив (см. [23.30]). Зато этот метод важен для решения PDE (это мы кратко рассмотрим позже в этой главе).

Для обратного метода Эйлера кажется, что это, по всей видимости, не имеет значения, но уравнения следует решать в терминах базисных переменных  $fSum$ , а не в терминах  $y$ . Из-за округления в выражении « $y_i + h(fSum)$ », в предположении, что  $y$  и  $fSum$  имеют примерно одинаковую величину, после масштабирования и сложения некоторая точность  $fSum$  теряется. Итак, если  $y_i$  имеет точность  $\epsilon$ , то бессмысленно повышать точность  $fSum$  дальше  $\frac{\epsilon \|y_i\|_\infty}{h}$ . Но  $fSum$ , вычисленная с такой точностью, позволяет вычис-

лить  $y_i$  с точностью до  $\epsilon$ , что лучше, чем напрямую вычислять  $y_{i+1}$  с точностью до  $\epsilon$ . Тем не менее если  $fSum$  намного больше, чем  $y$ , задача плохо масштабируется, и эта формула не будет работать должным образом, поэтому следует использовать указанную точность  $\|y_i\|_\infty$ . Хорошее начальное значение равно 0, потому что  $y_i$  является хорошим начальным значением для  $y_{i+1}$ .

Каким бы ни было правило определения шага, адаптивный шаг на основе деления  $h$  пополам выглядит так:

```
template<typename YX_FUNCTION>
Vector<double> evalYX(double x, Vector<double> const& y, YX_FUNCTION const& f)
{ // предполагается, что последний аргумент - это x
  Vector<double> yAugmented = y;
```

```

    yAugmented.append(x);
    Vector<double> fAugmented = f(yAugmented);
    fAugmented.removeLast();
    return fAugmented;
}

template<typename YX_FUNCTION, typename STEP_F> pair<Vector<double>, double>
adaptiveStepper(YX_FUNCTION const& f, STEP_F const& s, double x0,
double xGoal, Vector<double> y0, double localERelAbs = defaultPrecEps,
int maxIntervals = 100000, int minIntervals = -1,
double upFactor = pow(2, 0.2))
{ // предполагается, что f0 повторно не используется
    if(minIntervals == -1) minIntervals = sqrt(maxIntervals);
    assert(xGoal > x0 && minIntervals > 0 && upFactor > 1);
    int D = y0.getSize();
    double hMax = (xGoal - x0)/minIntervals, hMin = (xGoal - x0)/maxIntervals,
        linearError = 0, h1 = hMax;
    Vector<double> y = y0,
        y1 = Vector<double>(D, numeric_limits<double>::quiet_NaN()), f0;
    bool last = false;
    for(double x = x0; !last;)
    {
        if(x + h1 > xGoal)
        { // последний шаг выполняется точно
            h1 = xGoal - x;
            last = true;
        }
        double h2 = h1/2, xFraction = h1/(xGoal - x0),
            tolERelAbs = max(highPrecEps, localERelAbs * sqrt(xFraction)),
            solveERelAbs = tolERelAbs/10;
        if(isnan(normInf(y1))) y1 = s(f, x, y, h1, solveERelAbs);
        Vector<double> y2 = s(f, x, y, h2, solveERelAbs),
            firstY2 = y2;
        y2 = s(f, x + h2, y2, h2, solveERelAbs);
        double normError = normInf(y2 - y1), normY2 = normInf(y2);
        if(h2 < hMin || isEEEqual(normY2 + normError, normY2, tolERelAbs))
        { // шаг принимается
            x += h1;
            y = y2;
            linearError += normError;
            y1 = Vector<double>(D, numeric_limits<double>::quiet_NaN());
            if(h2 >= hMin) h1 = min(hMax, h1 * upFactor); // используется больший шаг
        }
        else
        { // используется половина шага
            y1 = firstY2;
            h1 = h2;
            last = false;
        }
    }
    return make_pair(y, linearError);
}

```

Метод *Радану IIА* пятого порядка хорош как в теории (L-устойчивость по [23.30]), так и на практике (см. сравнения в работе [23.30]). Это выглядит так:

$(4 - \sqrt{6})/10$	$(88 - 7\sqrt{6})/360$	$(296 - 169\sqrt{6})/1800$	$(-2 + 3\sqrt{6})/225$
$(4 + \sqrt{6})/10$	$(296 + 169\sqrt{6})/1800$	$(88 + 7\sqrt{6})/360$	$(-2 - 3\sqrt{6})/225$
1	$(16 - \sqrt{6})/36$	$(16 + \sqrt{6})/36$	1/9
	$(16 - \sqrt{6})/36$	$(16 + \sqrt{6})/36$	1/9

Метод основан на *квадратуре Гаусса* — *Радану* и является FSAL:

```
template<typename YX_FUNCTION> struct RadauIIA5Function
{
    Vector<double> y;
    double x, h;
    YX_FUNCTION f;
    Vector<double> operator() (Vector<double> fSumDiffs) const
    {
        assert(fSumDiffs.getSize() % 3 == 0);
        int D = fSumDiffs.getSize()/3;
        double s6 = sqrt(6), ci[3] = {(4 - s6)/10, (4 + s6)/10, 1}, A[3][3] =
        {
            {(88 - 7 * s6)/360, (296 - 169 * s6)/1800, (-2 + 3 * s6)/225},
            {(296 + 169 * s6)/1800, (88 + 7 * s6)/360, (-2 - 3 * s6)/225},
            {(16 - s6)/36, (16 + s6)/36, 1.0/9}
        };
        Vector<Vector<double> > ki(3);
        for(int i = 0; i < 3; ++i)
        {
            Vector<double> yi(y);
            for(int j = 0; j < D; ++j) yi[j] += h * fSumDiffs[i * D + j];
            ki[i] = evalYX(x + h * ci[i], yi, f);
        }
        for(int i = 0; i < 3; ++i)
        {
            Vector<double> fSumi(D, 0);
            for(int j = 0; j < 3; ++j) fSumi += ki[j] * A[i][j];
            for(int j = 0; j < D; ++j) // преобразование фиксированной точки в 0
                fSumDiffs[i * D + j] = fSumi[j] - fSumDiffs[i * D + j];
        }
        return fSumDiffs;
    }
};

struct RadauIIA5StepF
{
    template<typename YX_FUNCTION> Vector<double> operator() (
        YX_FUNCTION const& f, double x, Vector<double> y, double h,
        double solveERelAbs) const
    {
        RadauIIA5Function<YX_FUNCTION> r5f = {y, x, h, f};
        int D = y.getSize();
```

```

Vector<double> fSumDiffs(3 * D, 0);
fSumDiffs = solveBroyden(r5f, fSumDiffs, max(solveERelAbs,
    numeric_limits<double>::epsilon() * normInf(y)/h)).first;
for(int j = 0; j < D; ++j) y[j] += h * fSumDiffs[2 * D + j];
return y;
}
};

```

Несмотря на 5-й порядок, из-за ограничений решения нелинейных уравнений выход за пределы локальной точности  $10^{-12}$  или подобный видится неэффективным. Вероятно, это связано с тем, что функция  $f$  в случае жестких уравнений несколько чувствительна к неточным значениям (см. [23.30]), и даже работа с базисом  $fSum$  имеет свои ограничения. Адаптивные стратегии не позволяют распознать случаи, когда уменьшение  $h$  не приносит дополнительной пользы (и непонятно, как это сделать эффективно), но метод в моих экспериментах Радау ПА с фиксированным  $h$  позволял достичь точности  $10^{-10}$  всего за 100 шагов. По всей видимости, не существует встроенного метода Радау ПА, такого как метод Дорманда — Принса, поэтому используйте деление шага пополам.

Использование Дорманда — Принса для нежестких уравнений и Радау ПА для жестких уравнений — это хороший вариант по умолчанию — не обязательно лучший, но и не плохой (хотя для больших систем Радау ПА может требовать слишком много памяти).

Для умеренно жестких ОДУ решение неявных уравнений не гарантируется, но нежесткий адаптивный решатель может оказаться эффективным и надежным. Таким образом, разумная стратегия «черного ящика» состоит в том, чтобы попробовать его для некоторого количества оценок и переключиться на жесткий решатель, если полученная в результате точность окажется слишком мала. На следующем этапе этот выбор было бы хорошо выполнять автоматически, но это не рассматривается далее.

Если  $f$  не такая гладкая, как предполагалось, порядок сходимости будет ограничен количеством непрерывных производных, как и в случае интерполяции (хотя в литературе, похоже, на этот счет нет никаких теорем). Когда это число невелико, методы высокого порядка впустую тратят вычислительное время, потому что методы низкого порядка, такие как метод Эйлера, получают тот же порядок за меньшее количество вычислений. Но потери ограничены небольшим постоянным коэффициентом, поэтому в общем случае методы более высокого порядка представляют собой наилучший компромисс как в плане использования высокого порядка, так и в устранении неэффективности, где это возможно.

Представленные явные и неявные реализации метода Рунге — Кутты на текущий момент являются наиболее актуальными, но хорошие библиотечные реализации выглядят сложнее.

## 23.17. Краевые задачи для ОДУ

Начальные условия ОДУ позволяют выиграть время. Более общая задача состоит в том, чтобы определить условия для начальной и конечной точек. В стандартной форме есть функция  $g$ , такая как  $g(y_0, y_b) = 0$  (см. [23.20, 23.32, 23.51]). Распространенным решением является приведение к ОДУ *методом стрельбы*. В формулировке итерации с фиксированной точкой он выглядит так:



1. Получить  $y_0$  из начального предположения для граничных условий.
2. Решить полученное ОДУ для  $y_b$ .
3. Отрегулировать предположение граничного условия и повторить вычисление.

Решение не существует, если у функции нет корня, и в дополнение к условиям теоремы Пикара нужны дополнительные условия, чтобы гарантировать, что решатель ОДУ не сломается для произвольного производного от начального предположения  $y_0$ . Эти вопросы здесь больше не обсуждаются — см. [23.24 и 23.57]. Наиболее полная и подробная книга по этой теме — [23.2].

В простейшем случае, который похож на стрельбу из пушки путем выбора угла обстрела, имеется единственный неизвестный параметр, и тогда вы можете использовать метод секущих для нахождения корней  $g$ . Нужен какой-то полезный способ указать результат, потому что в этом случае нам требуется некоторое количество внутренних точек. Самый простой способ сделать это с помощью адаптивного решателя ОДУ — сначала найти параметры граничных условий. Затем запускается второй раунд, где заданное количество точек интервала решается в каждом интервале, а конечный результат рассматривается как начальное условие для следующего интервала:

```
template<typename YX_FUNCTION, typename BOUNDARY_FUNCTION>
struct BoundaryFuncor
{
    YX_FUNCTION const& f;
    BOUNDARY_FUNCTION const& bf;
    double x0, xGoal;
    double operator() (double b) const
    {
        return bf.evaluateGoal(adaptiveStepper(f, RadauIIA5StepF(),
            x0, xGoal, bf.getInitial(b)).first);
    }
};

template<typename YX_FUNCTION, typename BOUNDARY_FUNCTION>
Vector<Vector<double>> > solveBoundaryValue(YX_FUNCTION const& f, double x0,
double xGoal, Vector<double> const& xPoints, BOUNDARY_FUNCTION const& bf,
double b0 = 0)
{
    BoundaryFuncor<YX_FUNCTION, BOUNDARY_FUNCTION> fu = {f, bf, x0, xGoal};
    double bFound = solveSecant(fu, b0).first;
    Vector<Vector<double>> > result;
    if (isfinite(bFound))
    {
        Vector<double> y0 = bf.getInitial(bFound);
        for (int i = 0; i < xPoints.getSize(); ++i)
        {
            y0 = adaptiveStepper(f, RadauIIA5StepF(), x0, xPoints[i],
                y0).first;
            x0 = xPoints[i];
            result.append(y0);
        }
    }
    return result;
}
```

В случае нескольких параметров используют многомерные методы поиска корней. Кроме того, может потребоваться настроить код так, чтобы разрешить более быстрые нежесткие решатели и передачу им параметров. Некоторые решатели ОДУ имеют параметр плотного вывода (см. раздел комментариев), который можно использовать вместо того, чтобы запрашивать решение в конечном числе точек.

## 23.18. Уравнения с частными производными: некоторые размышления

УЧП являются обобщением ОДУ и краевых задач. Дополнительная сложность позволяет эффективно решать УЧП типа «черный ящик», поэтому большинство методов обычно работают с линейными УЧП, которые подразделяются на *параболические*, *гиперболические* и *эллиптические* (см. [23.32]). Одной из полезных моделей является *уравнение теплопроводности* — функция  $u$  от временной переменной  $t$  и одномерной пространственной переменной  $x$ , определяемая как:

- ◆ уравнение  $u_t = cu_{xx}$ ;
- ◆ пространство предметной области:  $0 \leq x \leq L$ ;
- ◆ начальное условие времени:  $u(0, x) = f(x)$ ;
- ◆ левая граница пространства:  $u(t, 0) = a$ ;
- ◆ правая граница пространства:  $u(t, L) = b$ .

Самое простое решение называется *методом прямых*. Он заключается в дискретизации пространственной переменной  $x$  с помощью формулы конечных разностей и решении полученной системы многих уравнений с помощью решателя ОДУ, настроенного на возврат значений в нужные моменты времени. Например, для уравнения теплопроводности с использованием центральной разности 2-й производной получается система уравнений:

$$y_i^{(2)}(t) = c \frac{y_{i+1}'(t) - 2y_i'(t) + y_{i-1}'(t)}{\Delta x^2} \quad \text{для } y_i(t) = u(t, x_i).$$

Метод работает за счет известной формы уравнения.

Такие системы могут быть жесткими и большими, поэтому использование  $> 1000$  пространственных точек может замедлить вычисления. К тому же системы разрежены, поэтому это тоже можно использовать. Метод работает для  $D > 1$  аналогично, но оказывается более экономичным при дискретизации из-за проклятия размерности. При использовании всех методов для  $D > 1$  и особенно для  $D > 2$  требуется большое внимание к вычислительной осуществимости.

Можно также воспользоваться одним из следующих способов:

1. *Методы конечных разностей* — использовать аппроксимации конечной разности производных для преобразования задачи в обычно полосчатую или разреженную систему линейных или нелинейных уравнений.
2. *Методы конечных элементов* — предполагается, что решение задается в терминах базисных функций с локальной поддержкой, таких как  $B$ -сплайны (обсуждаются в комментариях), и система решается для коэффициентов представления путем со-

поставления значений производной и граничного условия. Результатом является решение системы.

3. *Спектральные методы* — аналогично конечному элементу, но с использованием полиномиального базиса Чебышева.

Метод (1) является самым простым, (2) — допускает наиболее гибкие граничные условия и (3) — дает очень высокую точность при работе, причем каждый из них имеет свою нишу в определенных областях. Существует много теоретической и практической литературы по всем этим методам, и большая часть ее появилась недавно, поэтому выбрать лучшую книгу из представленных я не берусь. Начать можно с [23.32 и 23.51]. Подробный список литературы по УЧП приведен в работе [23.24]. Для удобства я преобразовал его в список идей Amazon: <http://a.co/7Zrtmj1>.

## 23.19. Некоторые выводы

Еще раз подчеркнем, что численные алгоритмы не являются полноценными решениями. Любой алгоритм рассматривает определенные вопросы:

- ◆ достигается ли сходимость во всех случаях? — базовый алгоритм должен сходиться, по крайней мере, при некоторых известных аналитических условиях. Разного рода металогики, такая как эвристический выбор параметров или адаптивное управление, обычно не работает. В случае детерминированных алгоритмов следует подумать о том, имеют ли смысл ответ и оценки, если используются одни и те же точки оценки;
- ◆ устойчив ли алгоритм? — а он должен быть устойчив, если для обратного нет достаточных причин;
- ◆ является ли алгоритм со всей металогикой надежным во всех случаях? — обычно нет. Но он должен быть надежен в большинстве случаев, а остальные случаи должны быть известны. Некоторые алгоритмы, такие как прямые матричные манипуляторы, БПФ и решение уравнений бинарного поиска, очень надежны и могут считаться «черным ящиком». Если алгоритм не очень надежен, то нужно знать, по крайней мере, четко определенное подмножество, в котором известно, что алгоритм точно работает;
- ◆ является ли задача, которую вы решаете, хорошо обусловленной? Если да, нужно определить число состояний теоретически или оценить его численно. Если оно велико, а алгоритм стабилен, можно попытаться использовать его для решения. Но не следует ожидать слишком многого, и, возможно, вместо этого нужно будет решить другую проблему. Лишь несколько типов задач гарантируют хорошую обусловленность во всех случаях — например, абсолютное условие интегрирования;
- ◆ существует ли надежная оценка или граница прямой ошибки? Обычно существуют эвристические оценки, которые ненадежны, но имеют хороший порядок величин;
- ◆ был ли алгоритм тщательно протестирован на наборе задач? Ведь численные алгоритмы, в отличие от обычных логических, имеют гораздо больше различных случаев. Простое изменение знака при обычном однократном тестировании может проскочить незамеченным, поэтому необходимо настроить хорошее автоматическое регрессионное тестирование, чтобы убедиться, что вы избегаете хотя бы глупых

ошибок и рассматриваете все особые случаи, которые возможно охватить. Простая эвристика — проверить ответ с одинарной точностью. Она работает всегда.

Таким образом, никакие численные результаты не должны приниматься как должное, и когда они используются в научных исследованиях, ожидается, что их внимательно анализируют.

## 23.20. Примечания по реализации

Каждая отдельная реализация выполнена «по учебнику», но некоторые принятые в них решения оригинальны. Каждый алгоритм исследовался в течение значительного времени. Большинство из них кажутся надежными, но у меня есть сомнения по поводу ограниченной памяти метода Бройдена.

В реализации метода Дорманда — Принса я ошибочно набрал 64448/6561 как 64448/6581». Из-за схожести цифр это было трудно заметить. Процесс отладки был следующим:

1. Понять, что ошибка метода слишком велика.
2. Проверить порядок и понять, что для этой задачи он равен от 2 до 3.
3. Написать простую неадаптивную программу и понять, что она имеет большую ошибку, чем 4-й порядок Рунге — Кутты.
4. Найти отработанный пример и обнаружить некоторую неточность в вычислении  $k_5$ .

## 23.21. Комментарии

Моя реализация быстрого преобразования Фурье с  $O(n \lg(n))$ , вероятно, самая простая, но не самая быстрая в постоянных множителях. Идея разложения по степени двойки распространяется и на другие составные числа, поэтому более быстрый алгоритм будет примерно таким:

1. Вычислить простую факторизацию  $n$  ( $O(n)$  с помощью решета Эратосфена — см. главу 12. *Разные алгоритмы и методы*).
2. Выполнять рекурсию с использованием составного БПФ до тех пор, пока не будут достигнуты простые множители.
3. Решать задачи на простые множители, используя алгоритмы простых чисел.

Подробнее почитать о современной реализации можно в работе [23.21], где много логики тратится на выбор наилучшего алгоритма для минимизации постоянных факторов с учетом как  $n$ , так и аппаратного обеспечения.

В случае полиномиальной оценки Чебышева более стабильным способом является использование специализированной барицентрической формулы  $O(n)$ , в которой выражения для весов упрощены (см. [23.61]). Используйте инверсию ДКП для обратного отображения на точки данных и эту формулу (реализовано в Chebfun, может дополняться нулями до степени двойки для эффективности ДКП и сопоставляться с «псевдоточками», к которым применяется специализированная формула, но алгоритм Ченшоу работает за  $O(n)$  и достаточно стабилен.

Мы также не обсудили *интерполяцию Эрмита*, в которой также есть значения  $f'$ , полезные в некоторых специальных приложениях.

У кусочных многочленов проблема в том, что производные не являются непрерывными в узлах и в некоторых приложениях требуется гладкость  $f'$  и, возможно,  $f''$ . В этом случае чаще всего выбираются кубические сплайны — в основном из-за механической интерпретации. И скорость, и ускорение у них непрерывны, поэтому у функции нет резких движений. В работе [23.20] приведены примеры, а в работе [23.14] описана теория. Но варианты использования этого метода ограничены:

- ◆ наблюдаемая непрерывность подвержена ошибкам с плавающей точкой, и наблюдаемые различия производных по сравнению с кусочными полиномами могут быть невелики;
- ◆ в численном случае полиномы и кусочные полиномы работают лучше — например, для вычисления сплайна нужны все данные для решения, но пользователей интересует обычно только сопредельный кусок. Можно возразить, что, возможно, некоторым приложениям требуется оценка первой и второй производных, где должна обеспечиваться непрерывность оценок, но это случается редко, и в любом случае будут использоваться специальные методы, такие как нестандартные сплайны;
- ◆ работа со сплайнами является многомерной и использует численно устойчивые *B-сплайны* и обобщение для кривых и поверхностей. Используемые методы обычно зависят от приложения. Например, в графике принято представлять поверхности в 2D- или 3D-виде, и для этого было разработано множество специальных методов. Некоторые многомерные обобщения полезны для решения УЧП. Сплайны в основном применяются именно к этой категории задач;
- ◆ некоторые методы регрессии выполняют аппроксимацию (а не интерполяцию) с использованием *B-сплайнов* со штрафом за гладкость, если точки определения (называемые *узлами*) не соответствуют точкам данных (см. [23.31]). Но, в отличие от многих других методов (см. главу 27. *Машинное обучение: регрессия*), сплайны не распространяются на большие  $D$ , а в одном измерении используются другие методы;
- ◆ для построения графиков в MATLAB по умолчанию используется линейная интерполяция, потому что другие опции, такие как кубические сплайны, могут привести к колебаниям (см. [23.46]). Рекомендую взглянуть на программу *rschip*, которая использует сплайны Эрмита для обеспечения монотонности, возможно, для более визуально приятного вида, но преимущества такой плавности сомнительны, а графика в этой книге не рассматривается). В большинстве финансовых графиков для соединения рыночных значений используются прямые линии.

Использование полиномов Чебышева является лишь одним из нескольких хороших способов выполнения несплайновой интерполяции:

- ◆ *интерполяция рациональных дробей* иногда работает лучше — в работе [23.61] приведены некоторые конкретные случаи, такие как отсутствие дифференцируемости, бесконечные диапазоны или необходимость более точной оценки по фиксированному числу известных значений  $f$ . Но алгоритмы намного сложнее. Согласно [23.9] рациональная интерполяция «определено возможна», т. е. в будущем она может стать более стабильной, более надежной и более эффективной;
- ◆ *наилучшее приближение* — выполняется алгоритмом Ремеза для вычисления минимаксного многочлена с наименьшей ошибкой аппроксимации в худшем случае. Он

находит экстремумы  $f$  и поэтому не является «черным ящиком». Интерполяция Чебышева даст ту же ошибку с немного большей степенью и намного быстрее (см. [23.61]). Наилучшая интерполяция полезна только для оценки элементарной функции (некоторые примеры приведены в работе [23.48]).

Интерполяция при больших  $D$  подвержена проклятию размерности и даже для  $D > 3$  во многих приложениях неосуществима.

*Гауссова квадратура* — это самый точный метод интегрирования для количества оценок для гладкой  $f$ . Традиционно она работает намного медленнее, чем Кленшоу — Кертис (см. [23.51]), но дает алгоритм  $O(n)$  (см. [23.8]). Тем не менее точечное повторное использование не работает, и, поскольку Кленшоу — Кертис почти так же точен (см. [23.61]), обычно используют его.

В задаче многомерного интегрирования авторы [23.16] используют рекурсивную оценку ошибки, предоставляя больше ресурсов для базового случая и других интеграций небольшого объема. Чтобы выбрать более простую стратегию, нужны эксперименты.

В работе [23.28] обсуждаются более продвинутые схемы разбиения. В задаче одномерной адаптивной интеграции идея состоит в том, чтобы дать больше оценок областям с более высокой оценочной ошибкой. Кроме того, основной детерминированный подход может быть как менее, так и более эффективным, чем повторная интеграция, в зависимости от  $f$  (см. [23.42]).

Другие кубатурные методы подразделяют область интегрирования непосредственно на небольшие гиперпрямоугольники или другие примитивы и интегрируют каждый из них, предполагая некоторую интерполяцию или приближение. Например, для прямоугольных областей в 2D можно сделать следующее:

1. Выполнить разбиение на маленькие прямоугольники, используя сетку.
2. Разрезать каждый маленький прямоугольник на треугольники.
3. Для любого треугольника интерполировать плоскость через угловые точки.
4. Интегрировать, предполагая, что  $f$  правильно моделируется плоскостью.

Также при выборке всех угловых точек и повторном интегрировании можно использовать правило трапеций для тензорного произведения. Порядок не имеет значения, т. к. все точки обладают одинаковым весом. При этом используется больше точек, чем при триангуляции, но возникает ошибка того же порядка. Примеры общих областей в 2D приведены в кубатурном разделе [23.50]. Согласно одному из квадратных правил используются 5 точек (угловые и центр), чтобы получить порядок  $O(h^4)$  для квадрата размером  $h$ . Правило произведения Симпсона требует использования 9 точек. Правила произведения не оптимальны, но для повторного интегрирования можно независимо адаптировать каждое измерение.

Можно масштабировать гиперпрямоугольник до гиперкуба, и для этого нужны только правила гиперкуба. Учитывая любое правило типа Лобатто, которое использует все углы, можно создать адаптивный интегратор, так же как и для 1D, путем удвоения или использования соответствующих расширений Кронрода. Например, *правило Лайнесса* имеет степень 5 и в дополнение к углам использует точки  $\pm\sqrt{2/5}$  в каждой координате (см. [23.15, стр. 379]). Поскольку гиперкуб разбивается на  $2^D$  гиперкубов одинакового размера, сходимость зависит от  $\|\Delta x\|$ , каким бы ни был порядок, и, чтобы сократить его

вдвое, потребуется работа  $O(2^D)$ . Таким образом, чтобы достичь лучшего результата, чем дает последовательность Соболя, нужен порядок  $> 2^D$  и  $f$  с достаточно большим количеством непрерывных производных. Подразделение неэкономично, если  $D$  больше порядка сходимости.

Особый интерес представляют формулы, которые интегрируют все многомерные полиномы до некоторой степени в определенной области точно с минимальным количеством оценок, например интегрирование Гаусса в 1D (правило Лайнесса является одним из примеров). Они могут быть эффективны для небольших  $D$  и обычных областей, таких как гиперпрямоугольники и гиперсферы. Для получения дополнительной информации см. работы [23.11 и 23.15] и ссылки в них.

Как правило, при наличии надежного и достаточно быстрого метода не следует использовать более быстрые и чуть менее надежные методы. Это особенно заметно при решении уравнений с одной переменной, ибо методов, стремящихся превзойти деление пополам, великое множество. Они могут использовать  $1/3$  вычислений, но это не имеет большого значения, учитывая, что деление тоже не требовательно. Популярный *метод Брента*, сочетающий деление пополам с параболической интерполяцией, в худшем случае может иметь проблемы с производительностью (см. [23.72]).

У метода Бройдена с ограниченной памятью явная формула (см. [23.64]) также позволяет неявно вычислять  $B$  с той же сложностью, хотя представленный подход оказывается проще за счет непосредственного вычисления  $B_m x$  и эквивалентен формуле. Эвристика надежности для решения уравнений является моей оригинальной идеей.

Для решения уравнений при большом  $D$  *метод Ньютона* — *Крылова* лучше, чем метод Бройдена с ограниченной памятью, если известен шаблон разреженности якобиана (т. е. информация о том, какие компоненты  $f$  не зависят от каких переменных (см. [23.36])). Используйте метод Ньютона с возвратом, но оценивайте якобианы аналитически или по конечным разностям, а для уравнений используйте итерационные решатели, такие как CCNR. Это очень хорошо работает для многих приложений, таких как решение уравнений в частных производных, где аналитическая структура известна, а метод не является полностью автоматическим.

Существует нерабочая идея, которая заключается в применении некоторой формы спуска по координатам (см. главу 24. *Численная оптимизация*), потому что все  $f$ -компоненты могут зависеть от любого подмножества переменных, создавая неквадратную систему. Например, можно решить одно уравнение за раз с одной переменной за раз, но не гарантируется, что уравнение зависит от этой переменной. Имея большую систему, можно решить последовательность меньших систем, используя QR Бройдена. Здесь, возможно, уравнения и переменные выбираются случайным образом и повторяются без случайных шагов, пока не будет достигнут локальный максимум. Такие методы являются эвристическими.

Для нахождения всех корней полинома лучшего всего использовать недавнюю модификацию решения собственных значений матрицы-компаньона (см. [23.3]).

Получение производных от формул Рунге — Кутты и доказательство их аналитических свойств затруднительно. Возможно, самое подробное объяснение принадлежит изобретателю эффективного метода вычисления производных (см. [23.10], а также [23.29 и 23.41]).

До открытия формул Дорманда — Принса лучше всех были *формулы Фельберга* (см. [23.29]). Существует формула Дорманда — Принса 8-го, которая работает весьма хорошо (см. [23.29, 23.51]), но по какой-то причине она не включена в набор MATLAB (см. [23.55]). Может быть, дополнительная сложность не стоит того.

Основным историческим конкурентом методов Рунге — Кутты являются *методы предиктора-корректора* (МПК). Идея их состоит в том, чтобы интерполировать значения  $f$  из последних  $k$  шагов, интегрировать интерполяцию, используя текущее значение  $y_i$  в качестве константы интегрирования, и использовать результат для вычисления  $y_{i+1}$ . Учитывая выбранное значение  $h$ , необходимо несколько компонентов:

- ◆ явный *метод Адамса — Бэшфорда*:  $f_i$  является последним. Метод удобен, но имеет плохие свойства с точки зрения устойчивости (см. [23.1]);
- ◆ неявный *метод Адамса — Моултона*:  $f_{i+1}$  является последним, поэтому нужно предположение о значении  $y_{i+1}$  для оценки. Метод обладает хорошей устойчивостью (см. [23.1]), но требует решения неявного уравнения.

МПК получается в результате использования:

- ◆ метода Адамса — Бэшфорда, чтобы угадать  $y_{i+1}$  для метода Адамса — Моултона;
- ◆ одного шага итерации с фиксированной точкой для решения неявного уравнения, т. е. использования  $f_{i+1}(x_{i+1}, y_{i+1})$ . Также обычно задействуют  $k + 1$  последних значений  $f$ , не удаляя самое старое;
- ◆ суммы коррекций предположения  $y_{i+1}$ , что может быть хорошей оценкой ошибки (называемой *ошибкой Милна*).

Некоторые формулы и примеры с интервалами  $s$ , равными  $h$ , приведены в работе [23.20]. Но они бесполезны, потому что для решения нужны адаптивные переменные шаги, поэтому лучше использовать интерполяцию напрямую. Традиционно использовалась старая *интерполяция Ньютона* (см. [23.20]), но барицентрическая интерполяция является более устойчивой (хотя и неустойчивой для экстраполяции, т. е. интегральной оценки для предиктора).

Эта формулировка также позволяет динамически изменять порядок, используя желаемое количество последних точек. Реализация MATLAB в некоторых случаях изменяет  $k$  до 13 (см. [23.55]), что, по утверждению авторов, лучше метода Дорманда — Принса для высокоточных вычислений с дорогостоящей оценкой  $f$ .

Практическая реализация МПК кажется слишком сложной. В работе [23.29] описываются некоторые детали реализации:

- ◆ в качестве первого шага используйте шаг Эйлера и задайте порядок, задействовав описанный ранее механизм. Но чтобы получить точность, близкую к  $\epsilon_{\text{machine}}$ , этот шаг должен быть столь же мал, а это, по-видимому, не позволяет быстро увеличить размер шага на несколько порядков до того, где он должен быть при правильном порядке;
- ◆ можно использовать барицентрическую форму или форму Ньютона для автоматической интерполяции, но прямое интегрирование не представляется возможным. Например, очевидная идея обращения барицентрической матрицы дифференцирования несостоятельна, поскольку последняя является сингулярной. Численное интегриро-



вание, точное в случае полиномов высокой степени, является очевидным вариантом, но может быть медленным и недостаточно точным. Поэтому для интегрирования формы Ньютона используются специальные методы.

Логика изменения порядка эвристическая. Непонятно, можно ли сделать что-то лучше. Сравним ее с методом Дорманда — Принса:

- ◆ теоретически методы Рунге — Кутты более устойчивы, чем многошаговые методы, особенно методы высокого порядка (см. [23.57]). Для первых повышение порядка улучшает устойчивость;
- ◆ тесты, приведенные в работе [23.29], говорят, что Дорманд — Принс (8-й порядок) всегда выполняется быстрее, чем код, на котором основана реализация MATLAB. Реализация последнего, как указано ранее, влечет много накладных расходов;
- ◆ основываясь на тех же тестах, код МПК выполняет меньше оценок при высокой точности. Возможно, именно поэтому код МПК MATLAB рекомендуется использовать для сверхдорогих  $f$  с высокой точностью. Но Дорманд — Принс может приблизиться к  $\epsilon_{\text{machine}}$  за не слишком большое количество вычислений, а вычисление гладких  $f$  редко требует больших затрат, потому что библиотеки элементарных функций работают быстро, а по-настоящему дорогие  $f$  едва ли гладкие.

*Методы экстраполяции*, за которые выступают авторы [23.51], будучи использованными для многошаговых методов, аналогичны. В исследованиях они показывают себя не столь хорошо, и в большинстве крупных библиотек их игнорируют (см. [23.34, 23.37]).

Использование метода Бройдена для жестких ОДУ — это нестандартный ход. В старых реализациях чаще используется метод Ньютона, который не обновляет конечно-разностные  $J$ . Обновления по методу Бройдена строго лучше. Вы можете сохранять текущую матрицу Бройдена для следующего шага, что позволит существенно сэкономить вычислительные ресурсы, но этот метод требует дальнейшего изучения.

Для жестких уравнений в MATLAB есть хорошие реализации неявных многошаговых методов, которые основаны на другой интерполяции (см. [23.55]). Эксперименты показывают, что код `radau5` от `Radau IIA` (см. [23.30]) медленнее, но безопаснее, чем многоэтапный метод `MATLAB ode15s` (см. [23.54, 23.27]). Теоретически, согласно *теореме Дальквиста о втором барьере*, ни один многошаговый метод порядка  $> 2$  не является А-стабильным (см. [23.57]). Но некоторые практические методы, реализованные в MATLAB (включая неявные), удовлетворяют облегченной версии А- и L-устойчивости, и этого оказывается достаточно для хорошей производительности (см. [23.55]).

Анализ сходимости для явных и неявных методов выполняется сложнее, поскольку приходится учитывать несколько начальных значений, и одной согласованности недостаточно. Хотя, впрочем, об этом заботятся такие условия, как нулевая стабильность (см. [23.57]). Интуитивная причина, по которой методам с запоминанием требуется больше ресурсов, чем одношаговым методам, заключается в том, что используемые значения предыдущего шага вычисляются с некоторой ошибкой, которая распространяется на дальнейшие вычисления.

## 23.22. Советы по дополнительной подготовке

- ◆ На существующих данных сравните интегрирование по правилу трапеций с кусочно-барицентрической интерполяцией, в которой используются фрагменты малой степени путем взятия слева направо нескольких точек за раз (в последнем фрагменте задействуются все оставшиеся точки).
- ◆ Расширьте барицентрическую интерполяцию возможностями работы с динамическим набором точек, как у линейной интерполяции. Вместо того, чтобы выбирать ближайшие точки в качестве ближайших соседей, возможно, стоит убедиться, что стабильность достигается не за счет использования дополнительной охватывающей точки. Протестируйте интерполяцию на вычисление производной с заданными точками и подумайте, может ли она конкурировать с динамической линейной интерполяцией. Еще один вариант — вычисление кусочной интерполянты статически из всех доступных данных (при этом последняя или средняя точка получает больше баллов, если не является точно кратной степени).
- ◆ Расширьте интеграцию Монте-Карло, чтобы она допускала использование обобщенного семплера вместо простого принятия-отклонения. Это позволит повысить эффективность в особых случаях.
- ◆ Внедрите адаптивное интегрирование с использованием правила Лайнесса и сравните его эффективность с эффективностью конкурирующих методов.
- ◆ Отсортируйте корни для нахождения всех корней с помощью интерполяции Чебышева.
- ◆ Расширьте код Дорманда — Принса возможностью обработки систем уравнений. Для методов МПК это непросто. Преобразуйте код в вычислитель шагов, который позволяет выполнять определяемую пользователем функцию, по умолчанию невыполняемую. Функции потребуется передавать оценку локальной ошибки.
- ◆ Научите решатели ОДУ использовать единственное ограничение — количество вычислений функций.
- ◆ Докажите, что у интерполяции формы Ньютона нет вариантов использования. Пользовательский вариант МПК может в этом помочь, но его можно обновить до барицентрической формы, что даст лучшие результаты. Согласно [23.7] барицентрическая форма не является единственно полезной, но этому утверждению не приводится никакого обоснования. Можно оттолкнуться от известного факта, что барицентрическая форма не стабильна для экстраполяции.
- ◆ Неравенство братьев Марковых дает очень слабые границы константы ошибки в случае аппроксимации производной интерполяцией. Изучите литературу и попробуйте получить лучшие оценки или лучшие общие доказательства.
- ◆ Проверьте все представленные алгоритмы, чтобы убедиться, что они дают соответствующие ошибки, когда некоторые входные данные равны NaN или бесконечности.

## 23.23. Список рекомендуемой литературы

- 23.1. Ascher U. M., & Greif C. (2011). *A First Course on Numerical Methods*. SIAM.
- 23.2. Ascher U. M., Mattheij R. M., & Russell R. D. (1995). *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations* (Vol. 13). SIAM.
- 23.3. Aurentz J. L., Mach T., Robol L., Vandebril R., & Watkins D. S. (2018). *Core-Chasing Algorithms for the Eigenvalue Problem*. SIAM.
- 23.4. Beatson R. K. (1986). On the convergence of some cubic spline interpolation schemes. *SIAM Journal on Numerical Analysis*, 23(4), 903–912.
- 23.5. Beatson R. K., & Chacko E. (1990). Which cubic spline should one use?
- 23.6. Beebe N. H. F. (2017). *The Mathematical-Function Computation Handbook*. Springer.
- 23.7. Berrut J. P., Trefethen L. N. (2004). Barycentric Lagrange Interpolation. *SIAM Review*, 46(3), 501–517.
- 23.8. Bogaert I. (2014). Iteration-Free Computation of Gauss-Legendre Quadrature Nodes and Weights. *SIAM Journal on Scientific Computing*, 36(3), A1008–A1026.
- 23.9. Boyd J. P. (2014). Solving Transcendental Equations: The Chebyshev Polynomial Proxy and Other Numerical Rootfinders, Perturbation Series, and Oracles. SIAM.
- 23.10. Butcher J. C. (2016). *Numerical Methods for Ordinary Differential Equations*. Wiley.
- 23.11. Cools, R. (2002). Advances in multidimensional integration. *Journal of Computational and Applied Mathematics*, 149(1), 1–12.
- 23.12. Corless R. M., & Fillion N. (2013). *A Graduate Introduction to Numerical Methods*. Springer.
- 23.13. Cormen T. H., Leiserson C. E., Rivest R. L., & Stein C. (2009). *Introduction to Algorithms*. MIT Press.
- 23.14. Dahlquist G., & Björck Å. (2008). *Numerical Methods in Scientific Computing*. SIAM.
- 23.15. Davis P. J., & Rabinowitz P. (1984). *Methods of Numerical Integration*. Dover.
- 23.16. de Doncker E., Li S., & Kaugars K. (2007). Error distribution for iterated integrals. *WSEAS Transactions on Mathematics*, 6(1), 86.
- 23.17. de Villiers J. (2012). *Mathematics of Approximation*. Atlantis Press.
- 23.18. Dennis Jr J. E., & Schnabel R. B. (1996). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM.
- 23.19. Erdős P., & Vértesi P. (1980). On the almost everywhere divergence of Lagrange interpolatory polynomials for arbitrary system of nodes. *Acta Mathematica Hungarica*, 36(1–2), 71–89.
- 23.20. Fausett L. V. (2003). *Numerical Methods: Algorithms and Applications*. Pearson.
- 23.21. Frigo M., & Johnson S. G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 216–231.
- 23.22. Gander W., Gander M. J., & Kwok F. (2014). *Scientific Computing — An Introduction using Maple and MATLAB*. Springer.
- 23.23. Gao J., & Iserles A. (2016). An adaptive Filon algorithm for highly oscillatory integrals.
- 23.24. Gautschi W. (2011). *Numerical Analysis*. Birkhäuser.
- 23.25. Golub G. H., & Van Loan, C. F. (2012). *Matrix Computations*. JHU Press.
- 23.26. Gonnet P. (2009). *Adaptive Quadrature Re-revisited* (Doctoral dissertation, ETH Zürich).
- 23.27. Gonnet P., Dimopoulos S., Widmer L., & Stelling J. (2012). A specialized ODE integrator for the efficient computation of parameter sensitivities. *BMC systems biology*, 6(1), 46.
- 23.28. Hahn T. (2005). CUBA—a library for multidimensional numerical integration. *Computer Physics Communications*, 168(2), 78–95.

- 23.29. Hairer E., Nørsett S. P., & Wanner G. (1993). Solving Ordinary Differential Equations I: Nonstiff Problems. Springer.
- 23.30. Hairer E., & Wanner G. (1996). Solving ordinary differential equations II. Springer.
- 23.31. Hastie T., Tibshirani R., & Friedman J. (2009). The Elements of Statistical Learning. Springer.
- 23.32. Heath M. T. (2018). Scientific Computing: An Introductory Survey. SIAM.
- 23.33. Higham N. J. (2002). Accuracy and Stability of Numerical Algorithms. SIAM.
- 23.34. Hosea M. E., & Shampine L. F. (1994). Efficiency comparisons of methods for integrating ODEs. Computers & Mathematics with Applications, 28(6), 45–55.
- 23.35. Isaacson E., & Keller H. B. (1994). Analysis of Numerical Methods. Dover.
- 23.36. Kelley C. T. (2003). Solving Nonlinear Equations with Newton's Method. SIAM.
- 23.37. Ketcheson D., & bin Waheed U. (2014). A comparison of high-order explicit Runge — Kutta, extrapolation, and deferred correction methods in serial and parallel. Communications in Applied Mathematics and Computational Science, 9(2), 175–200.
- 23.38. Kopriva D. (2009). Implementing Spectral Methods for Partial Differential Equations: Algorithms for Scientists and Engineers. Springer.
- 23.39. Kranzer H. C. (1963). An error formula for numerical differentiation. Numerische Mathematik, 5(1), 439–442.
- 23.40. Kythe P. K., & Schäferkotter M. R. (2004). Handbook of Computational Methods for Integration. CRC.
- 23.41. Lambert J. D. (1991). Numerical Methods for Ordinary Differential Systems: The Initial Value Problem. Wiley.
- 23.42. Li S. (2005). Online Support for Multivariate Integration. (Doctoral dissertation, Western Michigan University).
- 23.43. Lyons R. (2015). Four Ways to Compute an Inverse FFT Using the Forward FFT Algorithm. <https://www.dsprelated.com/showarticle/800.php>. Accessed August 5, 2017.
- 23.44. Martinez J. M. (2000). Practical quasi-Newton methods for solving nonlinear systems. Journal of Computational and Applied Mathematics, 124(1), 97–121.
- 23.45. Mason J. C., & Handscomb D. C. (2002). Chebyshev Polynomials. CRC.
- 23.46. Moler C. B. (2013). Numerical Computing with MATLAB. [https://www.mathworks.com/moler/index\\_ncm.html](https://www.mathworks.com/moler/index_ncm.html).
- 23.47. Moré J. J., Garbow B. S., & Hillstom K. E. (1981). Testing unconstrained optimization software. ACM Transactions on Mathematical Software (TOMS), 7(1), 17–41.
- 23.48. Muller J. M. (2016). Elementary Functions. Birkhäuser.
- 23.49. Nocedal J., Wright S. (2006). Numerical Optimization, 3rd ed. Springer.
- 23.50. NIST (2017). NIST Digital Library of Mathematical Functions 1.0.16. <http://dlmf.nist.gov/>.
- 23.51. Press W. H., Teukolsky S. A., Vetterling W. T., & Flannery B. P. (2007). Numerical Recipes: The Art of Scientific Computing, 3rd ed. Cambridge University Press.
- 23.52. Powell M. J. D. (1981). Approximation Theory and Methods. Cambridge University Press.
- 23.53. Prenter P. M. (1975). Splines and Variational Methods. Dover.
- 23.54. Sandu A., & Sander R. (2006). Technical note: Simulating chemical systems in Fortran90 and Matlab with the Kinetic PreProcessor KPP-2.1. Atmospheric Chemistry and Physics, 6(1), 187–195.
- 23.55. Shampine L. F., & Reichelt M. W. (1997). The MATLAB ODE suite. SIAM Journal on Scientific Computing, 18(1), 1–22.
- 23.56. Smith J. O. (2007). Mathematics of the Discrete Fourier Transform (DFT): With Audio Applications. W3K.

- 23.57. Süli E., & Mayers D. F. (2003). *An Introduction to Numerical Analysis*. Cambridge University Press.
- 23.58. Trangenstein J. A. (2018a). *Scientific Computing: Vol. I: Linear and Nonlinear Equations*. Springer.
- 23.59. Trangenstein J. A. (2018b). *Scientific Computing: Vol. II: Eigenvalues and Optimization*. Springer.
- 23.60. Traub J. F., & Werschulz A. G. (1998). *Complexity and Information*. Cambridge University Press.
- 23.61. Trefethen L. N. (2019). *Approximation Theory and Approximation Practice*. SIAM.
- 23.62. Ueberhuber C. W. (1997a). *Numerical Computation 1: Methods, Software, and Analysis*. Springer.
- 23.63. Ueberhuber C. W. (1997b). *Numerical Computation 2: Methods, Software, and Analysis*. Springer.
- 23.64. van de Rotten B. (2003). *A Limited Memory Broyden Method to Solve High-dimensional Systems of Nonlinear Equations*. (Doctoral dissertation, University of Leiden).
- 23.65. Wikipedia (2017a). Discrete Fourier transform. [http://en.wikipedia.org/wiki/Discrete\\_Fourier\\_transform](http://en.wikipedia.org/wiki/Discrete_Fourier_transform). Accessed January 4, 2017.
- 23.66. Wikipedia (2017b). Chirp Z-transform. [https://en.wikipedia.org/wiki/Chirp\\_Z-transform](https://en.wikipedia.org/wiki/Chirp_Z-transform). Accessed August 6, 2017.
- 23.67. Wikipedia (2017c). Fubini's theorem. [https://en.wikipedia.org/wiki/Fubini%27s\\_theorem](https://en.wikipedia.org/wiki/Fubini%27s_theorem). Accessed August 12, 2017.
- 23.68. Wikipedia (2017d). Runge-Kutta methods. [https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta\\_methods](https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods). Accessed August 20, 2017.
- 23.69. Wikipedia (2017e). Markov brothers' inequality. [https://en.wikipedia.org/wiki/Markov\\_brothers%27\\_inequality](https://en.wikipedia.org/wiki/Markov_brothers%27_inequality). Accessed October 15, 2017.
- 23.70. Wikipedia (2017f). Discrete cosine transform. [https://en.wikipedia.org/wiki/Discrete\\_cosine\\_transform](https://en.wikipedia.org/wiki/Discrete_cosine_transform). Accessed September 2, 2017.
- 23.71. Wikipedia (2017g). Hessian matrix. [https://en.wikipedia.org/wiki/Hessian\\_matrix](https://en.wikipedia.org/wiki/Hessian_matrix). Accessed November 26, 2017.
- 23.72. Wilkins G., & Gu M. (2013). A modified Brent's method for finding zeros of functions. *Numerische Mathematik*, 123(1), 177–188.

## 24. Численная оптимизация

Если алгоритм сходится неоправданно быстро, наверное, используется метод Ньютона.  
*Джон Деннис*

### 24.1. Введение

Мы сделаем здесь упор на алгоритмы неограниченной оптимизации — как локальной, так и глобальной, поскольку их реализации весьма подробно описаны. В идеале читатель должен иметь за плечами курс по численным методам.

### 24.2. Некоторые общие идеи

Задачи минимизации в некоторой степени плохо обусловлены (см. [24.36]): у двух  $f$  с двумя непрерывными производными разложение Тейлора в точке  $x^*$ , где  $f'(x^*) = 0$  для  $x \in [x^* \pm \varepsilon]$ , приводит к  $|f(x) - f(x^*)| \leq O(\varepsilon^2)$ . То есть рядом с точкой минимума  $f$  не различает два значения  $x$  с относительной ошибкой  $o(\sqrt{\varepsilon_{\text{machine}}})$ , потому что возмущения  $x$  такой величины приводят к возмущениям  $f$  настолько малым, что оба результата приближаются к одному и тому же значению  $f$ . Так что большей точности ждать не приходится. Тем не менее к функциям  $f$ , которые не удовлетворяют условиям ряда Тейлора, например  $f(x) = |x|$ , это не относится.

В задаче поиска корней такой проблемы не возникает, и корни можно найти с точностью до  $O(\varepsilon_{\text{machine}})$ . Следует убедиться, что возведение объекта в квадрат делает градиент в корне равным 0 (по правилу цепного дифференцирования). Из-за этого сведение поиска корня к задаче минимизации кажется плохой идеей, но в целом это мало что меняет, потому что в первую очередь требуется найти именно корень, особенно в многомерном случае.

Задача минимизации несколько отличается от задачи поиска корней. Для корней важна точность  $x$ , а для оптимизации нужен любой  $x$ , такой, что  $f(x)$  не слишком далеко от  $f(x^*)$ . Таким образом, важнее всего относительная точность по оси  $y$ , хотя обычно довольствуются абсолютной точностью по оси  $y$ . Это лучше видно в случае комбинаторной оптимизации, т. к. точная природа найденного объекта не важна, и вы не можете оценить его, в отличие от значения  $y$ .

Работает и обратный метод — сведение задачи минимизации к нахождению корня. Для этого ищется корень градиента функции (предполагая дифференцируемость  $f$  и возможность точного вычисления  $\nabla f$ ). Градиенты дают дополнительную информацию о задаче, что позволяет улучшить обусловленность. Но точность улучшится только по  $x$ , а по  $y$  — нет. Также вам потребуется дополнительная информация, чтобы отличить минимумы от максимумов.

Имеет ли точность по  $x$  значение для условий завершения. Обсудим некоторые тезисы:

- ◆ в некоторых случаях это единственный вариант. Например, в методе золотого сечения и спуска по координатам (далее в этой главе) она используется напрямую;
- ◆ как мы уже говорили, задавать высокую точность по  $x$  на локальном минимуме бессмысленно. Но это может быть полезно в случае использования итерационных алгоритмов, таких как спуск по координатам, потому что меньшие значения могут привести к преждевременному завершению. Золотое сечение отличается тем, что одинарная точность по умолчанию становится актуальной только вблизи минимума, что хорошо работает;
- ◆ алгоритм должен проверять только условие завершения. Качество решения (близость корня/градиента к 0 для решения/оптимизации соответствующего уравнения) должно проверяться вызывающей стороной или оборачивающей функцией с независимой логикой. Проблем с масштабированием при проверке градиента чаще всего можно избежать за счет правильного масштабирования (см. [24.15]), и связи этой логики с алгоритмом не требуется. В частности, пользователь должен проверить, что относительный градиент  $\frac{\|\nabla f\|_2}{\max(1, |y|)}$  достаточно мал. Это хорошо работает, даже если  $y$  постоянный аддитивный коэффициент, поскольку проверки точности основаны на значениях  $f$ ;
- ◆ если точность по  $y$  работает, то точность по  $x$  использоваться не будет, и эта стратегия кажется надежной. Еще одна точка зрения заключается в том, что точность по  $x$  используется для обнаружения стагнации, а точность по  $y$  (оценочный градиент, в частности, для некоторых алгоритмов) служит для принятия решения об успешной остановке. Но точность по  $y$  не имеет проблем с условиями вблизи минимума и выглядит столь же надежной. Ее использование позволяет избежать ситуации, когда  $\Delta x$  мало, а  $\nabla f$  велико, что может привести к проблемам с альтернативными тестами. В случаях, когда необходимо задействовать точность по  $x$ , проверка градиента на равенство 0 являет собой дополнительное условие завершения, но в этом случае любая хорошая логика быстро делает  $x$  ниже точности завершения.

Во всех полезных базовых методах используется некоторая локальная информация, основанная на выборке  $f$  и ее производных. Поэтому при выполнении глобального поиска нужна явная логика глобализации.

### 24.3. Минимизация унимодальной функции с одной переменной

Пусть функция  $f$  унимодальна на  $[a, b]$ , т. е. существует значение  $m$  такое, что при  $x \leq m$  функция  $f$  монотонно убывает, а при  $x \geq m$  монотонно растет (см. [24.47]). Унимодальная  $f$  не обязательно должна быть непрерывной, а одномерная унимодальная функция называется *квазивыпуклой* (определение дано далее в этой главе).

Поиск золотого сечения выполняется по значениям  $(a, b, c)$  таким, что  $f(a) \leq f(b) \leq f(c)$ :

1. До сходимости (обычно от  $a$  до  $c$ ):
2. Выберите  $x \in$  большему из  $[a, b]$ ,  $[b, c]$ .

3. Если  $a \leq x \leq b \leq c$ , используйте  $(a, x, b)$ .

4. Если  $a \leq b \leq x \leq c$ , используйте  $(b, x, c)$ .

Оптимальное значение  $x$  находится не посередине, а 1 — золотое сечение в большем интервале от  $b$ . Алгоритм завершается даже в арифметике с плавающей точкой и стохастической разрывной  $f$ .

```
template<typename FUNCTION> pair<double, double> minimizeGS(
    FUNCTION const& f, double xLeft, double xRight,
    double relAbsXPrecision = numeric_limits<double>::epsilon())
{ // не хочется, чтобы точность была слишком низкой
    assert(isfinite(xLeft) && isfinite(xRight) && xLeft <= xRight &&
        relAbsXPrecision >= numeric_limits<double>::epsilon());
    double GR = 0.618, xMiddle = xLeft * GR + xRight * (1 - GR),
        yMiddle = f(xMiddle);
    while(isELess(xLeft, xRight, relAbsXPrecision))
    {
        bool chooseR = xRight - xMiddle > xMiddle - xLeft;
        double prevDiff = xRight - xLeft, xNew = GR * xMiddle + (1 - GR) *
            (chooseR ? xRight : xLeft), yNew = f(xNew);
        if(yNew < yMiddle)
        {
            (chooseR ? xLeft : xRight) = xMiddle;
            xMiddle = xNew;
            yMiddle = yNew;
        }
        else (chooseR ? xRight : xLeft) = xNew;
    }
    return make_pair(xMiddle, yMiddle);
}
```

Без ограничения общности предположим, что  $x > b$ . Пусть  $w = \frac{b-a}{c-a}$  есть доля пути  $b$

в  $[a, c]$  и  $z = \frac{x-b}{c-a}$  есть доля пути  $x$  в  $[a, c]$  после  $b$ . Следующий интервал — это  $[a, x]$

относительной длины  $w + z$  или  $[b, c]$  относительной длины  $1 - w$ . Наихудший случай является также наименьшим, когда они равны  $\rightarrow z = 1 - 2w$ . Предполагая, что предыдущий выбор оптимален,  $x$  — та же часть пути в  $[b, c]$ , что и  $b$  в  $[a, c]$ , поэтому

$$w = \frac{x-b}{c-b} = \frac{z}{1-w} \rightarrow w = 1 \text{ дает золотое сечение. Вначале } b = c \text{ не оптимально, но интервалы быстро становятся оптимальными. Из-за геометрически сокращающихся интервалов сходимость является линейной, т. е. требуется оценка } O(\lg(1/\epsilon_{\text{absolute}})).$$

Если  $f$  не является унимодальной, метод не обязательно должен работать, и в лучшем случае будет найден минимум в некоторой одномодальной области, которая может быть даже хуже, чем одна из конечных точек. Для работы с неунимодальными  $f$  нужны глобальные стратегии оптимизации, которые теоретически ничем не лучше случайного поиска.

Если  $f$  не является унимодальной, метод не обязательно должен работать, и в лучшем случае будет найден минимум в некоторой одномодальной области, которая может быть даже хуже, чем одна из конечных точек. Для работы с неунимодальными  $f$  нужны глобальные стратегии оптимизации, которые теоретически ничем не лучше случайного поиска.

У стохастических  $f$ , ожидаемое значение которых является унимодальным, можно получить достаточно точные средние значения и минимизировать результирующую



детерминированную функцию эвристическими методами. Оценка ошибки будет точной только до ожидаемого уровня шума.

Если известна начальная точка, а охватывающий интервал неизвестен, как в задаче нахождения корня, используйте брекетинг с достаточно большим начальным расстоянием, чтобы превысить возможный шум. Здесь это хорошо масштабированное значение  $\max(1, |x_i|) / 10$ . Итак, если предположить, что  $f$  унимодальна, найдем трехточечный шаблон  $\{x_0, x_1, x_2\}$  такой, что  $f(x_0) \geq f(x_1) \leq f(x_2)$ :

1. Используйте первые две оценки, чтобы определить направление минимума.
2. Удваивайте значение, пока не перешагнете через минимум.
3. Примените золотое сечение к полученному интервалу.

```
template<typename FUNCTION> pair<double, double> unimodalMinBracket (FUNCTION
    const& f, double x0, double fx, bool twoSided, double d, int maxEvals)
{
    assert(abs(d) > 0 && isfinite(x0) && isfinite(d)); // && maxEvals > 2?
    pair<double, double> best(x0, x0 + d);
    double fMin = f(x0 + d);
    maxEvals -= 2;
    if(fx < fMin && twoSided) // проверка направления убывания, если
                            // функция двусторонняя
    {
        d *= -1; // максимальный паттерн должен находиться в другом направлении
        fMin = fx;
        swap(best.first, best.second);
    }
    if(!(fx < fMin)) // если 1-сторонняя функция не может
                  // увеличить текущий интервал
        while(d * 2 != d && maxEvals-- > 0) // обработка случая d = 0, inf и NaN
        {
            d *= 2;
            double xNext = x0 + d, fNext = f(xNext);
            if(fNext < fMin)
            { // сдвиг
                best.first = best.second;
                best.second = xNext;
                fMin = fNext;
            }
            else
            { // найден 3-точечный паттерн, формируем интервал
                best.second = xNext;
                break;
            }
        } // получаем отсортированный интервал
    if(best.first > best.second) swap(best.first, best.second);
    return best;
}

template<typename FUNCTION> pair<double, double> minimizeGSBracket(
    FUNCTION const& f, double x, double fx = numeric_limits<double>::
    quiet_NaN(), bool twoSided = true, double step = 0.1,
    double relAbsXPrecision = defaultPrecEps, int bracketMaxEvals = 100)
```

```

{
    if(!isfinite(x)) return make_pair(x, fx);
    if(!isfinite(fx))
    {
        fx = f(x);
        --bracketMaxEvals;
    }
    pair<double, double> bracket = unimodalMinBracket(f, x, fx, twoSided,
        step * max(1.0, abs(x)), bracketMaxEvals), result = minimizeGS(f,
        bracket.first, bracket.second, relAbsXPrecision); // проверка на неубывание
    return result.second < fx ? result : make_pair(x, fx);
}

```

Для двустороннего поиска и выпуклого  $f$  сходимость гарантируется тем, что существуют конечные начальные  $x$  и  $d$ . Если  $f$  невыпуклая, алгоритм всегда завершается, и результатом обычно является какой-то локальный минимум.

## 24.4. Минимизация многомерных функций: введение и координатный спуск

Для  $D > 1$  оптимальное направление движения неизвестно, т. к. сужающегося интервала решения не существует. Все методы начинаются с начального предположения и угадывают направление на основе значений ближайших точек или информации о градиенте. Ни один алгоритм не сходится к локальному оптимуму за предсказуемое количество итераций, даже для достаточно дифференцируемой  $f$  (если нет ограничений на константу Липшица для  $f$  и не используются очень специфические алгоритмы — см. [24.5]). Поскольку оптимизационный ландшафт может быть разделен на области локального минимума, локальный поиск обычно находит минимум внутри такой области.

Простой алгоритм — координатный спуск:

1. Начать с измерения  $i = 0$  и заданной пользователем начальной точки.
2. Пока алгоритм не сойдется или не будет превышено указанное количество итераций, выполнить одномерную оптимизацию на  $x[i]$ :
3. Найти граничный интервал.
4. Выполнить поиск по золотому сечению.
5. Установить  $i = (i + 1) \% D$ .

Вместо циклического перебора координат лучше рандомизировать порядок переменных перед каждым циклом. Для определения границ начальные шаги выполняются для каждого измерения. Начните с независимого, хорошо масштабированного значения 0,1, затем используйте значение  $\min(0,1, \text{относительное изменение любой координаты в последней итерации относительно})$ . Цикл останавливается, когда результат окажется меньше относительной/абсолютной точности, или алгоритм перестанет вносить улучшения. Алгоритм наиболее эффективен для  $f$ , вычисляемых инкрементно, т. е. обновлять  $f$  инкрементно можно только из  $x[i]$  и некоторой предварительно вычисленной информации. И для этого предусмотрен специальный интерфейс:

```

template<typename INCREMENTAL_FUNCTION> double unimodalCoordinateDescent(
    INCREMENTAL_FUNCTION &f, int maxEvals = 1000000,
    double xPrecision = highPrecEps)
{
    int D = f.getSize();
    Vector<int> order(D);
    for(int i = 0; i < D; ++i) order[i] = i;
    double y = f(f.getXi()), relStep = 0.1;
    while(f.getEvalCount() < maxEvals) // значение может быть превышено, но это нормально
    { // используются полные циклы со случайным порядком
        GlobalRNG().randomPermutation(order.getArray(), D);
        double yPrev = y, maxRelXStep = 0;
        for(int i = 0; i < D && f.getEvalCount() < maxEvals; ++i)
        {
            int j = order[i];
            f.setCurrentDimension(j);
            pair<double, double> resultJ = minimizeGSBracket(f, f.getXi(),
                y, true, relStep, xPrecision);
            maxRelXStep = max(maxRelXStep, abs(resultJ.first - f.getXi())/
                max(1.0, abs(f.getXi())));
            f.bind(resultJ.first);
            y = resultJ.second;
        } // выполняется, если x и y не улучшаются
        if(maxRelXStep < xPrecision || !isELess(y, yPrev)) break;
        relStep = min(0.1, maxRelXStep); // после достижения
                                         // сходимости первые шаги делаются малыми
    }
    return y;
}

```

Резонный вопрос заключается в том, является ли эта стратегия лучшей. Она хороша в случае, когда шаг  $1D$  выполняется аналитически, что важно для регрессии LSVM (см. главу 27. *Машинное обучение: регрессия*). Упомянутая ранее реализация численно близка к этой. Эвристика для управления размером начального шага разумна и сходится. В поиске по компасу (обсуждаемому далее в этой главе) задействуется аналогичная стратегия. Вместо этого использование одного и того же относительного размера шага 0,1 неэффективно в последующих проходах, а запоминание отдельного шага для каждой координаты может преждевременно отключить эту координату с шагом 0. В общем случае оптимизация некоторых координат даже на небольшую величину может позволить оптимизировать другие координаты от 0 до большого изменения. Кроме того, любая реализация, которая не выполняет полный одномерный поиск, а выполняет лишь один шаг экспоненциального/двоичного поиска, сталкивается с указанной ранее проблемой с запомненными размерами шагов.

Можно использовать обертку для работы с общими  $f$ :

```

template<typename FUNCTION> struct IncrementalWrapper
{
    FUNCTION f;
    mutable Vector<double> xBound;
    int i;
    mutable int evalCount;
}

```

```

public:
    IncrementalWrapper(FUNCTION const& theF, Vector<double> const& x0):
        f(theF), xBound(x0), i(0), evalCount(0) {}
    void setCurrentDimension(double theI)
    {
        assert(theI >= 0 && theI < xBound.getSize());
        i = theI;
    }
    int getEvalCount()const{return evalCount;}
    int getSize()const{return xBound.getSize();}
    Vector<double> const& getX()const{return xBound;}
    double getXi()const{return xBound[i];}
    double operator()(double xi)const
    {
        double oldXi = xBound[i];
        xBound[i] = xi;
        double result = f(xBound);
        ++evalCount;
        xBound[i] = oldXi;
        return result;
    }
    void bind(double xi)const{xBound[i] = xi;}
};

template<typename FUNCTION> pair<Vector<double>, double>
unimodalCoordinateDescentGeneral(FUNCTION const& f, Vector<double> const&
x0, int maxEvals = 1000000, double xPrecision = highPrecEps)
{
    IncrementalWrapper<FUNCTION> iw(f, x0);
    double y = unimodalCoordinateDescent(iw, maxEvals, xPrecision);
    return make_pair(iw.xBound, y);
}

```

На первый взгляд, нежелательной особенностью этой обертки является тот факт, что она должна запоминать текущие значения, не оставляя эту работу самому алгоритму. Но это нужно для сведения одномерного поиска к «черному ящику», поэтому объектно-ориентированный дизайн изменить не получится.

Это применимо ко всем случаям с затратами времени на итерацию  $O(D \lg(1 / \varepsilon_{\text{золотое сечение}}))$ . Но сходимость даже к локальному минимуму не гарантируется или может быть очень медленной, если не применяются особые случаи, потому что значение  $f$  может улучшаться в смешанных, но не координатных направлениях. На практике производительность часто бывает хорошей, особенно когда одномерный поиск имеет аналитическое решение, — например, для лассо-регрессии (см. главу 27. *Машинное обучение: регрессия*).

Для недифференцируемых  $f$  сходимость не гарантируется из-за возможности застревания. Например, рассмотрим двумерную функцию:

$$f(x) = \frac{1}{2} \max(\|x - c\|_2^2, \|x + c\|_2^2), \text{ где } c = (-1, 1).$$

Тогда для любой точки  $(a, a)$  при  $a \neq 0$ , координаты не движутся в нисходящем направлении (см. [24.14]). На самом деле, эта функция теоретически превосходит любой

инкрементальный алгоритм, выполняющий одномерные шаги. Таким образом, инкрементная оптимизация хорошо работает только для специальных  $f$ .

Координатный спуск, вероятно, больше подходит для случая любого времени, чем другие алгоритмы, в том смысле, что его можно остановить после любого количества полных циклов и по-прежнему давать хорошие значения для любых переменных. У разделимых  $f$  алгоритм сходится за один цикл.

## 24.5. Линейный поиск

Если  $f$  дифференцируема, то в локальном минимуме  $\nabla f = 0$  и любое направление  $d$  такое, что  $d\nabla f < 0$ , является *направлением спуска*, движение в котором на некоторый шаг уменьшает значение  $f$ . Алгоритмы на основе градиента многократно выполняют одномерный поиск в выбранном направлении спуска до сходимости, т. е. когда  $\nabla f < \varepsilon$  для некоторого  $\varepsilon$ . Но способ может не сработать, если оценивать  $\nabla f$  конечными разностями, поэтому более надежно выполнить остановку тогда, когда значение  $f$  перестает улучшаться. Такой *линейный поиск* теоретически находит точный минимизатор на линии в любом направлении. Часто вместо этого используют приближительную минимизацию, которая дает достаточное снижение.

Условия Вульфа для поиска сходящейся линии в направлении спуска  $d$  требуют, чтобы для шага  $s$  и некоторых констант  $0 < c_1 < c_2 < 1$  шаг не был слишком велик (рис. 24.1):

- ◆ длинное:  $f(x + sd) \leq f(x) + c_1 s d\nabla f(x)$  — условие Армиджо/достаточного спуска;
- ◆ короткое:  $\nabla f(x + sd) \geq c_2 d\nabla f(x)$  — условие кривизны.

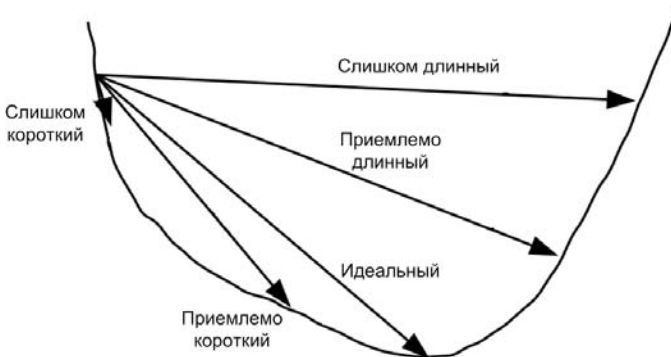


Рис. 24.1. Различные варианты размера шага

Такие  $c_1$  и  $c_2$  всегда существуют при заданном направлении спуска (см. [24.32]). *Теорема Зутендейка*: линейный поиск сходится к локальному минимуму, если:

- ◆  $f$  непрерывно дифференцируема, ограничена снизу и  $\nabla f$  липшицева;
- ◆ выбираются направления спуска, угол которых от уклона не превышает 90 градусов;
- ◆ шаги поиска удовлетворяют условиям Вульфа.

Рассмотрим *метод возврата*, который работает в случае нелинейных уравнений: уменьшать вдвое начальный шаг  $s$  до тех пор, пока не будет достигнут достаточный

спуск с  $c_1 = 10^{-4}$ . Предполагается, что алгоритм начинается с  $s_0$  достаточно большого, чтобы удовлетворить второму условию, и делится пополам, пока не будет выполнено первое. Эксперименты показывают, что у алгоритмов, которые производят хорошо масштабированные направления, такие как ньютоновские (обсуждаемые далее в этой главе), начальный шаг должен асимптотически равняться 1, чтобы иметь суперлинейную локальную сходимость (см. [24.32]). Но у возврата от  $s = 1$  есть свои проблемы:

- ◆ он не удовлетворяет условию кривизны в целом (требуется, среди прочего, сильная выпуклость  $f$  — см. [24.10]), поэтому теорема Зутендейка оказывается неприменима;
- ◆ он может оказаться недостаточно большим для обычных алгоритмов, отличных от метода Ньютона (см. [24.32]).

У метода возврата есть собственная теория сходимости, предотвращающая получение слишком малых значений  $s$  (см. [24.15]), но на практике это не имеет значения. Однако естественным первым шагом к улучшению алгоритма является удвоение  $s$ , если первое проверенное значение удовлетворяет условию достаточного спуска. Алгоритм останавливается, когда достаточный спуск не находится, или значения  $f$  перестают улучшаться. Теоретически это почти эквивалентно запуску обычного возврата с более крупного шага. Отсутствие простого убывания означает, что близок локальный минимум, что, в свою очередь, означает, что близлежащая ступенька удовлетворяет условию кривизны. В частности, выбранный шаг не более чем в два раза больше, чем переход к локальному минимуму.

Наличие информации о градиенте добавляет дополнительное условие остановки, что производная по направлению  $< 0$  (см. [24.32]). Дальше для поиска хорошего качества используются *сильные условия Вульфа* с повышением кривизны до  $|\nabla f(x + sd)| \geq c^2 d |\nabla f(x)|$  и значение  $c_2 = 0,1$  (авторы [24.32] рекомендуют для квазиньютоновских алгоритмов использовать значение 0,9, но в случае конечно-разностных градиентов получается более строгое значение).

В настоящее время лучший алгоритм линейного поиска описан в работе [24.29]. Моя реализация основана на ее логике, но использует алгоритмическую структуру [24.32], основанную на более старом алгоритме.

Пусть функция поиска строки  $\phi(s) = f(x + sd)$ ;  $\phi'(s) = f'(s)d$  — производная по направлению в направлении  $d$ . Таким образом, можно оценить их напрямую, без повторной оценки градиентов. Создадим новую функцию  $\psi(s) = \phi(s) - \phi(0) - c_1 \phi'(0)s$ ;  $\psi'(s) = \phi'(s) - c_1 \phi'(0)$ . Учитывая интервал, в котором присутствует удовлетворительная точка, используем следующие правила для его уменьшения:

1. Пусть  $s = (s_{\text{low}} + s_{\text{high}}) / 2$ .
2. (Случай 1) Если  $\psi(s) > \psi(s_{\text{low}})$ , установить  $s_{\text{high}} = s$ .
3. (Случай 2) В противном случае, если  $\psi'(s)(s_{\text{low}} - s) > 0$ , установить  $s_{\text{low}} = s$ .
4. (Случай 3) Иначе установить  $s_{\text{low}} = s$  и  $s_{\text{high}} = s_{\text{low}}$ .

В случае  $s_{\text{low}}$  и  $s_{\text{high}}$  нижнее значение является лучшим, и они не обязательно должны быть упорядочены. Это доказуемо дает интервал с хорошей точкой в точной арифметике (см. [24.29]). Также используйте небольшую модификацию этого алгоритма для инициализации:

1. Начните с  $s = 1$ ,  $s_{\text{low}} = 0$  и  $s_{\text{high}} = \infty$ .
2. В случае 2 выполнить  $s = 2s$ .
3. Продолжать по обычному алгоритму.

Детали реализации:

- ◆ если случай 1 не срабатывает, проверьте, удовлетворяет ли  $s$  строгим условиям Вульфа, чтобы избежать ненужного дальнейшего сжатия. Интервал может оказаться слишком маленьким;
- ◆ в отличие от решения уравнения с возвратом, для завершения используется точное значение  $y$ . Простым критерием является то, когда  $|\text{производная по направлению } x \times (s_{\text{high}} - s_{\text{low}})|$  слишком мала, чтобы заметно повлиять на  $f$ -значение;
- ◆ работайте непосредственно с  $f$  без определения  $\psi$  или  $\phi$ .

```
template<typename FUNCTION, typename DIRECTIONAL_DERIVATIVE> bool
strongWolfeLineSearchMoreThuente(FUNCTION const& f,
    Vector<double> const& gradient, DIRECTIONAL_DERIVATIVE const& dd,
    Vector<double>& x, double& fx, int& maxEvals, Vector<double> const& dx,
    double yEps)
{
    double dd0 = dotProduct(dx, gradient), sLo = 0, fLo = fx, s = 1,
        sHi = numeric_limits<double>::infinity(), temp = 0.0001 * dd0;
    if(!isfinite(dd0) || dd0 >= 0) return true;
    while(maxEvals > 0 && isfinite(s) &&
        (!isfinite(sHi) || isELess(fx + dd0 * abs(sHi - sLo), fx, yEps)))
    {
        double fNew = f(x + dx * s);
        if(!isfinite(fNew)) break;
        --maxEvals;
        if(fNew - s * temp > fLo - sLo * temp) sHi = s; // случай 1
        else
        {
            double ddS = dd(x + dx * s, dx);
            maxEvals -= dd.fEvals();
            if(abs(ddS) <= -0.1 * dd0) // проверка раннего завершения
            {
                sLo = s;
                fLo = fNew;
                break;
            }
            if((ddS - temp) * (sLo - s) <= 0) sHi = sLo; // случай 3
            // случаи 2 и 3
            sLo = s;
            fLo = fNew;
        }
        if(isfinite(sHi)) s = (sLo + sHi)/2; // масштабирование
        else s *= 2; // случай 2 с начальным удвоением
    }
    if(sLo > 0)
    { // любое ненулевое значение гарантирует достаточный спуск
        x += dx * sLo;
```

```

    double fxFirst = fx;
    fx = fLo;
    return !isELess(fx, fxFirst, yEps); // должен получиться хороший прогресс
}
return true;
}

```

Этот алгоритм надежен. Возможные результаты:

- ◆ найдена удовлетворительная точка;
- ◆ инициализация продолжает удваиваться, пока не достигнет  $\infty$  или не превысит бюджет оценки —  $f$  не имеет конечных минимумов;
- ◆ интервал становится слишком маленьким численно — он все еще содержит хорошую точку, но не выигрывает от дальнейшего сокращения;
- ◆ найденное  $s \approx 0$  — близко к локальному минимуму, или  $d$  почти ортогонально направлению наибольшего спуска.

Наиболее близким к точному поиску строк является алгоритм золотого сечения. К сожалению, он:

- ◆ работает с квазивыпуклым  $f$ , а линейная функция не обязана быть таковой. Например, он может увеличиваться в выборке внутри точек или иметь несколько выпуклых подобластей, и алгоритм будет сходиться к любой из них;
- ◆ не гарантирует выполнение ни одного сильного условия Вульфа: оба должны выполняться, когда  $f$  квазивыпукла, потому что найден точный минимум, но они могут не выполняться, если  $f$  не квазивыпукла или спуск недостаточно велик.

Эксперименты показывают, что алгоритм лучше работает с градиентами конечных разностей. У больших  $D$  оценка градиента требует гораздо больше вычислений, чем линейный поиск. С аналитическими градиентами это, вероятно, не нужно. Обычно в этом случае сразу находится сильная точка Вульфа, но так бывает не всегда. Если алгоритм:

- ◆ находит локальный минимум, кривизна будет удовлетворена;
- ◆ находит удаленный локальный минимум, достаточный спуск может не удовлетворяться;
- ◆ не сработает из-за невыпуклости, по крайней мере вы получите приблизительный шаг.

Поэтому запустите строгий поиск Вульфа по результату, чтобы избежать проблем:

```

template<typename FUNCTION1D> struct EvalCountWrapper
{ // для отслеживания использованных вычислений, которых нет в золотом сечении
    FUNCTION1D f;
    mutable int evalCount;
    EvalCountWrapper(FUNCTION1D const& theF): f(theF), evalCount(0){}
    double operator()(double x) const
    {
        ++evalCount;
        return f(x);
    }
};

```



```

template<typename FUNCTION, typename DIRECTIONAL_DERIVATIVE> bool
goldenSectionLineSearch(FUNCTION const& f, Vector<double> const& gradient,
DIRECTIONAL_DERIVATIVE const& dd, Vector<double>& x, double& fx,
int& maxEvals, Vector<double> const& dx, double yEps, bool useExact = true)
{
    if(!(norm(dx) > 0)) return false; // таким образом также обрабатываются значения NaN
    double step = 1; // хорошо масштабируется для большинства алгоритмов
    if(useExact)
    {
        EvalCountWrapper<ScaledDirectionFunction<FUNCTION> > f2(
            ScaledDirectionFunction<FUNCTION>(f, x, dx));
        pair<double, double> result = minimizeGSBracket(f2, f2.f.getS0(), fx, false);
        maxEvals -= f2.evalCount; // значение может быть превышено, но это нормально
        if(isELess(result.second, fx, yEps))
            step = result.first - f2.f.getS0();
    } // обеспечиваем строгие условия Вульфа
    return strongWolfeLineSearchMoreThuente(f, gradient, dd, x, fx, maxEvals,
        dx * step, yEps);
}

```

## 24.6. Алгоритмы линейного поиска

Самый простой алгоритм линейного поиска — *градиентный спуск*. Для заданной стартовой точки  $x$  этот алгоритм выполняет линейный поиск в направлении  $-\nabla f$ . По теореме Зутендейка он сходится к локальному минимуму при линейном поиске, удовлетворяющем условиям Вульфа. Сходимость является линейной с константой  $sa$ , которая зависит от числа обусловленности  $\nabla^2 f$ , поэтому она очень медленная, если это число велико (см. [24.32]). В моих экспериментах он часто не сходится или имеет низкую точность с заданным бюджетом оценки (по умолчанию  $10^6$  для большинства описанных алгоритмов).

Теорема (см. [24.32]): пусть  $B$  — положительно определенная матрица и  $\exists M > 0$ , такая что  $\kappa(B) < M$  ( $\kappa$  — число обусловленности в любой норме). Тогда  $-B^{-1}\nabla f$  — направление спуска.

В частности, *метод Ньютона* использует направление  $-(\nabla^2 f)^{-1} \nabla f$  с шагом  $= 1$  (см. [24.32]). Также имеет место масштабная инвариантность. Таким образом, при сильном поиске линии Вульфа он сходится к локальному минимуму, если гессиан всегда положительно определен. Сходимость квадратична в окрестности решения. Но:

- ◆ гессиан не обязательно должен быть положительно определенным, и в этом случае он может не привести к направлению спуска и даже не быть обратимым. При любом исправлении, таком как добавление кратного тождества или условие положительности собственных значений, не удастся найти  $s$  без дополнительных неявных или явных предположений, поэтому теряется масштабная инвариантность;
- ◆ гессиан может стать плохо обусловленным и давать направления, где угол с градиентом сколь угодно близок к  $90$  градусам или даже численно немного больше;
- ◆ каждая итерация занимает время  $O(D)$ .

Алгоритм *BFGS* улучшает метод Ньютона, обновляя гессиан из значений функции и производных, например, метода Бройдена для решения уравнений. Метод Ньютона для оптимизации — это метод решения уравнения для градиента  $= 0$ , но, в отличие от якобиана, гессиан симметричен и положительно определен, что будет учитываться в хорошем правиле обновления. Таким образом, правило обновления гарантирует, что значение  $H_{k+1}$  положительно определено, если  $H_k$  таково:

$$H_{k+1} = V_k H_k V_k^T + p_k s_k^2,$$

где:

- ◆  $H = B^{-1}$ ;
- ◆  $p_k = \frac{1}{y_k^T s_k}$ ;
- ◆  $V_k = I - p_k \times (y_k, \otimes s_k)$ ;
- ◆  $s_k = x_{k+1} - x_k$ ;
- ◆  $y_k = \nabla f_{k+1} - \nabla f_k$ .

Доказательства приведены в работах [24.32 и 24.15]. Конечно-разностная аппроксимация для  $H_0$  вообще не обязательно должна быть положительно определенной, и любая поправка приведет к потере масштабной инвариантности. Кроме того, если вы начинаете с любой положительно определенной матрицы, этот метод, в отличие от метода Бройдена, сработает, а вычисление для среды  $D$  будет очень дорогим. Таким образом, начальное значение  $H_0$  обычно равно  $I$ , возможно, в масштабе. Во всех случаях должна обеспечиваться положительная определенность ( $B$  есть, когда есть  $H$ ).

Выбор масштаба приводит к вопросу о том, каким должен быть первый шаг. Некоторые подходы:

- ◆ шаг наискорейшего спуска достаточно корректен по масштабу, и поиск по линии отрегулирует результирующий размер шага, чтобы сделать его длиннее или короче. Но градиент только дает направление, и неизвестно, как долго в этом направлении он будет уменьшаться. Также слишком маленькое  $\Delta x$  может привести к числовым проблемам со слишком маленьким  $\Delta f$ , так что удвоения не произойдет. То есть при сходимости  $\nabla f \rightarrow 0$  и в случае  $\|\Delta x\|_2 = \|\nabla f\|_2$  относительное  $y$  может оказываться ниже  $\varepsilon_{\text{machine}}$  из-за плохой обусловленности оптимизации вблизи минимума. Это является проблемой для запуска в точках с малым градиентом. Но предполагая, что важна только информация о второй производной, производная по направлению увеличивается линейно с неизвестной константой, поэтому после шага размера  $O(\|\nabla f\|_2)$  она станет равной 0. Отсутствие явного масштабирования предполагает, что эта константа  $= 1$ . Масштабирование градиента, как обсуждается позже, вероятно, позволит дополнительно улучшить ситуацию, но для первых шагов следующий вариант более безопасен;
- ◆ если дана пользовательская оценка размера первого шага  $s$ , масштабируйте направление градиента на  $s / \|\nabla f\|_2$  (см. [24.32]). Разумной стратегией является использование  $s = \max(1, \|x\|_2) / 10$  или подобного значения, что позволяет снова защитить большие значения  $x$  и быть намного выше плохо обусловленного размера шага; 10 — это логическое значение, которое, по-видимому, ведет к разумному изменению  $x$ . Подобные значения хорошо подходят для первых шагов методов, которые не

используют градиенты, но не сработают в случае перезапусков, потому что такой шаг, вероятно, будет слишком большим. Но в последнем случае хорошо работает другая стратегия — например, определение размера последнего улучшающего изменения.

У методов с состоянием, вроде метода BFGS, значения  $H_0$  важны для других шагов, поэтому их нужно улучшать только для выполнения обновлений. После вычисления шага (масштабируется независимо), но перед обновлением предположим, что  $H_0 = \frac{I}{p_k y_k^2}$ ,

где  $k$  — последний шаг. Это хорошо работает согласно [24.32]. В работе [24.15] приведено некоторое теоретическое обоснование.

*Алгоритм L-BFGS* уменьшает затраты памяти, т. к. использует только  $m$  последних обновлений, как метод Бroyдена с ограниченной памятью. Некоторые отличия:

- ◆ направление спуска гарантировано, потому что применяются положительно определенные обновления;
- ◆ обновление имеет ранг 2, и для него разработана специальная формула.

L-BFGS сохраняет  $j \leq m$  последних  $s_i$  и  $y_i$  в очереди и использует их для рекурсивного вычисления  $H_k$ . BFGS сохраняет всю историю поиска. Таким образом, еще одним его преимуществом является сопротивление накоплению плохо обусловленного состояния, т. е.  $B$  может перестать быть положительно определенным численно. Общая проблема заключается в том, что любой метод оптимизации, который обновляет некоторое состояние между итерациями, рискует сделать это состояние плохо обусловленным. В литературе часто мимоходом утверждается, что экспериментально BFGS и L-BFGS работают одинаково (например, некоторые сравнения приведены в работе [24.25]), но только алгоритм BFGS наследует сверхлинейную сходимость метода Ньютона. При  $m = \infty$  эти два параметра эквивалентны, но на практике побеждают малые значения  $m$ , как показано далее. Таким образом, L-BFGS является предпочтительным методом, хотя BFGS тоже часто используется и упоминается.

Экспериментально  $m \in [3, 20]$ , хороший выбор  $m = 8$  (см. [24.32]). Поскольку очередь использует вектор для хранения, чтобы избежать потерь, удобнее выбирать степень двойки, поэтому лучше всего подходят значения 4, 8 и 16.

Вычисляйте  $d = H_k \nabla f$  без явного формирования  $H$ . Поскольку матрица  $H_{k-j}$  является диагональной матрицей, можно расширить формулу обновления, вычисляя также вычисления  $d$  из  $H_{k-j}$  и сохраненных  $s_i$  и  $y_i$ , так что каждая итерация будет затрачивать время и пространство  $O(nm)$  (см. [24.32]):

1.  $d \leftarrow \nabla f$ .
2.  $\forall i \in [k-1, k-j]:$
3.     сохранить  $a_i = p_i s_i d$ ;
4.      $d- = a_i y_i$ .
5.  $d^* = H_{k-j}$ .
6.  $\forall i \in [k-j, k-1]:$
7.      $d+ = s_i (a_i - p_i y_i d)$ .

Как и метод Ньютона, L-BFGS дает хорошо масштабированные направления, поэтому поиск линии начинается с шага = 1. Алгоритму требуется дополнительная логика надежности, помимо той, что предлагается в работе [24.32], потому что здесь акцент делается на использовании градиентов конечных разностей вместо фактических, которые традиционно считаются доступными. Даже при наличии аналитических градиентов в случае очень плохо обусловленного гессиана ничто не помогает, потому что чистый градиентный спуск будет слишком медленным, а направлениям Ньютона может потребоваться большой угол, который будет искажен числовым шумом. У числовых градиентов проблема намного хуже — базовые алгоритмы часто не могут продвинуться дальше некоторой точности, например 2–4 знаков после запятой. Но в моем эксперименте это происходит при очень небольшом количестве вычислений функции (< 10 000). Так что, учитывая, что на практике чаще всего используется гибридный алгоритм (обсуждаемый далее в этой главе), с технической точки зрения это не так.

Простое исправление состоит в том, чтобы перезапустить алгоритм, когда поиск строки не сможет завершиться, очищая историю поэлементно. После того как вся история будет очищена, алгоритм превратится в градиентный спуск, который наиболее численно устойчив к плохому вычислению градиента. Если и это не сработает, то при начальном размере шага, равном размеру последнего шага, алгоритм завершается. Во многих задачах этот метод дает гораздо более высокую точность, хотя и тратит весь бюджет оценки:

```
template<typename FUNCTION, typename GRADIENT, typename DIRECTIONAL_DERIVATIVE>
pair<Vector<double>, double> LBFGSMinimize(Vector<double> const& x0,
FUNCTION const& f, GRADIENT const& g, DIRECTIONAL_DERIVATIVE const& dd,
int maxEvals = 1000000, double yPrecision = highPrecEps,
int historySize = 8, bool useExact = true)
{
    typedef Vector<double> V;
    Queue<pair<V, V> > history;
    pair<V, double> xy(x0, f(x0));
    V grad = g(xy.first), d;
    int D = xy.first.getSize(), gEvals = g.fEvals(D);
    maxEvals -= 1 + gEvals;
    double lastGoodStepSize = max(1.0, norm(x0))/10;
    while(maxEvals > 0)
    { // возврат с использованием d для получения достаточного спуска
        if(history.getSize() == 0) d = grad * (-lastGoodStepSize/norm(grad));
        pair<V, double> xyOld = xy;
        if(goldenSectionLineSearch(f, grad, dd, xy.first, xy.second, maxEvals,
            d, yPrecision, useExact))
        { // неудача
            if(history.getSize() > 0)
            { // попытка перезапуска путем очистки истории
                history.pop();
                continue;
            }
            else break; // шаг градиентного спуска не удался
        }
    }
}
```

```

else lastGoodStepSize = norm(xy.first - xyOld.first); // успех
if((maxEvals -= gEvals) < 1) break; // выход за границы
V newGrad = g(xy.first);
if(history.getSize() >= historySize) history.pop();
history.push(make_pair(xy.first - xyOld.first, newGrad - grad));
// двойная рекурсия, обновление d
d = grad = newGrad;
Vector<double> a, p;
int last = history.getSize() - 1;
for(int i = last; i >= 0; --i)
{
    double pi = 1/dotProduct(history[i].first, history[i].second),
        ai = dotProduct(history[i].first, d) * pi;
    d -= history[i].second * ai;
    a.append(ai);
    p.append(pi);
} // начальное значение Гессiana масштабируется по диагонали
d *= 1/(dotProduct(history[last].second, history[last].second) *
    p[last]);
for(int i = 0; i < history.getSize(); ++i)
{
    double bi = dotProduct(history[i].second, d) * p[last - i];
    d += history[i].first * (a[last - i] - bi);
}
d *= -1;
}
return xy;
}

```

Алгоритм L-BFGS линейно сходится к локальному минимуму, если (см. [24.25]):

- ◆ функция  $f$  выпукла и дважды непрерывно дифференцируема;
- ◆  $\forall z \in \mathbb{R}^n \exists m, M > 0$  такое, что  $m\|z\|^2 \leq z\nabla^2 f z \leq M\|z\|^2$ ;
- ◆  $\kappa(H_0) < \infty$ ;
- ◆ линейный поиск удовлетворяет условиям Вульфа.

Эти условия нужны лишь для успокоения. На практике алгоритм очень надежен и обычно сходится, даже если  $f$  не удовлетворяет условиям в каждой точке. Но даже если это так, ограничения на количество итераций не существует.

Если градиента нет, он вычисляется по конечным разностям. Эксперименты показывают, что этот способ работает хорошо, пока оценка не слишком неточна, но теоретического обоснования этому нет:

```

template<typename FUNCTION> struct GradientFunctor
{
    FUNCTION f;
    GradientFunctor(FUNCTION const& theF): f(theF) {}
    Vector<double> operator() (Vector<double> const& p) const
    {return estimateGradientCD(p, f);}
    int fEvals(int D) const{return 2 * D;}
};

```

```
template<typename FUNCTION> struct DirectionalDerivativeFunctor
{
    FUNCTION f;
    DirectionalDerivativeFunctor(FUNCTION const& theF): f(theF) {}
    double operator()(Vector<double> const& x, Vector<double> const& d)
        const{return estimateDirectionalDerivativeCD(x, f, d);}
    int fEvals()const{return 2;}
};
```

Для градиентов с конечной разностью числовой шум в конечном итоге будет давать случайные направления, что ограничивает предельно достижимую точность у больше, чем любые другие факторы. Возможные результаты для сходящегося алгоритма:

1. Значение  $f$  локально оптимально с некоторой точностью.
2.  $\nabla f$  слишком мало.
3. Закончился бюджет оценки.
4. Ошибка градиента настолько велика, что предполагаемое направление наибольшего спуска не является спуском, а алгоритм преждевременно останавливается.

Для пункта (2) градиент делится на  $\max(1, |f(x)|)$  (см. [24.15]). Эта единица позволяет получать масштабированные оценки градиента только для больших значений  $f$ . Опасность заключается в том, что при аддитивных постоянных факторах в  $f(x)$  это приведет к недооценке, но лучшего решения все равно нет.

Обычно случай (4) не происходит. Учитывая случай (1), в точной арифметике также имеет место случай (2). Но в моих тестах это не всегда так, т. е. масштабированное  $\nabla f$  может быть порядка 0,01, но не близко к тому, что можно было бы считать точной сходимостью. Таким образом, принимать решение о сходимости проблематично, если основываться только на градиенте конечной разности. Более надежным индикатором является то, что поиск линии с оценкой направления наискорейшего спуска завершается ошибкой, как это реализовано.

Используемые здесь тестовые функции взяты из работ [24.15 и 24.28]. Используются только те из них, которые имеют известное оптимальное значение во всех измерениях и, по-видимому, не имеют локальных минимумов (некоторые исключения: Trig, Wood и Guld R&D). Но все это задачи наименьших квадратов, и они могут быть нерепрезентативными, поскольку метод наименьших квадратов имеет свою особую структуру.

Методы на основе градиента в одном измерении не имеют смысла из-за необходимости применения линейного поиска, основанного на золотом сечении, или других приближенных методов поиска, которые можно запускать напрямую. Основное преимущество градиента заключается в задании направления спуска, но в одном измерении можно дешево попробовать оба варианта. Однако можно также дешево реализовать метод центральной разности Ньютона (с обратным отслеживанием), повторно используя во 2-й производной оценивание оценки 1-й производной.

## 24.7. Методы, не требующие вычисления производных

У дифференцируемой  $f$  выбрать направление спуска легко, даже не зная  $\nabla f$ , потому что до тех пор, пока  $d\nabla f \neq 0$ , направление  $-d$  или  $d$  является направлением спуска. Но это неверно для недифференцируемых  $f$ .

*Алгоритм Нелдера — Мида* — это один из наиболее успешных методов для функций «черного ящика», особенно при  $D < 10$ , где он очень эффективен. Алгоритм работает с начальным *симплексом*, который представляет собой гипертреугольник из  $D + 1$  точек и состоит из предполагаемой точки и ее смещений в каждом ортогональном направлении. Здесь используется  $uniform01 \times s \max(1, |x_i|) / 10$ , где  $s$  = начальный размер шага, по умолчанию равный 1. Важными являются точки с лучшими, худшими и предпоследними по качеству значениями. На каждой итерации алгоритм вытягивает симплекс от худшего к лучшему (рис. 24.2), используя:

- ◆ *масштабирование* — заменяет наихудший случай выпуклой комбинацией самой себя и центроидой других точек. Когда масштабный коэффициент =  $-1$ , в результате получается *отражение*;
- ◆ *уменьшение* — сдвигает все точки на  $1/2$  пути к лучшему.

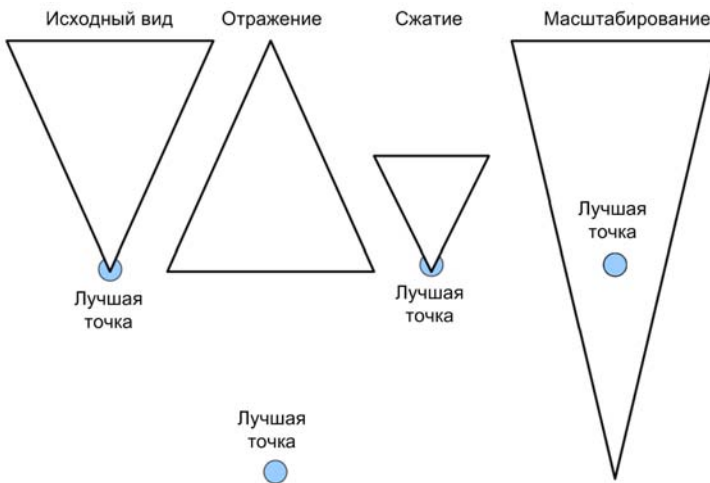


Рис. 24.2. Возможные преобразования текущего симплекса

```
template<typename FUNCTION> struct NelderMead
{
    FUNCTION f;
    int D;
    Vector<double> vertexSum; // инкрементная центроида
    typedef pair<Vector<double>, double> P;
    Vector<P> simplex;
    double scale(P& high, double factor, int& maxEvals)
    {
        P result = high;
```

```

    // сочетание верхних точек и центроиды оставшихся вершин
    // centroid = (vertexSum - high)/D и
    // result = centroid * (1 - factor) + high * factor
    double centroidFactor = (1 - factor)/D;
    result.first = vertexSum * centroidFactor +
        high.first * (factor - centroidFactor);
    result.second = f(result.first);
    --maxEvals;
    if(result.second < high.second)
    { // масштабирование принимается, если оно улучшает результат
        vertexSum += result.first - high.first;
        high = result;
    }
    return result.second;
}

public:
    NelderMead(int theD, FUNCTION const& theFunction = FUNCTION()):
        D(theD), f(theFunction), simplex(D + 1){assert(D > 1);}
};

```

Основная логика:

1. Инициализировать симплекс.
2. До схождения:
3. Найти лучшую, вторую лучшую и худшую точки.
4. Схождение достигнуто, если лучшая  $\approx$  худшей.
5. Попробовать отразить.
6. Если отраженная точка лучше, чем лучшая, попробуйте масштабировать новую лучшую вдвое.
7. В противном случае попробуйте масштабировать на  $1/2$ .
8. Если результат хуже следующего худшего, уменьшаем все точки в лучшем направлении.

Для эффективности, если вам не нужна высокая точность, используйте значение  $\varepsilon = 0,001$  или близкое с правильно масштабированным значением  $f$ . Если лучшая точка сходится линейно и обновляется каждые  $D$  итераций, требуется  $O(D \log(1 / \varepsilon))$  оценок. Но поскольку по умолчанию требуется высокая точность, можно выделить  $10^6$  оценок:

```

P minimize(Vector<double> const& initialGuess, int maxEvals = 1000000,
    double yPrecision = highPrecEps, double step = 1)
{ // инициализация симплекса
    vertexSum = initialGuess;
    for(int i = 0; i < D; ++i) vertexSum[i] = 0;
    for(int i = 0; i <= D; ++i)
    {
        simplex[i].first = initialGuess;
        if(i > 0) simplex[i].first[i - 1] += GlobalRNG().uniform01() *
            step * max(1.0, abs(initialGuess[i - 1]))/10;
        simplex[i].second = f(simplex[i].first);
        --maxEvals;
    }
}

```



```

    vertexSum += simplex[i].first;
}
for(;;)
{ // вычисление high, low и nextHigh, которые должны быть разными
    int high = 0, nextHigh = 1, low = 2;
    if(simplex[high].second < simplex[nextHigh].second)
        swap(high, nextHigh);
    if(simplex[nextHigh].second < simplex[low].second)
    {
        swap(low, nextHigh);
        if(simplex[high].second < simplex[nextHigh].second)
            swap(high, nextHigh);
    }
    for(int i = 3; i <= D; ++i)
    {
        if(simplex[i].second < simplex[low].second) low = i;
        else if(simplex[i].second > simplex[high].second)
        {
            nextHigh = high;
            high = i;
        }
        else if(simplex[i].second > simplex[nextHigh].second)
            nextHigh = i;
    } // проверка, не достигнуто ли схождение
    if(maxEvals <= 0 || !isELess(simplex[low].second,
        simplex[high].second, yPrecision)) return simplex[low];
    // попытка выполнить отражение
    double value = scale(simplex[high], -1, maxEvals);
    // попытка удвоить, если результат лучше, чем low
    if(value <= simplex[low].second) scale(simplex[high], 2, maxEvals);
    else if(value >= simplex[nextHigh].second)
    { // попытка разделить отраженное/неотраженное пополам, если
        // значение принято/отклонено
        double yHi = simplex[high].second;
        if(scale(simplex[high], 0.5, maxEvals) >= yHi)
        { // сократить все, чтобы избавиться от высокой точки
            vertexSum = simplex[low].first;
            for(int i = 0; i <= D; ++i) if(i != low)
            {
                vertexSum += simplex[i].first = (simplex[i].first +
                    simplex[low].first) * 0.5;
                simplex[i].second = f(simplex[i].first);
                --maxEvals;
            }
        }
    }
}
}
}

```

Алгоритм Нелдера — Мида выбирает хорошие направления для типичных  $f$  и всегда сходится к чему-то, что не обязательно должно быть локальным минимумом. Он может

остановиться, потому что объем симплекса может стать слишком маленьким, и не восстановиться, особенно для  $D > 10$ . Самый простой способ справиться с этим — перезапустить результат, пока процесс не прекратится. По умолчанию максимальное количество перезапусков равно 10. Если для схождения требуется большее, то алгоритм Нелдера — Мида, вероятно, неэффективен для решения задачи. Но даже если он не может сойтись, он обычно дает значительный прирост  $f$ -значения после относительно небольшого количества оценок:

```
P restartedMinimize(Vector<double> const& initialGuess,
    int maxEvals = 100000, double yPrecision = highPrecEps,
    int maxRepeats = 10, double step = 1)
{
    P result(initialGuess, numeric_limits<double>::infinity());
    while(maxRepeats--)
    {
        double yOld = result.second;
        result = minimize(result.first, maxEvals, yPrecision, step);
        if(!isELess(result.second, yOld, yPrecision)) break;
    }
    return result;
}
```

Последующие перезапуски более эффективны, если близки к решению. Хороший гибридный метод — запустить сперва L-BFGS. Обычно удастся быстро найти решение, а затем повысить точность. Но поскольку алгоритм Нелдера — Мида использует память  $O(D^2)$ , он годится до значений  $D = 200$  или около того:

```
template<typename FUNCTION> pair<Vector<double>, double> hybridLocalMinimize(
    Vector<double> const& x0, FUNCTION const& f, int maxEvals = 1000000,
    double yPrecision = highPrecEps)
{
    GradientFunctor<FUNCTION> g(f);
    DirectionalDerivativeFunctor<FUNCTION> dd(f);
    int D = x0.getSize(), LBFGSsevals = D < 200 ? maxEvals/2 : maxEvals;
    pair<Vector<double>, double> result = LBFGSMinimize(x0, f, g, dd,
        LBFGSsevals, yPrecision);
    if(D > 1 && D < 200)
    {
        int nRestarts = 30;
        NelderMead<FUNCTION> nm(x0.getSize(), f);
        result = nm.restartedMinimize(result.first,
            (maxEvals - LBFGSsevals)/nRestarts, highPrecEps, nRestarts);
    }
    return result;
}
```

Это, вероятно, самый эффективный общий минимизатор «черного ящика». Ни один из компонентов не дает никаких теоретических гарантий. Даже если теоретические условия для L-BFGS соблюдаются, в нем все равно присутствует некоторое приближение из-за конечно-разностных градиентов. Но на практике L-BFGS быстро найдет оптимум, и, если он не сработает, Нелдер — Мид, скорее всего, добьется успеха благодаря лучшему значению  $f$ .

Для дифференцируемых  $f$  с неизвестными аналитическими производными теоретическую сходимость в реализованном виде дают только *методы поиска закономерностей* (см. [24.32, 24.23, 24.14, 24.2]). Например, алгоритм *поиска по компасу*:

1. Начать с некоторого начального размера шага  $s$ , здесь  $0.1\max(1, \|x\|_2)$ .
2. До сходимости, когда  $s$  становится слишком малым на основе точности  $x$ , равной  $\varepsilon$ :
3. Для любого координатного направления:
  4. Если для  $s \times$  направление достигнуто улучшение (обсуждается позже):
    5. Принять шаг.
    6. Удвоить  $s$ .
    7. Завершить цикл.
    8. Уменьшить  $s$  вдвое.

Метод компаса кажется особенно эффективным для пошагового вычисления  $f$ , т. к. он всегда работает с одним измерением. В типичной реализации перебираются все направления в случайном порядке в цикле, но поскольку улучшение направления может быть найдено быстро, алгоритм может замедлиться из-за генерации перестановок. Таким образом, простое решение состоит в том, чтобы переставить все 2D-оценки. Но поскольку необходимо опросить все измерения, перестановка производится только после завершения цикла.

Этот алгоритм интересен теоретически хотя бы потому, что гарантируется сходимость для дифференцируемой  $f$  (см. [24.2]) и уникальная стратегия поиска с возвратом. К со-

Функция	Компас		UnimodalCD		NelderMead		RestartedNelder		LBFGSMinimize		HybridLocalMin	
	Ошибка	Оценки	Ошибка	Оценки	Ошибка	Оценки	Ошибка	Оценки	Ошибка	Оценки	Ошибка	Оценки
ExtendedRosenbrock2	-11.16	80746	-13.66	271	-13.41	193	-13.60	384	-8.08	47371	-13.66	47705
ExtendedPowellSingular4	-4.75	1000008	-4.01	1000010	-13.22	729	-13.60	1138	-4.83	1000052	-13.39	500838
HelicalValley	-11.60	27125	-11.39	398978	-13.39	489	-13.44	834	-13.03	8240	-13.87	8913
VariableDimensionF2	-15.65	200	-13.90	3738	-13.52	251	-13.60	402	-14.19	1032	-14.19	1372
LinearFFullRank2	-13.65	425	-14.47	337	-13.39	233	-13.60	399	-15.65	1000000	-15.65	500347
BrownBadScaled	-7.80	1534	-8.61	473	-13.59	435	-13.98	616	-14.00	1399	-14.15	1852
Beale	-12.66	2911	-12.60	22494	-13.92	216	-13.82	618	-10.03	1996	-13.78	2339
BiggsExp6	-2.23	1000012	-2.23	1000022	-3.69	2474	-2.22	1624	-4.61	1000055	-13.35	503479
ExtendedRosenbrock10	-5.60	1000020	-1.82	1000055	-10.82	6992	-13.19	13000	-8.45	64759	-12.98	71703
ExtendedPowellSingular12	-3.58	1000024	-3.26	1000048	0.00	8256	-11.45	27279	-4.62	1000024	-9.10	543961
VariableDimensionF10	-11.59	334075	-11.76	749456	-11.63	8405	-13.07	13357	-13.33	5741	-13.40	8140
LinearFFullRank10	-13.38	7997	-13.90	1921	-12.81	4199	-13.41	7230	-15.65	397	-15.65	3019
ExtendedRosenbrock30	-0.54	1000060	-0.63	1000067	-0.58	35433	-7.38	463063	-7.52	134706	-10.96	259814
ExtendedPowellSingular32	-2.52	1000064	-2.58	1000096	0.00	159722	-5.37	618889	-4.61	1000075	-7.19	685426
VariableDimensionF30	0.00	1000060	0.00	1000088	0.00	39674	-6.48	627111	-11.57	33397	-12.21	86511
LinearFFullRank30	-13.06	48841	-14.08	6151	0.00	49749	-11.30	374158	-15.65	601	-15.65	10209
ExtendedRosenbrock100	0.00	1000200	-0.11	1000204	-0.18	1000200	-3.49	1000203	-7.27	80115	-8.02	562807
ExtendedPowellSingular100	-1.30	1000200	-1.78	1000222	0.00	1000200	-4.37	1000201	-4.85	1000105	-5.41	1000052
VariableDimensionF100	0.00	1000200	0.00	1000246	0.00	334728	-1.41	1000200	-8.93	154946	-9.52	617026
LinearFFullRank100	-15.65	34153	-13.46	21101	0.00	1000200	-9.98	700649	-15.65	1306	-15.65	34638
Средние ранги	6.6	7.0	6.7	7.6	7.2	3.5	4.8	5.5	3.7	5.4	1.9	5.7

**Рис. 24.3.** Производительность некоторых основных методов на нескольких задачах (среди многих других проверенных) с разными не слишком большими размерностями. Ошибка является относительной и абсолютной в десятичных цифрах от правильного ответа 0. Гибридный алгоритм, безусловно, является лучшим методом «черного ящика»

жалению, практическая производительность оставляет желать лучшего — алгоритм часто не сходится из-за исчерпания вычислений, поэтому его реализация не представлена. Координатный спуск на практике кажется лучшим выбором для пошагового  $f$  (рис. 24.3).

Для больших  $D$  компас и координатный спуск работают намного хуже, чем L-BFGS, поэтому гибридизация не имеет смысла (рис. 24.4). Их следует использовать только для постепенно вычисляемых  $f$  (возможно, даже в качестве гибрида), где они намного эффективнее. Кроме того, компас работает с прямоугольными ограничениями, что позволяет выполнять пошаговые проверки. С помощью отклонения можно легко применить общие ограничения функций «черного ящика».

Функция	Компас		UnimodalCD		LBFGSMinimize	
	Ошибка	Оценки	Ошибка	Оценки	Ошибка	Оценки
ExtendedRosenbrock1000	0.00	1002000	0.00	1002020	-6.75	1001219
ExtendedPowellSingular1000	0.00	1002000	0.00	1002014	-5.64	1001225
VariableDimensionF1000	0.00	1002000	0.00	1002005	-7.15	10365
LinearFFullRank1000	0.00	1002000	-12.97	230001	-13.85	10429
ExtendedRosenbrock10000	0.00	1020000	0.00	1020041	-5.31	1004075
ExtendedPowellSingular10000	0.00	1020000	0.00	1020069	-4.37	1003726
VariableDimensionF10000	0.00	1020000	0.00	1020051	-5.85	80309
LinearFFullRank10000	0.00	1020000	-4.17	1020009	-15.65	100315
Средние ранги	2.3	2.1	2.0	2.9	1.0	1.0

Рис. 24.4. Производительность масштабируемых методов для больших  $D$ . L-BFGS работает хорошо. Остальные — нет, поэтому трудно создать хороший гибридный метод

Конкретные  $f$  из важных задач обычно минимизируются с помощью специальных методов, таких как *обратное распространение* для нейронных сетей (см. главу 26. *Машинное обучение: классификация*).

## 24.8. Негладкая минимизация

Некоторые  $f$ , такие как  $|x|$ , непрерывны, но недифференцируемы на всей области определения. Производных/градиентов у такой функции не существует, а обычные правила исчисления не применяются. Но логика рассуждений все равно похожа. В частности, производная и градиент обобщаются до *субградиента*, которым в одномерном случае является любая касательная к функции (рис. 24.5).

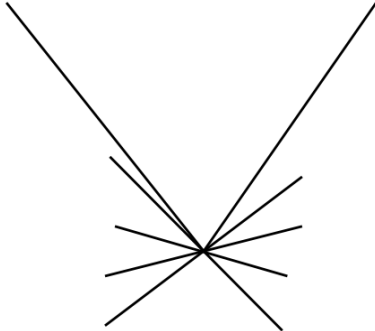


Рис. 24.5. Возможные субградиенты на вершине  $f(x) = |x|$

Для функции  $f(x) = |x|$  субградиент задается выражением  $f'(x) = x > 0?1:-1$ . Это не единственный возможный выбор, но его легко запрограммировать. Не имеет значения, какой конкретный субградиент выбран, потому что в точке, не поддающейся дифференциации, происходит какое-то необычное событие.

Для  $D > 1$  нужно найти касательную плоскость и вычислять по одному измерению за раз, как в случае с градиентами. В большинстве случаев расчет можно проводить по следующему правилу: пусть  $f(x) = \text{if}(c(x), g(x), h(x))$ ; тогда  $f'(x) = \text{if}(c(x), g'(x), h'(x))$  с произвольным разрешением в точке разрыва. В противном случае следуйте обычному цепному правилу исчисления.

Теория устроена намного сложнее. Возможно, самое простое (хотя даже оно не так уж просто) введение приведено в работе [24.4]. Наиболее хорошо изучены случаи, когда  $f$  является выпуклой (где субградиенты технически определены) или локально липшицевой (*субградиент Кларка*), хотя на практике не проблема, если можно вывести субградиент. Например, для вогнутой функции  $f(x) = 1 - |x|$  можно следовать той же логике. Для более общего случая в работе [24.34] приводится очень математическое описание и правила исчисления для нескольких вариантов определения негладких производных. Основная идея состоит в том, что только в выпуклом/вогнутом случае существуют хорошие правила исчисления, а в других — не хватает ограничений вроде необходимости замены равенств неравенствами и т. д.

Все возможные субградиенты в точке образуют *субдифференциальное множество*. У него есть ряд полезных с точки зрения оптимизации свойств — в частности, для выпуклой  $f$  оно содержит 0 в локальном минимуме.

При использовании субградиентов с существующими алгоритмами на основе градиента возникают проблемы:

- ◆ направление наибольшего спуска, вычисленное по субградиенту, не обязательно должно вести к спуску. Например, представим двумерную функцию  $f(x) = 2|x_1| + x_2$ . Тогда при  $\nabla f = (x_1 > 0?2: -2, 1)$  наискорейший спуск с линейным поиском будет сходиться к точке  $(0, a)$  для некоторого конечного  $a$ , зависящего от начальной точки, даже если решение равно  $(0, -\infty)$ . Проблема в том, что первая координата с большей производной препятствует перемещению второй, и, в отличие от гладкого случая, первая компонента градиента  $\nrightarrow 0$ , и существует  $f$  и  $x$ , где лишь малый диапазон углов дает спуск. Нужно направление, которое образует правильный угол со всеми членами субдифференциального множества, т. е. скалярное произведение любого элемента (направление, элемент)  $< 0$ ;
- ◆ если существует направление спуска, то в точке возврата условие достаточного спуска не имеет смысла из-за возможного резкого уменьшения производной по этому направлению.

Тот факт, что конкретный субградиент не ведет к спуску, означает, что в задаче недостаточно информации. Возможны следующие решения:

- ◆ попробовать выбрать случайное направление, но это может быть очень неэффективно для больших  $D$ ;
- ◆ надеяться, что близлежащая точка ведет к спуску, и попробовать использовать ее субградиент, но он, вероятно, приведет обратно к исходной точке, если спуск там не найдется;

- ◆ можно использовать несколько субградиентов как разные точки, чтобы вывести лучший субградиент, — возможно, тот, у которого наименьшая норма или какое-то другое свойство, но неясно, как их собрать.

Было разработано несколько конвергентных стратегий:

- ◆ *субградиентный спуск* — не выполнять линейный поиск и принимать ухудшающиеся шаги. При определенных ограничениях размера шага (подобных условиям Роббинса — Манро для стохастической оптимизации, обсуждаемым далее в этой главе) можно получить доказуемую сходимость. Для реализации, возможно, придется использовать то же субградиентное масштабирование, что и для первого шага L-BFGS, но ограничить норму константой — например, 1, чтобы предотвратить расхождение. Несмотря на простоту, этот алгоритм очень медленный. Возможно, его удастся использовать для спуска в случаях, когда обычные алгоритмы вроде L-BFGS застревают, но это, похоже, не рассматривалось в литературе;
- ◆ *пакетные методы* и их варианты — например, L-BFGS, позволяют сохранять набор прошлых субградиентов и пытаются использовать их с целью найти субградиент спуска.

Обзор этих методов и их вариантов приведены в работе [24.4]. Судя по описанным там тестам, алгоритмы не полностью надежны. Алгоритмы все еще разрабатываются — например, см. [24.22].

Если  $f$  к тому же выпукла, проксимальные алгоритмы и варианты оказываются эффективными. Общий обзор приведен в работах [24.5 и 24.20], где алгоритмы применялись к лассо-регрессии. Как вариант можно использовать спуск по координатам, который также хорошо работает для выпуклых  $f$  и теоретически эффективен в особых случаях, например в лассо-регрессии. См. также [24.31]. В центре внимания текущих исследований находятся алгоритмы, не являющиеся алгоритмами «черного ящика», которые реализуют, по крайней мере, способность вычислять субградиент. Эти знания имеют решающее значение для производительности. Более глубокое обсуждение приведено по ссылкам.

Алгоритм BFGS с возвратом хорошо работает как эвристика (см. [24.24]). Согласно экспериментам, описанным в работе [24.40], он работает лучше, чем некоторые из ранее упомянутых специализированных методов. Существует множество оправданий хорошей работы (дополнительная информация приведена в работе [24.24]):

- ◆ липшицевы  $f$  дифференцируемы почти всюду, поэтому вероятность найти недифференцируемую точку мала, если она не является решением;
- ◆ недифференцируемые  $f$  — это пределы плохо обусловленных дифференцируемых  $f$ , для которых метод Ньютона и его варианты работают достаточно хорошо.

Таким образом, представленный гибридный алгоритм непрерывной оптимизации по-прежнему кажется предпочтительным методом для функций «черного ящика», несмотря на использование сильного линейного поиска Вульфа с L-BFGS, в отличие от того, что сделано в работе [24.24]. Использование конечных разностей обеспечивает дополнительную гладкость, если  $f$  является локально липшицевой, однако имейте в виду, что градиент конечной разности не обязательно должен быть точным, поскольку остаток ряда Тейлора не работает. Но центральная разность, по сути, вычисляет конечно-разностную производную в ближайшей точке. Кроме того, сильная реализация поиска по линии Вульфа по-прежнему будет принимать точку после сужения интервала, что

должно быть полезно. Возможно также, что основная причина использования гибридного алгоритма заключается в том, что у функций «черного ящика» неизвестно, являются ли они полностью дифференцируемы. Она может даже не иметь субградиентов и т. д.

## 24.9. Глобальная минимизация

Алгоритмы, описанные в предыдущих разделах, находят локальные, а не глобальные минимумы, и обычно предполагают дифференцируемость. К сожалению, для произвольных  $f$  оптимизация доказуемо невозможна, по крайней мере в худшем случае (см. [24.31]). У функции может быть произвольно маленькая лунка для гольфа на плоском плато, которую практически невозможно обнаружить.

Основное полезное предположение состоит в том, что  $f$  является липшицевой. Тогда поиск по сетке будет эффективен для нахождения приблизительной области глобального минимума для малых  $D$ . Создайте равномерную сетку с известными границами поля и запустите алгоритм в каждой точке сетки. Задача, таким образом, превращается в оптимизацию дискретного набора, которую мы обсудим далее в этой главе. Равномерной сеткой можно покрыть всю область с точностью до  $\varepsilon$ , но этот вариант неэффективен. Из-за предположения о том, что функция липшицева, решение не может быть далеко от ближайшей точки. Но потребуются большое число разбиений (степенное по  $D$ ), чтобы получить конкретное значение  $\varepsilon$ . В машинном обучении для  $D \leq 3$  часто используют именно поиск по сетке.

Обычно более случайный поиск эффективнее выполняется на основе какого-либо семплера независимых случайных данных, особенно для больших  $D$ . Когда информации нет, можно использовать вариации Леви, но часто известны границы поля, специфичные для предметной области, которые уменьшают область выборки и обеспечивают однородную выборку. Для неограниченного семплера нужно использовать больший коэффициент масштабирования, чтобы учесть тот факт, что семплер должен быть глобальным.

Предположим, что семплер выполняет разумную проверку ошибок, т. е. генерирует точки с конечной нормой. Но  $f$  может возвращать NaN или  $\infty$ , что может говорить о нарушении ограничения, и алгоритмы должны учитывать возможность такого случая, например игнорировать точки NaN, рассматривая их как  $\infty$ :

```
class UnboundedSampler
{
    Vector<double> center;
    double scaleFactor;
public:
    UnboundedSampler(Vector<double> const& theCenter, double theScaleFactor =
        10): center(theCenter), scaleFactor(theScaleFactor)
        {assert(isfinite(norm(theCenter)));}
    Vector<double> operator () () const
    {
        Vector<double> next = center;
        for(int i = 0; i < next.getSize(); ++i) next[i] = (*this)(i);
        return next;
    }
}
```

```

double operator() (int i) const
{ // проверка выборки на конечность
    double result = numeric_limits<double>::infinity();
    while (!isfinite(result)) result = center[i] + max(1.0, abs(center[i])) /
        10 * scaleFactor * GlobalRNG().Levy() * GlobalRNG().sign();
    return result;
}
};

class BoxSampler
{
    Vector<pair<double, double> > box;
public:
    BoxSampler(Vector<pair<double, double> > const& theBox): box(theBox){}
    Vector<double> operator() () const
    {
        Vector<double> next(box.getSize());
        for(int i = 0; i < box.getSize(); ++i)
            next[i] = GlobalRNG().uniform(box[i].first, box[i].second);
        return next;
    }
    double operator() (int i) const
        {return GlobalRNG().uniform(box[i].first, box[i].second);}
};

```

Случайный поиск хорош для приближения к области с глобальным минимумом, но обычно требуется найти локальный минимум в уже известной области, поэтому сразу после этого нужно выполнять локальный поиск (см. [24.51]). Это делается у всех протестированных алгоритмов. Бюджет на оценку делится примерно как 9 к 1 в пользу разведки. Этот метод является отправной точкой для дальнейших улучшений.

Глобальная непрерывная оптимизация сильно отличается от глобальной комбинаторной оптимизации:

- ◆ у комбинаторной оптимизации локальные решения хорошо разделены дискретной заменой;
- ◆ непрерывная выборка более приемлема для случайного независимого случая. Случайный выбор до локального поиска кажется гораздо менее полезным в случае комбинаторной выборки;
- ◆ в непрерывном случае можно попробовать любое случайное направление;
- ◆ в непрерывном случае переменные, скорее всего, будут разделимыми или частично разделимыми.

Случайная выборка также подвержена проклятию размерности:

- ◆ вероятность найти точку в приемлемой области равна (приемлемая область) / (область поиска). Учтите, что (объем гиперсферы) / (объем ограничивающего гиперкуба)  $\rightarrow 0$  экспоненциально быстро даже в больших приемлемых областях, а это проблема. Случайный поиск эффективен только тогда, когда очень мало переменных оказывают существенное влияние на значение функции;
- ◆ поскольку хочется попасть в приемлемую область с высокой вероятностью  $= 1 - \alpha$  при равномерной выборке, нужно  $(\log(1/\alpha) \cdot \text{домен поиска}) / (\text{приемлемая область})$  вы-



борок (см. [24.26]). Здесь вводится упрощение  $\log(1 - x) \approx -x$  для малых  $x$  — согласно [24.51];

- ◆ для  $f(x) = |x|$  на  $[-1, 1]$  распределение значений  $f$  является стандартным равномерным. Таким образом, по порядковой статистике (см. [24.49]) минимум из  $n$  следует бета-распределению  $(1, n)$  и имеет ожидаемое значение  $1/(n + 1)$ . Таким образом, здесь сходимость ожидаемого значения составляет  $O(1/n)$  и характерна для 1D, поскольку в ограниченных областях распределения для различных  $f$  значения обычно близки к однородным;
- ◆ для  $f(x) = 12 / D \sum |x|$  на  $[-1, 1]^D$  распределение суммы асимптотически нормально со средним значением  $1/2$  и дисперсией 1. Нормальная статистика минимального порядка равна приблизительно  $\mu - \sigma \Phi^{-1}\left(\frac{1}{n}\right)$  (см. [24.42]). Поскольку хвост спадает экспоненциально, для больших  $D$ , где справедливо нормальное приближение,  $\Phi^{-1}\left(\frac{1}{n}\right)$  фактически постоянна. Например, типичное  $n = 10^6$  дает только около 4,75, а меньшее  $10^3$  — около 3,09. У других распределений появляются аналогичные проблемы;
- ◆ как обсуждалось в главе 16. *Комбинаторная оптимизация*, вы встретитесь с проблемой центрального предела в случаях, когда решения имеют тенденцию быть близкими к среднему значению. Фактически сначала около  $1/2$  переменных приведут к выигрышу, а остальные — к убытку. По мере продвижения оптимизации становится все меньше шансов найти случайное улучшение, потому что шансов получить плохой образец больше, чем получить хороший. Так что в лучшем случае вам придется потратить  $o(n)$  оценок на случайный поиск, а возможно,  $\sqrt{n}$  или около того.

Таким образом, при небольших  $D$  случайная выборка = поиск по сетке = поиск решения по графику  $f$ . Одним из возможных усовершенствований случайного поиска является использование зашифрованных квазислучайных последовательностей, таких как последовательность Соболя. Некоторые идеи (см. [24.9]):

- ◆ поиск по сетке неэффективно пытается искать одни и те же точки в случаях, когда некоторые переменные не имеют значения, а случайный поиск позволяет избежать этого;
- ◆ случайный поиск может быть неудачным, если выполнен слишком мало раз;
- ◆ квазислучайные последовательности не становятся неудачными, избегая некоторых областей;
- ◆ скремблирование помогает снизить вероятность неудачи.

В моих тестах скремблированный гибрид последовательности Соболя превосходит случайный поиск, но не другие алгоритмы. В работе [24.9] рекомендуется другая последовательность, но это не имеет смысла, учитывая, что многие исследования показывают превосходство последовательности Соболя в других задачах.

При наличии нерелевантных параметров случайный поиск значительно выигрывает за счет выборки большего количества различных значений релевантных параметров (см. [24.7]), но при малом бюджете он иногда будет оказываться неудачным, не попадая

в некоторые области. Таким образом, при небольшом количестве параметров и четко определенном поле поиск по сетке по-прежнему остается предпочтительным методом.

При больших  $D$  необходимо использовать смещенную выборку, чтобы получить лучшую производительность. Согласно теоремам об отсутствии бесплатного обеда, никакое конкретное смещение не помогает во всех возможных случаях, так что остается надеяться, что выбранное смещение подойдет для используемых на практике  $f$ . Таким образом, для конкретной проблемы различные методы хороши ровно настолько, насколько хороши их погрешности. Некоторые часто имеющие место предубеждения:

- ◆ многие  $f$  являются разделимыми или почти разделимыми, т. е. могут выполнять случайный поиск по одной переменной за раз;
- ◆ многие  $f$ , которые не являются разделимыми, взаимодействуют только с несколькими переменными;
- ◆ лучшая точка обычно находится не слишком далеко от любой стартовой точки, и можно улучшить алгоритм для быстрого поиска оптимальной точки;
- ◆ большинство  $f$  масштабируются так, чтобы оптимум был близок к 0, а значение  $f$  было сколь угодно плохим вдали от 0.

Таким образом, *спуск по случайным координатам* — еще один хороший эталонный алгоритм, возможно, лучший для разделимых  $f$ . Кроме того, он работает с инкрементно вычисляемыми  $f$ , — здесь последующим локальным алгоритмом является регулярный спуск по координатам. Но в случае недифференцируемой локальной минимизации случайный спуск по координатам может застрять. Тем не менее он полезен как вспомогательный метод и для пошаговой оценки:

```
template<typename INCREMENTAL_FUNCTION, typename SAMPLER>struct ContinuousMove
{
    typedef Vector<double> X;
    typedef pair<double, int> MOVE;
    INCREMENTAL_FUNCTION& f;
    SAMPLER const& s;
    ContinuousMove(INCREMENTAL_FUNCTION& theF, SAMPLER const& theS):
        f(theF), s(theS){}
    double getScore(X const& x) const {return f(f.getXi());}
    pair<MOVE, double> proposeMove(X const& x) const
    {
        int i = GlobalRNG().mod(x.getSize());
        f.setCurrentDimension(i);
        double xi = s(i), yNext = f(xi), y = f(x[i]);
        return make_pair(MOVE(xi, i), // если начальная точка равна nan,
                        // все шаги допустимы
                        isnan(y) ? 0 : yNext - y);
    }
    void applyMove(X& x, MOVE const& move) const
    {
        x[move.second] = move.first;
        f.setCurrentDimension(move.second);
        f.bind(move.first);
    }
};
```

```

template<typename INCREMENTAL_FUNCTION, typename SAMPLER>
pair<Vector<double>, double> randomCoordinateDescent(
    INCREMENTAL_FUNCTION& f, SAMPLER const& s, int maxEvals = 1000000)
{
    assert(maxEvals > 0);
    Vector<double> x = localSearch(ContinuousSMove<
        INCREMENTAL_FUNCTION, SAMPLER>(f, s), f.getX(), maxEvals);
    return make_pair(x, f(f.getXi()));
}

template<typename FUNCTION, typename SAMPLER> pair<Vector<double>, double>
RCDGeneral(FUNCTION const &f, SAMPLER const& s,
    Vector<double> x0 = Vector<double>(), int maxEvals = 1000000)
{
    if(x0.getSize() == 0) x0 = s();
    IncrementalWrapper<FUNCTION> iw(f, x0);
    return randomCoordinateDescent(iw, s, maxEvals);
}

```

Для неинкрементных  $f$  используется обертка:

```

template<typename FUNCTION, typename SAMPLER> pair<Vector<double>, double>
RCDGeneral(FUNCTION const &f, SAMPLER const& s,
    Vector<double> x0 = Vector<double>(), int maxEvals = 1000000)
{
    if(x0.getSize() == 0) x0 = s();
    IncrementalWrapper<FUNCTION> iw(f, x0);
    double y = randomCoordinateDescent(iw, s, maxEvals);
    return make_pair(iw.xBound, y);
}

```

Производительность этого алгоритма, измерительных алгоритмов и некоторых других, упомянутых в комментариях, была описана для некоторых  $f$  в работах [24.39 и 24.21]. Во всех случаях предпочтение было отдано неразделимым функциям или тем, у которых есть серьезные проблемы — например, серьезная недифференцируемость. Маловероятно, что какой-либо метод выборки окажется эффективен при превышении некоторого значения  $D$ , например 100, особенно при наличии множества локальных минимумов плохого качества, поэтому в таких случаях тестирование не проводилось. Будьте внимательны, т. к. математическая разделимость (т. е. возможность разложить  $f$  на сумму функций) может быть скрыта, например, монотонным преобразованием  $f$  в другую функцию  $g$ . В этом случае  $g$  также будет решаться по одной переменной за раз.

Другой подход заключается в использовании марковского семплера:

- ◆ ему нужна отправная точка, например независимый 0 (но не используйте эту точку, если она является решением теста);
- ◆ предоставленный пользователем семплер с размером шага должен быть построен из распределения с толстым хвостом, чтобы можно было более эффективно выходить за пределы локальных минимумов. Это дает хороший баланс между исследовательскими задачами и практикой в том смысле, что после множества локальных перемещений можно совершить большой прыжок в другую окрестность. Но, независимо от распределения, величина среднего размера определяет масштаб и, по сути, является параметром дискретизации;

- ♦ он гарантированно сойдется, если в конечном итоге каждая точка может быть сгенерирована.

```
template<typename FUNCTION, typename M_SAMPLER> struct ContinuousMMove
{
    typedef Vector<double> X;
    typedef Vector<double> MOVE;
    FUNCTION const& f;
    M_SAMPLER const& s;
    ContinuousMMove(FUNCTION const& theF, M_SAMPLER const& theS): f(theF),
        s(theS){}
    double getScore(X const& x) const{return f(x);}
    pair<MOVE, double> proposeMove(X const& x) const
    {
        Vector<double> xNext = s(x);
        assert(isfinite(norm(xNext)));
        double yNext = f(xNext), currentScore = getScore(x);
        // если начальная точка равна nan, все шаги допустимы
        return make_pair(xNext, isnan(currentScore) ? 0 : yNext - getScore(x));
    }
    void applyMove(X& x, MOVE const& move) const{x = move;}
};

template<typename FUNCTION, typename M_SAMPLER> pair<Vector<double>,
double> markovianMinimize(FUNCTION const& f, M_SAMPLER const& s,
Vector<double> x0, int maxEvals = 1000000)
{
    assert(maxEvals > 0 && isfinite(norm(x0)));
    Vector<double> x = localSearch(ContinuousMMove<FUNCTION, M_SAMPLER>(f, s),
        x0, maxEvals);
    return make_pair(x, f(x));
}
```

Обсуждаемые алгоритмы охватывают все типы семплеров:

- ♦ случайный поиск по независимым случайным данным использует семплер с независимыми случайными данными;
- ♦ марковский поиск использует семплер смещения;
- ♦ спуск по случайным координатам использует инкрементальный семплер независимых случайных данных.

Все семплеры могут работать с ограничениями «черного ящика» и отбрасывать недопустимые точки (и, возможно, объявлять о провале алгоритма после 1000 или подобного числа отклонений и возвращать текущую точку или код ошибки). Для независимых случайных семплеров отклоненные значения часто не требуются в задачах с ограничениями «черного ящика», потому что можно делать выборки напрямую. Во всех случаях детали алгоритмов инкапсулируются, за исключением, возможно, необходимости обращать внимание на код возврата семплера.

У пошаговых семплеров вместо отклонения можно с помощью бинарного поиска уменьшать размер шага до удовлетворительного, предполагая, что он начинается с удовлетворительной точки в допустимом направлении. Но это приведет к большему количеству неудач в последующих примерах, поэтому неизвестно, какая стратегия

оптимальна, особенно учитывая, что, начиная с угла с высоким  $D$ , найти подходящее направление практически невозможно. Таким образом, для обработки ограничений случайная независимая выборка оказывается лучше.

В случае ограничений координат можно обрезать отдельные координаты предлагаемого  $x$  до границ блока.

Это позволяет избежать неэффективности при обработке общих ограничений:

```
struct AgnosticStepSampler
{
    Vector<double> operator() (Vector<double> const& x) const
    { // должны быть обеспечены конечные выборки по всем компонентам
      assert(isfinite(norm(x)));
      int maxTries = 10; // защита от бесконечных значений
      while(maxTries-->0)
      { // правильно с первой попытки, если рядом нет очень больших чисел
        Vector<double> u = GlobalRNG().randomUnitVector(x.getSize()),
          result = x + u *
            (findDirectionScale(x, u)/10 * GlobalRNG().Levy());
        if(isfinite(norm(result))) return result;
      }
      return x; // невозможно избежать бесконечности
    }
};

Vector<double> boxTrim(Vector<double> x,
  Vector<pair<double, double> > const& box)
{
    for(int i = 0; i < x.getSize(); ++i)
    {
        if(x[i] < box[i].first) x[i] = box[i].first;
        else if(isnan(x[i]) || x[i] > box[i].second) x[i] = box[i].second;
    }
    return x;
}

struct BoxConstrainedStepSampler
{
    Vector<pair<double, double> > box;
    AgnosticStepSampler s;
    BoxConstrainedStepSampler(Vector<pair<double, double> > const& theBox):
        box(theBox) {}
    Vector<double> operator() (Vector<double> const& x) const
    {return boxTrim(s(x), box);}
};
```

Может показаться, что любой алгоритм должен работать только с одним семплером, но тогда не появлялись бы гибридные алгоритмы (о них позже). В худшем случае независимые случайные семплеры ничем не хуже марковских семплеров. Но последние позволяют лучше эксплуатировать различные распространенные на практике отклонения.

С точки зрения НФЛ единственный осмысленный способ теоретического сравнения алгоритмов — это проверить, как они работают с распространенными отклонениями. Это лучше всего выявляется экспериментально, потому что, хотя некоторые предубеж-

дения могут быть явно рассчитаны, неясно, какой подход лучше перед тестированием проблемы. Поэтому при принятии решений на основе тестового набора, чтобы избежать переобучения, решайте как можно больше теоретически. Концептуальный подход к сравнению заключается в том, чтобы учитывать *расточительность*, т. е. алгоритм не должен оценивать одно и то же чаще, чем следует. Для низкой расточительности требуется сбалансировать разведку и эксплуатацию, учитывая предполагаемые отклонения.

Выбирайте такой окончательный алгоритм, типичные результаты которого считаются такими же хорошими, как и у любого другого алгоритма, если их условия с точки зрения везения будут примерно одинаковы. Итак, нам нужен гибрид нескольких алгоритмов. Эта стратегия выглядит хорошей даже при выполнении расчетов с фиксированным бюджетом:

- ◆ гибрид оказывается более устойчивым к сбою определенного алгоритма;
- ◆ предполагая, что используется небольшое количество (не более 5?) алгоритмов и равный бюджет оценок для каждого, сбой даже лучшего из них не должен иметь большого значения.

Если бюджет оценки невелик, например 100 оценок или около того, локальный поиск оказывается слишком затратным, т. к. не может достичь нужной точности. Используйте гибрид RCD и марковского поиска:

```
template<typename FUNCTION, typename SAMPLER, typename M_SAMPLER>
pair<Vector<double>, double> smallBudgetHybridMinimize(FUNCTION const& f,
    SAMPLER const& s, M_SAMPLER const& ms, Vector<double> x0,
    int maxEvals = 100)
{
    assert(maxEvals > 2); // использовать меньшее число не имеет смысла
    return RCDGeneral(f, s, markovianMinimize(f, ms, x0,
        maxEvals * 0.5).first, maxEvals * 0.5);
}
```

В моих тестах этот гибрид показал лучшие результаты, чем случайная выборка и отдельные алгоритмы. Порядок здесь имеет значение, потому что марковский поиск попадет в удачную часть ландшафта, а RCD уточнит результаты.

Имея достаточный бюджет вычислений (т. е. достаточно большой, чтобы аппроксимировать градиенты для локального поиска  $\rightarrow 100D$  или около того), исследователь, скорее всего, опробует на задаче пару многообещающих алгоритмов, чтобы узнать, дает ли какой-либо из них хорошее решение. Таким образом, для очень больших бюджетов оценки имеет смысл использовать больше алгоритмов, что будет рассмотрено далее.

Существует интересная стратегия, составляющая основу некоторых пошаговых мета-эвристик, — *перывистый поиск* (см. [24.6.]). Эта стратегия похожа на то, как человек ищет потерянные ключи на пляже:

- ◆ если у вас имеется пространственная память, вы используете некоторый шаблон, предполагая, что вы можете сканировать небольшие квадратные участки за раз;
- ◆ если памяти нет, то можно использовать прогулку по Леви, т. е. брать прыжки Леви между сканированиями участков и не сканировать во время прыжков. В отличие от марковского поиска, можно начать с последнего испробованного решения, а не с текущего лучшего. В некоторых случаях эта стратегия оказывается оптимальной.

На этой стратегии построена имитация отжига (см. главу 16. Комбинаторная оптимизация). В ее простой реализации для перемещения используется прыжок Леви. Производительность оказывается хорошей при наличии ограничения координат, но без них падает из-за выхода за границы, по крайней мере, в марковской версии. Марковский поиск способен обходить «препятствия», потенциально делая процесс отжига ненужным, в отличие от дискретных окрестностей комбинаторной оптимизации. Кроме того, симуляция отжига хорошо работает при прыжках через небольшие препятствия, но не через большие (см. работу [24.26]). В ней утверждается, что в этом случае более эффективен случайный перезапуск с последующим локальным поиском в случае успеха. Но этот алгоритм кажется лишь скромным улучшением случайного перезапуска, который подвержен проклятию размерности. В работе [24.51] авторы утверждают, что имитация отжига не столь эффективна, как некоторые методы случайного поиска, особенно те из них, в которых для удаления бесперспективных областей используется память и статистические рассуждения, но при этом они не говорят, какие именно алгоритмы лучше. То же самое касается и генетических алгоритмов (о них позже). Но методы, эффективно использующие память, намного более затратны в вычислительном отношении и обычно основаны на метамоделях (см. раздел комментариев).

Версия имитации отжига на основе RCD, поскольку локальный поиск работает лучше, но все же не так хорошо, как алгоритмы на основе популяции, приведенные далее, исключена из гибридного подхода. Но зато получаем лучший алгоритм для инкрементных  $f$ :

```
template<typename INCREMENTAL_FUNCTION, typename SAMPLER> pair<Vector<double>,
double> simulatedAnnealingSMinimizeIncremental(INCREMENTAL_FUNCTION&
f, SAMPLER const& s, Vector<double> x0, int maxEvals = 1000000)
{
    assert(maxEvals > 0 && isfinite(norm(x0)));
    Vector<double> x = selfTunedSimulatedAnnealing(ContinuousSMove<
        INCREMENTAL_FUNCTION, SAMPLER>(f, s), x0, maxEvals);
    return make_pair(x, f(f.getXi()));
}

template<typename INCREMENTAL_FUNCTION, typename SAMPLER> double
incrementalSABeforeLocalMinimize(INCREMENTAL_FUNCTION &f, SAMPLER const& s,
int maxEvals = 1000000, double xPrecision = highPrecEps)
{
    simulatedAnnealingSMinimizeIncremental(f, s, f.getX(), maxEvals * 0.9);
    // f запоминает лимит оценки
    return unimodalCoordinateDescent(f, maxEvals, xPrecision);
}
```

Еще один распространенный набор алгоритмов основан на популяции. В гибридной реализации такой алгоритм может быть первым в последовательности. Например, можно использовать *дифференциальную эволюцию*, упомянутую в работе [24.30] и обычно реализуемую в различных библиотеках. Она также была ключевым алгоритмом для нескольких задач SIAM (см. [24.8]). Основная идея алгоритма состоит в том, чтобы иметь совокупность решений и выполнять локальные перемещения для каждого члена независимо, используя шаг блочной координаты по направлению к специально сгенерированной выборке. Если известен текущий член популяции  $x_i$ :

1. Создать  $x_{\text{new}} = x_j + F(x_k - x_l)$ , где  $j, k, l$  — случайные различные индексы.
2. Выбрать случайный размер  $k_{\text{rand}}$  и установить  $x_{\text{next}} = x_i$ .

3. Для любой размерности  $k$  установить  $x_{\text{next}}[k] = x_{\text{new}}[k]$ , если  $k = k_{\text{rand}}$  или Bernoulli( $C_r$ ) истинно.

Логические значения:  $F = 1$  и скорость скрещивания  $C_r = 0,5$ . Но чаще используется значение  $F = 0,9$ , потому что  $F = 1$  приводит к меньшему разнообразию из-за уменьшения количества различных случайных выборов  $j$ ,  $k$  и  $l$  (подробности — в работе [24.37]). Размер популяции был выбран равным  $\text{maxEvals}^{1/3}$  (то же самое и для других алгоритмов популяции, обсуждаемых в комментариях). Это необходимо, чтобы размер совокупности  $\rightarrow \infty$ , поскольку  $\text{maxEvals} \rightarrow \infty$ , чтобы гарантировать сходимость, потому что начальная совокупность является случайной на основе независимой случайной выборки:

```
template<typename FUNCTION, typename SAMPLER> pair<Vector<double>, double>
differentialEvolutionMinimize(FUNCTION const& f, SAMPLER const& s,
Vector<pair<double, double> > const& box, int maxEvals = 1000000)
{
    assert(maxEvals > 0);
    int n = pow(maxEvals, 1.0/3);
    Vector<pair<Vector<double>, double> > population(n);
    for(int i = 0; i < n; ++i)
    {
        population[i].first = s();
        population[i].second = f(population[i].first);
    }
    maxEvals -= n;
    while(maxEvals > 0)
    {
        for(int i = 0; i < n && maxEvals-- > 0; ++i)
        { // мутация новой точки
            Vector<int> jkl = GlobalRNG().randomCombination(3, n);
            Vector<double> xiNew = population[i].first, xiMutated =
                boxTrim(population[jkl[0]].first + (population[jkl[1]].first -
                    population[jkl[2]].first) * 0.9, box);
            // скрещивание с мутировавшей точкой
            int D = xiNew.getSize(), randK = GlobalRNG().mod(D);
            for(int k = 0; k < D; ++k) if(GlobalRNG().mod(2) || k == randK)
                xiNew[k] = xiMutated[k];
            if(!isfinite(norm(xiNew)))
            { // обеспечение выполнения конечных выборов
                ++maxEvals;
                continue;
            }
            // выбираем лучшее из оригинала и мутации
            double yiNew = f(xiNew);
            if(yiNew < population[i].second)
            {
                population[i].first = xiNew;
                population[i].second = yiNew;
            }
        }
    }
}
```



```

pair<Vector<double>, double>& best = population[0];
for(int i = 1; i < n; ++i)
    if(best.second < population[i].second) best = population[i];
return best;
}

```

Теоретически алгоритм может не сработать, если популяция сократится до одной точки. Интересное наблюдение заключается в том, что если генерация выборки DE заменена независимым случайным семплером, то алгоритм превращается в параллельный алгоритм RCD, но с многоходовыми операциями, размер которых определяется биномиальной ( $C$ ) выборкой. Таким образом, семплирование получается качественным. Генерация DE проста, не обременена лишней логикой, часто реализуется и дает хорошую производительность. Механизм перемещения также поддерживает масштабирование автоматически.

Другой компонент гибрида — *генетический локальный поиск* (см. главу 16. Комбинаторная оптимизация). Моя реализация основана на:

- ◆ равномерном скрещивании;
- ◆ использовании RCD в качестве локального поиска.

```

template<typename FUNCTION, typename SAMPLER> class GAContinuousProblem
{
    FUNCTION const& f;
    SAMPLER const& s;
public:
    typedef Vector<double> X;
    GAContinuousProblem(FUNCTION const& theF, SAMPLER const& theS): f(theF), s(theS){}
    X generate()const{return s();}
    void crossover(X& x1, X& x2)const
    { // равномерное скрещивание
        assert(x1.getSize() == x2.getSize());
        for(int k = 0; k < x1.getSize(); ++k) if(GlobalRNG().mod(2))
            swap(x1[k], x2[k]);
    }
    X localSearch(X x, int nLocalMoves)const // RBCD для того же семплера
    {return RCDGeneral(f, s, x, nLocalMoves).first;}
    double evaluate(X const& x)const{return f(x);}
};

template<typename FUNCTION, typename SAMPLER> pair<Vector<double>, double>
geneticLocalSearchContinuous(FUNCTION const& f, SAMPLER const& s,
int maxEvals = 1000000)
{
    int nLocalMoves = int(pow(maxEvals, 1.0/3)), populationSize = nLocalMoves;
    return geneticLocalSearch(GAContinuousProblem<FUNCTION, SAMPLER>(f, s),
        populationSize, nLocalMoves, maxEvals);
}

```

Хороший гибридный подход выглядит следующим образом:

1. Дифференциальная эволюция для 30% оценок.
2. Генетический локальный поиск для 30% оценок.

3. Марковский поиск по 30% оценок лучшего из указанных.
4. Локальный поиск по оставшимся 10% оценок.

Значения процентов получены из идеи о том, что нужно отдать  $\approx 10\%$  на конечный локальный поиск, а остальное разделить поровну. Такой порядок, кажется, хорошо работает в этой дифференциальной эволюции, GLS дает хорошее начальное решение для марковского поиска для дальнейшего изучения, а сглаженный локальный поиск полирует результат, если это возможно:

```
template<typename FUNCTION, typename SAMPLER, typename M_SAMPLER>
pair<Vector<double>, double> hybridBeforeLocalMinimize(
    FUNCTION const& f, SAMPLER const& s, M_SAMPLER const& ms,
    Vector<pair<double, double> > const& box, int maxEvals = 1000000,
    double yPrecision = highPrecEps)
{
    assert(maxEvals > 1000); // нет смысла использовать меньшее значение
    pair<Vector<double>, double> glsSolution = geneticLocalSearchContinuous(f,
        s, maxEvals * 0.3), deSolution = differentialEvolutionMinimize(f, s,
        box, maxEvals * 0.3);
    return hybridLocalMinimize(markovianMinimize(f, ms, (deSolution.second <
        glsSolution.second ? deSolution : glsSolution).first, maxEvals * 0.3
        ).first, f, maxEvals * 0.1);
}
```

Этот гибрид кажется практичным выбором для больших бюджетов оценки, поскольку бессмысленно испытывать больше алгоритмов ради улучшения результата. Конкретный выбор не обязательно оптимален, но зато стабилен из-за простоты его составляющих и различий их стратегий решения. В каждом из них используется своя стратегия движения. Трудно придумать дополнительные алгоритмы, которые могут быть полезны в качестве компонентов. Например, можно заменить GSL имитацией отжига на основе пробоотборника компонентов, но, судя по приведенным далее сравнениям (рис. 24.6 и 24.7), первый вариант работает немного лучше, поэтому такая замена не делалась. Кроме того, алгоритмы первого этапа вообще не используют марковскую выборку, поэтому необходим только один такой метод. Алгоритм RCD не используется, потому что в GLS в качестве локального поиска задействуются координатные перемещения.

В случае зашумленных  $f$  обсуждаемые методы должны работать с небольшими уровнями шума, но выводы относительно их общей работоспособности могут оказаться неприменимы.

Из-за НФЛ любые сравнения производительности в лучшем случае сводятся к типичному практическому поведению с множеством репрезентативных тестов  $f$ . Но статистические выводы верны только в том случае, если они выбраны случайным образом. Такого никогда не бывает, поэтому в лучшем случае можно провести наблюдательное исследование (см. главу 21. *Вычислительная статистика*). В таком исследовании будет задействовано некоторое количество простых, средних и сложных задач, которые репрезентативны на практике. Различные предубеждения, такие как стремление решений быть ближе к началу системы координат и способность разделять переменные, также будут устранены из большинства задач. И даже в этом случае хорошее всестороннее сравнение вряд ли будет репрезентативным для конкретной области. Некоторые источники рекомендуют чередовать области задач, чтобы сделать сравнение раздели-

мых функций более справедливым. Но в работе [24.12] отмечается, что делать так зачастую бывает глупо, потому что в реальных задачах переменные имеют четкие значения и такое смешивание не имеет смысла. Также многие алгоритмы преднамеренно или непреднамеренно имеют отклонения, которые помогают решать задачи, в которых

Функция	RandomBLM	RCDBLM	IncrementalSABLM	MarkovianBLM	HybridBLM	Chol11CMA_ESBLM	SimulatedAnnealingBLM	SimulatedAnnealingSB LM	DifferentialEvolutionBLM	GeneticSBLM
Bukin6	-1,77	-1,68	-0,54	-1,63	-1,81	-1,79	-1,79	-1,71	-1,85	-1,72
Damavandi	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65
Easom	-15,65	-8,87	-4,16	-15,65	-15,65	-2,09	-15,65	-7,83	-15,65	-15,65
GulfrND	-12,72	-12,90	-5,84	-12,93	-15,56	-13,84	-12,75	-12,92	-15,65	-12,79
Price2	-15,65	-15,65	-7,11	-15,65	-15,65	-1,98	-15,65	-15,65	-15,65	-15,65
Trefethen	-5,08	-6,93	-3,03	-15,27	-15,03	-0,46	-14,41	-9,36	-15,03	-13,57
Ackley2	-15,59	-15,55	-2,75	-15,49	-15,65	-4,17	-15,52	-15,51	-15,65	-15,65
FletcherPowell2	-15,65	-15,60	-2,07	-15,64	-15,65	-15,64	-15,63	-15,64	-15,65	-15,64
Griewank2	-4,64	-7,09	-2,96	-15,65	-15,65	-1,38	-15,65	-9,34	-15,65	-10,70
Rastrigin2	-15,65	-15,65	-5,42	-15,65	-15,65	0,00	-15,65	-15,65	-15,65	-15,65
SchwefelDoubleSum2	-15,65	-15,65	-7,91	-15,63	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65
SchwefelMax2	-15,61	-15,26	-2,33	-15,50	-15,65	-15,65	-15,59	-15,60	-15,65	-15,48
StepFunction2	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65
Weierstrass2	-13,63	-14,41	-2,28	-15,65	-15,65	-13,95	-15,65	-13,29	-15,65	-13,35
Trig2	-15,65	-15,65	-8,07	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65
Pinter2	-15,65	-10,49	-4,85	-15,65	-15,65	-3,19	-15,65	-14,62	-15,65	-15,65
Salomon2	-4,42	-1,90	-1,70	-15,26	-15,65	-1,40	-15,03	-3,75	-15,65	-12,18
SchaeferF62	-8,83	-1,87	-3,44	-15,65	-15,65	-1,39	-15,65	-9,74	-15,20	-12,47
Ackley6	-5,44	-14,85	-2,13	-12,44	-14,89	-2,05	-9,90	-15,02	-1,04	-14,92
StepFunction6	0,00	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65
Weierstrass6	-2,31	-7,83	-1,55	-15,65	-15,65	-5,71	-15,65	-8,22	-15,65	-6,57
Trig6	-7,92	-7,17	-4,20	-15,65	-15,65	-14,04	-15,65	-9,93	-15,65	-15,05
Pinter6	-4,09	-6,27	-4,33	-7,33	-15,65	0,00	-14,59	-11,56	-15,65	-15,64
Salomon6	-2,65	-0,51	-0,70	-1,00	-1,00	-1,58	-1,00	-0,68	-1,00	-0,99
SchaeferF66	-0,14	-2,53	-2,80	-0,18	-1,30	0,00	-0,51	-7,81	-0,97	-1,32
Ackley10	0,00	-14,65	-1,76	0,00	-14,69	0,00	-9,45	-14,55	-12,34	-14,60
FletcherPowell10	-9,07	-7,60	0,00	-9,52	-12,02	-10,73	-11,43	-10,12	-14,27	-11,97
Griewank10	-3,40	-2,72	-1,19	-0,40	-15,65	-0,73	-0,96	-1,28	-15,65	-2,55
Rastrigin10	0,00	-15,65	-3,27	0,00	-15,65	0,00	0,00	-15,65	-15,65	-15,65
SchwefelDoubleSum10	-14,40	-15,03	-1,39	-15,29	-14,81	-15,65	-15,17	-14,95	-15,65	-14,94
SchwefelMax10	-15,65	-15,65	-1,14	-15,59	-15,65	-15,40	-15,65	-15,65	-15,65	-15,65
StepFunction10	0,00	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65
Weierstrass10	-0,18	-3,19	-1,14	-15,65	-15,65	-5,77	-15,65	-2,35	-15,65	-2,59
Trig10	-7,46	-8,05	-4,22	-15,65	-15,65	-11,72	-15,65	-6,16	-15,65	-6,43
Pinter10	0,00	-1,96	-3,52	0,00	-15,64	0,00	-0,97	-11,76	-15,65	-14,57
Salomon10	-0,19	-0,29	-0,42	-1,00	-1,00	-0,57	-0,93	-0,45	-1,00	-0,76
StepFunction10	0,00	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65
Weierstrass10	-0,18	-3,19	-1,14	-15,65	-15,65	-5,77	-15,65	-2,35	-15,65	-2,59
Trig10	-7,46	-8,05	-4,22	-15,65	-15,65	-11,72	-15,65	-6,16	-15,65	-6,43
Pinter10	0,00	-1,96	-3,52	0,00	-15,64	0,00	-0,97	-11,76	-15,65	-14,57
Salomon10	-0,19	-0,29	-0,42	-1,00	-1,00	-0,57	-0,93	-0,45	-1,00	-0,76
SchaeferF610	0,00	-0,95	-1,59	0,00	-1,05	0,00	0,00	-2,49	-0,20	-1,06
Средние ранги	5,81	4,98	7,76	3,95	1,57	6,12	3,76	4,76	1,71	3,57

**Рис. 24.6.** Некоторые методы решения ряда задач. Использовались семплы с ящичковыми ограничениями координат, а исходным решением была случайная выборка из ограничивающей области. В каждом случае использовалось 30 повторений. Всем алгоритмам давался бюджет около 1 млн оценок, и большинство израсходовало его почти полностью

Функция	RandomBLM	RCDBLM	IncrementalSABLM	MarkovianBLM	HybridBLM	Chol11CMA_ESBLM	SimulatedAnnealingBLM	SimulatedAnnealingSBLM	DifferentialEvolutionBLM	GeneticLSBLM
Bukin6	-1,38	-1,36	-0,49	-1,72	-1,25	-1,88	0,00	-1,39	-1,18	-1,16
Damavandi	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65
Easom	-15,65	-7,31	-4,21	-2,09	-15,65	-1,57	0,00	-6,78	-15,65	-15,65
GulfFRND	-12,91	-13,02	-4,92	-6,87	-12,84	-13,99	-2,85	-13,02	-12,81	-13,09
Price2	-15,65	-15,17	-6,68	-12,23	-15,65	-1,00	-1,00	-15,65	-1,00	-15,65
Trefethen	-4,92	-8,83	-2,55	-15,31	-15,07	-0,36	-1,31	-8,31	-15,05	-13,05
Ackley2	-15,49	-15,55	-2,71	-15,10	-15,65	-4,17	0,00	-15,46	-15,65	-15,52
FletcherPowell2	-14,85	-9,42	-3,33	-14,78	-11,49	-13,04	-1,18	-14,50	-14,35	-15,24
Griewank2	-6,61	-8,41	-3,18	-15,65	-15,65	-0,71	-3,14	-11,15	-15,65	-13,85
Rastrigin2	-15,65	-15,65	-3,98	-15,65	-15,65	-0,52	-4,18	-15,65	-15,65	-15,65
SchwefelDoubleSum2	-15,65	-15,63	-9,30	-15,58	-15,65	-15,65	-15,65	-15,65	-15,65	-15,65
SchwefelMax2	-15,65	-15,32	-2,56	-15,51	-15,65	-15,65	-15,62	-15,60	-15,65	-15,45
StepFunction2	-15,13	-15,65	-15,65	-15,65	-15,65	-15,65	0,00	-15,65	-15,65	-15,65
Weierstrass2	-12,42	-13,77	-2,27	-11,74	-15,58	-15,41	-0,95	-12,92	-15,65	-13,00
Trig2	-15,48	-15,42	-7,62	-15,44	-15,38	-15,65	-4,28	-15,41	-15,65	-15,20
Pinter2	-15,65	-12,55	-4,20	-15,65	-15,65	-5,77	-15,14	-14,62	-15,65	-15,65
Salomon2	-4,91	-2,78	-1,81	-15,47	-15,65	-0,45	-7,48	-5,81	-15,65	-13,05
SchaeferF62	-2,92	-2,18	-2,42	-15,14	-15,65	-0,90	-0,30	-6,10	-15,65	-8,83
Ackley6	-0,49	-14,92	-1,68	-10,36	-14,85	-0,49	-0,52	-14,95	0,00	-14,89
FletcherPowell6	-12,23	-12,22	-0,24	-12,15	-14,00	-9,91	0,00	-12,71	-12,57	-14,16
Griewank6	-0,26	-1,62	-1,43	-0,74	-15,65	-0,46	-0,32	-2,51	-15,65	-1,71
Rastrigin6	0,00	-15,65	-2,70	0,00	-15,65	0,00	0,00	-15,65	-15,65	-15,65
SchwefelDoubleSum6	-15,65	-15,65	-2,75	-15,65	-15,65	-15,65	-15,58	-15,65	-15,65	-15,64
SchwefelMax6	-15,59	-15,65	-0,65	-15,65	-15,65	-15,65	-15,62	-15,57	-15,65	-15,65
StepFunction6	0,00	-15,65	-15,65	-15,65	-15,65	-15,65	0,00	-15,65	-15,65	-15,65
Weierstrass6	-0,92	-5,03	-1,49	-3,55	-3,32	-3,77	-0,52	-4,34	-2,30	-4,96
Trig6	-9,89	-8,52	-4,79	-9,83	-12,56	-14,44	-0,45	-10,08	-9,47	-14,77
Pinter6	-0,02	-4,75	-1,47	-7,33	-15,65	0,00	0,00	-4,89	-15,65	-15,14
Salomon6	-1,14	-0,49	-0,64	-1,00	-1,00	-0,69	-0,68	-0,59	-1,00	-0,90
SchaeferF66	-0,03	-2,55	-1,97	-0,09	-2,28	0,00	-0,52	-4,99	-0,02	-1,31
Ackley10	0,00	-13,25	-1,41	0,00	-14,65	0,00	0,00	-14,70	0,00	-14,20
FletcherPowell10	-3,68	-3,67	0,00	-3,82	-0,46	-5,32	0,00	-3,20	-2,29	-1,37
Griewank10	-2,88	-2,71	-1,00	-0,41	-15,65	-0,73	-0,45	-1,17	-15,65	-2,46
Rastrigin10	0,00	-15,65	-1,69	0,00	-15,65	0,00	0,00	-15,65	-15,65	-15,65
SchwefelDoubleSum10	-14,72	-14,65	-0,23	-15,24	-15,15	-15,65	-14,79	-14,66	-15,65	-14,92
SchwefelMax10	-15,64	-15,50	-0,30	-15,65	-15,48	-15,58	-15,65	-15,65	-15,65	-15,65
StepFunction10	0,00	-15,65	-15,65	-15,65	-15,65	-15,65	0,00	-15,65	-15,65	-15,65
Weierstrass10	-0,08	-2,52	-1,13	-0,32	-1,93	-5,29	0,00	-1,96	-1,07	-2,62
Trig10	-8,25	-9,45	-4,46	-6,53	-10,27	-9,66	0,00	-8,11	-9,07	-9,13
Pinter10	0,00	-3,40	-1,34	0,00	-15,65	0,00	0,00	-8,80	-15,65	-14,58
Salomon10	-1,78	-0,31	-0,44	-1,00	-1,00	-0,62	-0,33	-0,40	-1,00	-0,81
SchaeferF610	0,00	-1,38	-1,17	0,00	-1,00	0,00	0,00	-2,04	0,00	-1,03
Средние ранги	5,19	4,55	7,24	4,48	2,60	5,17	7,93	3,88	3,02	3,21

Рис. 24.7. Тот же эксперимент, но без известных ограничений координат.

Производительность сейчас намного хуже, и многие методы не смогли улучшить результат из-за неудачного почти бесконечного начального начального числа.

Так что сравнение производительности в этом случае не слишком значимо

тоже есть отклонения. В работах [24.12 и 24.13] приведено более подробное обсуждение многих из этих вопросов. Чтобы выбрать хорошие алгоритмы, требуется много внешней логики. Например, можно выбирать простые алгоритмы, хорошо зарекомендовавшие себя, не намного хуже лучших и т. д. Представленные здесь гибридные алгоритмы, по всей видимости, тоже являются разумным выбором.

## 24.10. Оптимизация в дискретном множестве

Вы можете сгенерировать дискретный набор даже из непрерывного диапазона. Это работает, в частности, для экспоненциального диапазона  $2^i$ , где  $i \in$  некоторому дискретному интервалу, когда все решения в пределах малого мультипликативного множителя безразличны, т. е. преобразованное  $f$  является липшицевым.

*Поиск по сетке* — самый простой метод, который выполняет дискретизацию и пробует все возможности. Этот метод невозможно применить на больших  $D$ , но поскольку для липшицевой функции  $f$  находится решение, близкое к оптимальному, метод очень полезен при малых  $D$ , особенно для оптимизации параметров. Основная задача — сгенерировать все выборки значений. Идея решения состоит в том, чтобы упорядочить переменные, сохранить текущие индексы любых переменных и:

- ◆ после продвижения любой переменной перейти к следующей. Если не осталось переменных, которые можно перемещать, алгоритм завершается;
- ◆ когда текущий цикл переменных заканчивается, нужно сбросить все старшие переменные. Когда цикл переменной-0 заканчивается, генерация проходит через все выборки.

```
template<typename FUNCTION> Vector<double> gridMinimize(
    Vector<Vector<double> > const& sets, FUNCTION const& f = FUNCTION())
{
    assert(sets.getSize() > 0);
    Vector<double> best;
    for(int i = 0; i < sets.getSize(); ++i)
    {
        assert(sets[i].getSize() > 0);
        best.append(sets[i][0]);
    }
    double bestScore = f(best);
    Vector<int> current(sets.getSize(), -1);
    current.lastItem() = 0;
    for(int level = 0; level > -1;)
    {
        if(level < sets.getSize())
        {
            if(++current[level] < sets[level].getSize()) ++level;
            else current[level--] = -1;
        }
        else
        { // выбор значения для процесса
            Vector<double> values;
            for(int i = 0; i < sets.getSize(); ++i)
                values.append(sets[i][current[i]]);
            double score = f(values);
            if(score < bestScore)
            {
                bestScore = score;
                best = values;
            }
        }
    }
}
```

```

        --level;
    }
}
return best;
}

```

Другая стратегия чем-то похожа на поиск по компасу. Стратегия стремится по возможности уменьшить количество оценок, но не гарантирует оптимальности:

1. Для любой переменной предположим, что значения отсортированы.
2. Любая переменная инициализирует последнее успешное направление до + 1.
3. Пока алгоритм не сойдется или не будет достигнут предел итераций:
4.     Выбрать переменную.
5.     Попробовать двинуться в последнем успешном направлении.
6.     Если не получилось, попробовать двинуться в обратном направлении.

```

/* предполагается, что установленные значения отсортированы
(или отсортированы в обратном порядке)! */
template<typename FUNCTION> pair<Vector<double>, pair<double, int> >
compassDiscreteMinimizeHelper(Vector<Vector<double> > const& sets,
    Vector<int> current, FUNCTION const& f = FUNCTION(),
    int remainingEvals = 100)
{ // начнем с медианы
    Vector<double> best;
    for(int i = 0; i < sets.getSize(); ++i)
    {
        assert(0 <= current[i] && current[i] < sets[i].getSize());
        best.append(sets[i][current.lastItem()]);
    }
    double bestScore = f(best);
    Vector<int> preferredSign(sets.getSize(), 1);
    for(bool done = false; !done;)
    {
        done = true;
        for(int i = 0; i < sets.getSize(); ++i)
            for(int j = 0; j < 2; ++j)
            {
                int sign = preferredSign[i];
                if(j == 1) sign = -sign;
                int next = current[i] + sign;
                if(0 <= next && next < sets[i].getSize())
                {
                    if(remainingEvals-- < 1)
                        return make_pair(best, make_pair(bestScore, 0));
                    best[i] = sets[i][next];
                    double score = f(best);
                    if(score < bestScore)
                    {
                        current[i] = next;
                        bestScore = score;
                    }
                }
            }
    }
}

```

```

        done = false;
        preferredSign[i] = sign;
        j = 2;
    }
    else best[i] = sets[i][current[i]];
}
}
}
return make_pair(best, make_pair(bestScore, remainingEvals));
}

```

Удобно начинать алгоритм с медианы каждого набора переменных, что разумно, если предположить, что диапазоны выбраны правильно:

```

template<typename FUNCTION> Vector<double> compassDiscreteMinimize(
    Vector<Vector<double> > const& sets, FUNCTION const& f = FUNCTION(),
    int remainingEvals = 100)
{
    Vector<int> current;
    for(int i = 0; i < sets.getSize(); ++i)
    {
        assert(sets[i].getSize() > 0);
        current.append(sets[i].getSize()/2);
    }
    return compassDiscreteMinimizeHelper(sets, current, f, remainingEvals).first;
}

```

Алгоритм может застрять в локальном минимуме, поэтому можно адаптировать стратегии глобализации. В дальнейшем мы обсуждать это не будем, потому что в большинстве случаев эту проблему можно решить путем отображения на непрерывную задачу. В моих экспериментах алгоритм превосходит поиск по сетке для оптимизации параметров SVM (см. главу 26. *Машинное обучение: классификация*) с точки зрения количества оценок, но дает такое же качество обучения.

## 24.11. Стохастическая оптимизация

В случае зашумленных  $f$  нужно минимизировать  $E[f(x)]$ . Ожидание — это не единственный способ указать качества желаемого минимума. Например, часто минимизируют  $x_2$  с чуть большим  $E[f(x_2)]$ , но с гораздо меньшей дисперсией, но такие формулировки приводят к гораздо более сложной многокритериальной оптимизации.

Стохастическую задачу можно преобразовать в детерминированную с низким уровнем шума, используя *аппроксимацию выборочного среднего* (Sample Average Approximation, SAA), т. е. минимизировать  $g(x)$  = среднее значение оценки  $f(x)$   $n$  раз для некоторого  $n$ . При  $n \rightarrow \infty$ ,  $g(x) \rightarrow E[f(x)]$ . Это работает с дискретной, ограниченной и глобальной минимизацией. Но непонятно, как выбрать значение  $n$ .

*Аппроксимация пути выборки* (Sample Path Approximation, SPA) сначала решает задачу с малым  $n$ , а затем с удвоенным  $n$ , используя решение предыдущего раунда в качестве начального решения следующего. Таким образом, алгоритм, решающий детерминированную проблему, должен быстро сходиться в более поздних раундах, хотя для каждой

оценки используется больше симуляций. *Схождением* назовем ситуацию, когда средние значения и найденные решения перестают изменяться.

Поскольку функция  $g$  имеет небольшие скачкообразные разрывы, многие алгоритмы детерминированной оптимизации нельзя применять как есть. Например, конечно-разностная производная не подходит для выбора значения  $h$  по умолчанию. Алгоритм Нелдера — Мида, поиск по компасу и координатный спуск работают как есть. При достаточно большом начальном шаге, превышающем шум, они, скорее всего, найдут решение, настолько близкое к оптимуму, насколько позволяет шум.

Хотя это основной вариант использования алгоритма Нелдера — Мида, расточительно оценивать  $f$  много раз при одном и том же  $x$  для уменьшения шума. Более эффективные алгоритмы оптимизируют и уменьшают шум одновременно, избегая неверных решений и не требуя их точных значений.

## 24.12. Алгоритмы стохастической аппроксимации

Алгоритм Роббинса — Манро (RM) решает систему стохастических уравнений  $f(x) = 0$  с учетом наблюдений  $m(x)$  от  $f(x)$ , таких что  $E[m(x)] = f(x)$ :

1. Выберите начальный  $x_0$ .
2. Для некоторых больших  $n$  и размера шага  $s_i$ , пока  $i < n$ :
3.  $x_{i+1} = x_i - s_i m(x_i)$ .

Для схождения к корню должны выполняться некоторые условия:  $\sum_{0 \leq i \leq \infty} s_i = \infty$  и  $\sum_{0 \leq i \leq \infty} s_i^2 < \infty$ . Условия гладкости являются статистическими или основанными на ОДУ, но их невозможно проверить на практике. См. обзор [24.41].

При еще большем количестве условий  $s_i = \frac{1}{i+1}$  асимптотически оптимальна и приводит

к сходимости  $O(1/\sqrt{n})$  (см. [24.41]). Поскольку любой алгоритм, даже если он начинается с корня  $x$ , должен как минимум проверить, что  $f(x) = 0$ , используя метод Монте-Карло, сходимость которого также равна  $O(1/\sqrt{n})$ , RM асимптотически оптимален. Но найти корень важнее, чем оценить его значение, а более медленно убывающие

$s_i = \frac{1}{(i+1)^{0.501}}$  обычно позволяют ускорить поиск, несмотря на худшую асимптоти-

ческую сходимость (см. [24.41]). В некотором смысле асимптотическая сходимость имеет значение только для полировки точности, потому что она срабатывает после того, как точность уже достигнута. Часто используется константное значение  $s_i$ , но оно не удовлетворяет условиям. Алгоритм RM очень чувствителен к  $s_0$ , потому что слишком большие значения расходятся, а слишком малые сходятся слишком медленно. Что касается SPSSA (обсуждается позже), можно использовать поиск по сетке, чтобы найти подходящее значение.

В *стохастическом градиентном спуске* (Stochastic Gradient Descent, SGD) используется алгоритм RM для решения  $\nabla f = 0$ . Не зная  $f$ , можно придумать несмещенную оценку  $\nabla f$



при достаточном знании  $m$ . Пусть  $\varepsilon_i$  исходит из распределения вероятностей с функцией плотности вероятности  $p$ . Тогда

$$\nabla f(x) = \frac{\partial}{\partial x} \int_{-\infty \leq t \leq \infty} m(x, \varepsilon(t, x)) dt.$$

Если  $\varepsilon$  не зависит от  $x$  и можно поменять местами производную и интеграл, формула упрощается до

$$\nabla f(x) = E \left[ \frac{\partial m(x, \varepsilon)}{\partial x} \right].$$

Во многих случаях можно вычислить  $\frac{\partial m(x, \varepsilon)}{\partial x}$  аналитически. SGD также работает с негладкими  $f$ , используя субградиенты.

Например, если есть набор точек  $(x_i, y_i)$ , линейная регрессия вычисляет гиперплоскость  $y(x) = wx$  такую, что  $\sum (y(x_i) - y_i)^2$  минимальна. Когда точки поступают в реальном времени,  $m(w, \varepsilon_i) = (y(x_i) - y_i)^2$  является несмещенной оценкой истинной ошибки в точке  $i$  и  $\frac{\partial m(w, \varepsilon_i)}{\partial w} = x_i (y(x_i) - y_i)^2$ . Таким образом, итерация RM представляет собой  $w^- = s_i x_i (y(x_i) - y_i)^2$ .

Хотя RM легко сходится в задаче нахождения корней, SGD может быстро уйти в  $\infty$ , если  $s_i$  слишком велико, потому что уменьшение  $f$  не проверяется (рис. 24.8).

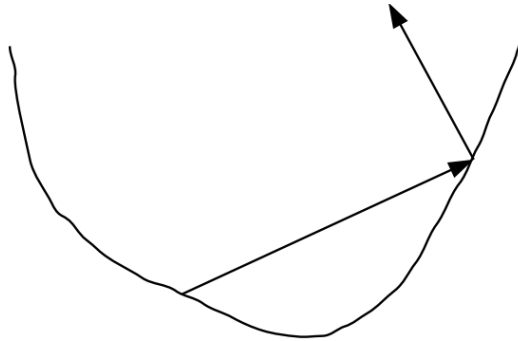


Рис. 24.8. Алгоритм расходится, если  $s_i$  слишком велико

Если аналитической несмещенной оценки градиента нет, алгоритм стохастической аппроксимации возмущения (Stochastic Perturbation Stochastic Approximation, SPSA) оценивает ее численно:

1. Выбрать начальную точку  $x_0$ , размер шага градиента  $h_i$  и векторы смещения  $v_i$ .
2. Пока  $i < n$ :
3. Для  $0 < j < D$ :

$$4. \quad x_{i+1}[j] = x_i[j] - s_i \frac{m(x_i + h_i v_i) - m(x_i - h_i v_i)}{2h_i v_i[j]}.$$

Обычно  $v_i$  являются векторами случайных величин Бернулли, которые равны 1 или  $-1$  с вероятностью  $1/2$ . Возможны и другие варианты, но они сложнее (см. [24.41]). Для сходимости к локальному минимуму, т. е. к корню  $\nabla f$ , нужно выполнение условий

гладкости, а также  $\sum_{0 \leq i \leq \infty} s_i = \infty$  и  $\sum_{0 \leq i \leq \infty} \left(\frac{s_i}{h_i}\right)^2 < \infty$ . Опять же, условия гладкости являются статистическими или основаны на ОДУ, но их невозможно проверить на практике.

Асимптотически оптимальные варианты:  $s_i = \frac{1}{i+1}$  и  $h_i = \frac{1}{i+1^{1/6}}$ . Эти значения дают сходимость  $O(n^{-1/3})$  (см. [24.41]), что хуже, чем у SGD. Как и для SGD, более медленное

уменьшение  $s_i = \frac{1}{i+1^{0.602}}$  и  $h_i = \frac{1}{i+1^{0.101}}$  обычно ведет к ускорению поиска. Смещение градиента SPSA составляет  $O(h_i^2)$ , как и у прямой оценки разности, но требуется 2, а не  $D$  оценок  $m$  (см. [24.41]).

SPSA очень чувствителен к  $s_i$  и  $h_i$ , особенно к начальным значениям. Поскольку алгоритм не гарантирует спуск, при быстром изменении  $f$  несколько неверных шагов могут отправить  $x$  далеко в  $\infty$ , если  $s_0$  слишком велико, или  $x$  будет медленно сползать к оптимуму, если  $s$  слишком мало. Желательно получить самое большое значение  $s_0$ , которое не приведет к очень далекому прыжку. Еще одна проблема заключается в том, что  $h_i$  практически не меняется.

Практический подход состоит в том, чтобы использовать один и тот же начальный шаг для  $s_i$  и  $h_i$ , чтобы сократить количество параметров и сделать поиск более стабильным. Затем используйте поиск по сетке с диапазоном экспоненциально уменьшающихся размеров шага, по умолчанию от  $2^{10}$  до  $2^{-20}$ , принимая только те проходы, которые дают улучшение. Это позволяет найти подходящий начальный шаг, потому что слишком большие шаги будут расходиться к  $\infty$ , а также повысить точность, потому что перезапуск поиска с меньшим начальным шагом из найденного решения слишком быстро преодолевает влияние от  $s_i \rightarrow 0$ :

```
template<typename POINT, typename FUNCTION> POINT SPSA(POINT x,
    FUNCTION const& f, int maxEvals = 10000, double initialStep = 1)
{
    POINT direction = x;
    for(int i = 0, D = x.getSize(); i < maxEvals/2; ++i)
    {
        for(int j = 0; j < D; ++j) direction[j] =
            GlobalRNG().next() % 2 ? 1 : -1;
        double step = initialStep/pow(i + 1, 0.101), temp =
            (f(x + direction * step) - f(x - direction * step))/
            (2 * pow(i + 1, 0.501));
        if(!isfinite(temp)) break;
        for(int j = 0; j < D; ++j) x[j] -= temp/direction[j];
    }
    return x;
}

template<typename POINT, typename FUNCTION> pair<POINT, double> metaSPSA(
    POINT x, FUNCTION const& f, int spsaEvals = 100000, int estimateEvals =
    100, double step = pow(2, 10), double minStep = pow(2, -20))
```

```

{
    pair<POINT, double> xy(x, numeric_limits<double>::infinity());
    for(;; step > minStep; step /= 2)
    {
        if(isfinite(xy.second)) x = SPSP(xy.first, f, spsaEvals, step);
        double sum = 0;
        for(int i = 0; i < estimateEvals; ++i) sum += f(x);
        if(sum/estimateEvals < xy.second)
        {
            xy.first = x;
            xy.second = sum/estimateEvals;
        }
    }
    return xy;
}

```

## 24.13. Линейное программирование

Предположим, нужно оптимизировать линейную функцию с учетом линейных ограничений. Для простоты обозначение транспонирования вектора будет опущено. *Симплекс-метод* решает задачу линейного программирования:

- ◆ минимум  $cx$  при условии:
- ◆  $Ax = b$ ;
- ◆  $x \geq 0$ ,

где  $A$  — матрица размера  $n \times m$ , где  $n \geq m$ , векторы  $c$  и  $x$  размера  $n$  и  $b$  размера  $m$ .

Значения  $x$  упорядочены так, что  $x = (x_b, x_n)$ , где  $x_b$  — вектор из  $m$  ненулевых *базовых переменных*, а  $x_n$  из  $n - m$  *нулевых переменных*. Аналогично  $C = (c_b, c_n)$  и  $A = (A_b, A_n)$ :

- ◆  $\min z = c_b x_b + c_n x_n$  с учетом:
- ◆  $A_b x_b + A_n x_n = b$ ;
- ◆  $x \geq 0$ .

Можно получить все решения, решив  $Ax = b$  для любой комбинации основных вариантов выбора переменных. Жадный локальный поиск работает более эффективно (см. [24.16]):

1. Начать с исходного базиса.
2. Пока решение не будет оптимальным:
3. Улучшать текущий базис путем замены базовой переменной на небазовую.

Пусть  $y = c_b A_b^{-1}$ . Тогда  $z = yb + (c_n - yA_n)x_n$ . Если  $c_n - yA_n > 0$ , нельзя уменьшить  $z$ , увеличив любое  $x_n$ , поэтому нельзя улучшить текущий базис путем замены переменных. Линейное программирование — выпуклое, поэтому такой локальный критерий оптимальности является глобальным и позволяет выбрать в качестве *входной неосновной переменной* ту, которая имеет наименьшее значение  $c_n - yA_n$ .

*Уходящей переменной* называют ту, которая позволяет входной переменной принимать максимальное положительное значение, сохраняя при этом другие переменные  $\geq 0$ .

Пусть вектор  $a$  будет столбцом  $A_n$ , соответствующим входной переменной, и  $v = A_b^{-1}sa$ . Поскольку все компоненты  $x_n$ , кроме выбранного  $s$ , остаются равными 0,  $b = A_b(x_b + v)$ . Чтобы значение  $b$  оставалось прежним,  $s$  увеличивалось, а базис был верным, одна компонента  $x_b$  должна уменьшиться до 0, а остальные изменять значение, но остаться  $\geq 0$ . Если  $v \leq 0$ , задача допускает бесконечно большое решение, поскольку произвольное значение  $s$  может стать очень большим при увеличении одной из базисных переменных. Переменная  $i$ , уменьшение которой до 0 дает максимальное  $s$ , минимизирует  $\frac{x_b(i)}{v(i)}$  при  $v(i) > 0$ , где  $x_b = A_b^{-1}b$ . Реализация позволяет избежать численно неустойчи-

вой инверсии с использованием LUP-разложений  $A_b$  и ее транспонирования для решения эквивалентных уравнений. Помните, что обратная и транспонированная матрицы коммутативны. В качестве альтернативы вместо транспонирования можно вычислить  $yA_n = c_b A_b^{-1} A_n$ , используя тот факт, что последнее матричное умножение можно выполнить, осуществляя решение по одному столбцу за раз (см. главу 22. Численные алгоритмы: введение и матричная алгебра, где описаны методы, позволяющие избежать использования инверсий):

```
struct LinearProgrammingSimplex
{
    Matrix<double> B, N;
    Vector<double> b, cB, cN, x;
    Vector<int> p;
    bool isUnbounded;
    bool performIteration()
    {
        LUP<double> lup(B), lupT(B.transpose());
        x = lup.solve(b);
        // проверяем, является ли x оптимальным,
        // или находим входящую переменную
        Vector<double> y = cN - lupT.solve(cB) * N;
        int entering = 0;
        double bestValue = y[0];
        for(int i = 1; i < y.getSize(); ++i) if(y[i] < bestValue)
        {
            bestValue = y[i];
            entering = i;
        }
        if(bestValue >= 0) return false;
        // находим уходящую переменную
        Vector<double> a;
        for(int i = 0; i < N.rows; ++i) a.append(N(i, entering));
        a = lup.solve(a);
        int leaving = -1;
        double minRatio, maxA = -1;
        for(int i = 0; i < x.getSize(); ++i) if(a[i] > 0)
        {
            double newRatio = x[i]/a[i];
            if(leaving == -1 || minRatio > newRatio)
            {
                leaving = i;
            }
        }
    }
};
```

```

        maxA = max(maxA, a[i]);
        minRatio = newRatio;
    }
}

if(maxA <= 0){isUnbounded = true; return false;}
// меняем переменные местами
for(int i = 0; i < N.rows; ++i){swap(B(i, leaving), N(i, entering));}
swap(p[leaving], p[entering]);
swap(cB[leaving], cN[entering]);
return true;
}
LinearProgrammingSimplex(Matrix<double>const&B0, Matrix<double>
    const& N0, Vector<double>const& cB0, Vector<double>const& cN0,
    Vector<double> const& b0): isUnbounded(false), B(B0), N(N0), cB(cB0),
    cN(cN0), b(b0), x(b)
{for(int i = 0; i < cB.getSize() + cN.getSize(); ++i) p.append(i);}
Vector<pair<int, double> > solve()
{
    while(performIteration());
    Vector<pair<int, double> > result;
    if(!isUnbounded)
        for(int i = 0; i < x.getSize(); ++i)
            result.append(make_pair(p[i], x[i]));
    return result;
}
};

```

Симплекс-метод не срабатывает, если  $A_b$  сингулярна, задача неограничена или *вырождена*, т. е. сходится к бесконечному циклическому переключению между двумя равными базисами равного значения. Более сложные правила выбора переменных позволяют избежать вырождения, но на практике достаточно учитывать ошибки округления. Количество итераций в худшем случае экспоненциально, но на практике линейно. Каждая итерция выполняется за время  $O(m^3)$  из-за LUP (в комментариях приведено дополнительное пояснение).

В общем случае линейное программирование минимизирует линейную комбинацию  $n$  переменных с учетом  $m$  ограничений неравенства на линейные комбинации переменных:

- ◆ минимум  $sx$  при условии:
- ◆  $Ax = b$ ;
- ◆  $x \geq 0$ .

Допустимая область состоит из значений  $x$ , которые удовлетворяют ограничениям. Избыточные ограничения на это не влияют. Чтобы превратить неравенства в равенства, нужно прибавить вектор *переменной резерва* и дополнить  $A$  тождеством  $m \times m$ :

- ◆ минимум  $sx$  при условии:
- ◆  $Ax + Is = b$ ;
- ◆  $x \geq 0$  и  $s \geq 0$ .

Другие преобразования допускают дальнейшие обобщения:

- ◆ если  $x(i) < 0$  для некоторого  $i$ , заменить  $x$  на  $x_1, x_2 \geq 0$ , такое что  $x_1 - x_2 \geq 0$ ;
- ◆ заменить любое равенство ограничений  $\sum A_{ji}x_i = b_j$  как на  $\sum A_{ji}x_i \geq b_j$ , так и на  $-\sum A_{ji}x_i \geq -b_j$ , чтобы обеспечить простой начальный допустимый базис.

В стартовом базисе переменные резерва равны нулю. Если это невозможно, расширьте допустимую область, включив в нее начало координат, добавьте к каждому нарушенному ограничению искусственную переменную с начальным значением  $> 0$ , а стоимость каждой переменной задайте равной 1, если она искусственная, и 0 в противном случае. Решение делает искусственные переменные равными 0 и завершается успешно, если программа имеет хотя бы одно допустимое решение. Тогда текущие значения обычных переменных будут приемлемы для начала программы.

Чтобы составить надежную реализацию, нужно учесть много деталей, описанных в работах [24.27 и 24.33].

## 24.14. Некоторые соображения о нелинейном программировании

Наиболее общая задача оптимизации с ограничениями выглядит следующим образом:

- ◆ минимум  $f(x)$  при условии:
- ◆  $g_i(x) \leq 0$ ;
- ◆  $h_i(x) = 0$ .

Если  $f$  и  $g_i$  выпуклы, а  $h_i$  линейна, задача является выпуклой. Если допустимая область, определяемая  $g_i$  и  $h_i$ , выпукла, а сами функции невыпуклые, можно преобразовать задачу в эквивалентную, в которой функции будут выпуклы (см. [24.10]).

Лагранжиан задачи равен  $L(x, l, v) = f(x) + \sum l_i g_i(x) + \sum v_i h_i(x)$ .

Пусть  $x^*$  — оптимальное решение.

Двойственная задача  $g(l, v) = \min_{x \in \text{допустимая область}} L(x, l, v)$  вогнута и  $< x^* \forall l, v$ . Для задачи требуется максимально возможная нижняя граница решения, для которой зададим отрицательное значение от целевого:

- ◆ минимум  $-g(l, v)$  при условии:
- ◆  $l_i \geq 0$ .

Задача выпуклая. Например, для линейной программы:

- ◆ минимум  $cx$  при условии:
- ◆  $Ax = b$ ;
- ◆  $x \geq 0$ ;

$g(a, b) = -bv$ , если  $Av - l + c = 0$ , и  $-\infty$  в противном случае. Итак, двойственная задача формулируется следующим образом:

- ◆ минимум  $bv$  при условии:
- ◆  $-Av \leq c$ .

Замена  $Ax = b$  на  $Ax \leq b$  приводит к замене  $-Av \leq c$  на  $-Av = c$ .

Если  $d^*$  — оптимальное решение двойственной задачи,  $d^* \leq x^*$ . Такая ситуация называется *слабой двойственностью*. Для *сильной двойственности* имеет место равенство  $d^* = x^*$ , если существует *квалификация ограничений*. Примеры квалификаций:

- ◆  $g_i(x^*)$  и  $h_i(x^*)$  линейно независимы;
- ◆  $g_i$  и  $h_i$  линейны;

Задача выпуклая, и для нее существует допустимое решение.

Если имеет место сильная двойственность, для любого оптимального набора  $x^*, l^*, v^*$  выполняются оптимальные *условия Каруши — Куна — Таккера* (ККТ):

- ◆  $g(x^*) \leq 0$ ;
- ◆  $h(x^*) = 0$ ;
- ◆  $l^* \geq 0$ ;
- ◆  $l^* g(x^*) = 0$ ;
- ◆  $\nabla L(x^*, l^*, v^*) = 0$ .

Если задача выпуклая, она имеет единственное решение. Двойственность и условия ККТ позволяют упростить задачу. Например, двойственное линейное программирование оказывается проще, если в задаче меньше переменных, чем ограничений.

Конкретные алгоритмы приведены в работах [24.32, 24.10] и в ссылках из них. Основные реализации рассмотрены в [24.1]. В работе [24.43] содержится множество актуальных обзоров с богатой библиографией и приложениями.

Основная трудность заключается в том, что случайная выборка не позволяет работать с ограничениями строгого равенства, поэтому для получения решения требуется дополнительная информация о задаче. Так, для решения ограничений равенства может понадобиться ручная предварительная обработка. Если существуют ненулевые допустимые области (т. е. без ограничений равенства), для адаптации обсуждаемых глобальных методов можно использовать случайную выборку.

Общий метод состоит в том, чтобы запускать неограниченные алгоритмы как есть, но изменить неограниченную функцию так, чтобы она возвращала  $\infty$  в недопустимых точках. Тогда для выпуклой  $f$  и выпуклой допустимой области без ограничений равенства многие алгоритмы дают разумные ответы даже без явных проекций их пути поиска на активное подмножество ограничений. Допускается также использовать большой ограниченный штраф, который можно постепенно увеличивать.

Выпуклость имеет решающее значение для поиска решения, а для невыпуклых задач эффективных методов нет.

## 24.15. Примечания по реализации

Почти в каждой реализации есть что-то оригинальное, и основной проблемой чаще всего был бы выбор алгоритма.

- ◆ L-BFGS — единственный алгоритм, работающий на основе градиента, и я дополнил его множеством эвристик устойчивости.
- ◆ Поиск Фибоначчи с расширениями является единственным надежным алгоритмом одномерного поиска.

- ♦ Алгоритмы глобальной оптимизации, как правило, являются расширениями соответствующих комбинаторных алгоритмов.

Реализация линейного программирования очень проста, и в ней можно использовать дополнительные приемы для повышения эффективности, но я решил сделать ее простой и не представлять более сложные алгоритмы, потому что для их хорошей реализации требуются большие усилия.

## 24.16. Комментарии

Метод золотого сечения является оптимальным для наихудшего случая, *поиск Брента* (который намного сложнее — см. [24.36]) для гладких  $f$  может работать быстрее.

В задаче сильного поиска по линии Вульфа в более сложных стратегиях удвоения/масштабирования используют квадратичную или кубическую интерполяцию на основе последних значений  $f$  и производной по направлению, но в таких задачах нужна защита от наихудшего случая — например, выполняемое время от времени деление пополам (см. [24.32, 24.29]). Они далее не рассматриваются главным образом потому, что при наличии конечно-разностных градиентов имеет смысл сначала выполнить «точный» поиск. Не стоит рассматривать предположительно лучший линейный поиск, основанный на приблизительных условиях Вульфа (см. [24.18]), которые предназначены для учета вычислений с плавающей точкой. Он не так популярен, как описанная в этой главе версия, и ошибки в числовых градиентах лишают смысла учет плавающей точки.

Для надежных L-BFGS лучше проверить угол между ступенькой и самым крутым спуском. Например, может потребоваться, чтобы косинус угла был  $< 0,01$ . Но в плохо обусловленных случаях такие тесты не работают, потому что могут потребоваться шаги с небольшим углом (см. [24.32]).

При использовании алгоритма BFGS можно установить значение  $B$ , равное градиентно-конечно-разностному приближению гессиана. Затем его разложение Холецкого обновляется с изменениями, например QR для метода Бройдена. Это также позволяет обеспечить положительную определенность и отслеживать обусловленность  $B$ , а также перезапускать или предпринимать другие корректирующие действия, если это необходимо (см. [24.15]). По-видимому, это дает большую стабильность по сравнению с формулой BFGS (в которой используется формула Шермана — Моррисона). Но, как уже было сказано, это сложно и неэффективно. В частности, это не позволяет использовать ограниченную память, время факторизации  $B$   $O(D^3)$  в начале не масштабируется, а неуклюжий мониторинг потенциально устаревшего состояния для числовых задач слишком медленный, и его проще вообще не запоминать. Никакой объем мониторинга состояния не может гарантировать сохранение хорошего численного состояния, и время от времени его необходимо перезапускать. Этого не стоит делать даже в алгоритме BFGS (см. [24.32]), потому что для хорошей работы достаточно начать с идентифицирующего значения  $B$ . В общем случае для поддержки полной модели лучше подходят методы доверительной области (обсуждаемые позже).

Методы *сопряженного градиента* (см. [24.32]) являются несколько более простой альтернативой L-BFGS, но у них есть проблемы:

- ♦ их производительность хуже. В работе [24.30] отмечается, что, по опыту ее автора, классические методы сопряженных градиентов работают не особенно хорошо. Бо-



лее новая формула Хагера — Чжана (см. [24.18]) работает быстрее (по мнению авторов), чем L-BFGS, несмотря на большее количество оценок. Кроме того, в последней версии этого метода (см. [24.18]) используется память, и он сопоставим с L-BFGS по числу оценок, но является более экономичным при обновлении состояния. Однако для дорогих  $f$  обновление состояния не является узким местом, а обновление L-BFGS дороже лишь в постоянном множителе, и чтобы установить достоинства этого метода, необходимы дополнительные исследования. В общем, количество оценок должно быть единственной интересующей метрикой.

- ◆ Теория сходимости этого метода сильно отличается. Алгоритм L-BFGS в конечном счете основан на методах Ньютона с положительно-определенным исполнением. В методе сопряженного градиента используется собственный подход и сохраняется ортогональность, основанная на предположениях, что значение минимизируется точно после  $D$  итераций. Но при больших отклонениях от квадратичных или плохо обусловленных гессианов возникают вычислительные проблемы, поэтому требуется периодический перезапуск. Вопрос о том, может ли версия с использованием памяти избежать этого, еще предстоит исследовать. Также в этой главе акцент делается на случае, когда для оценки градиентов задействуется конечная разность, которая кажется более стабильной при использовании ньютоновских методов, потому что положительную определенность поддерживать легче, чем ортогональность. Но этот вопрос тоже требует дальнейшего исследования.

Метод сопряженного градиента для гарантии сходимости меньше требует от задачи, (например, отсутствия ограничений на  $\kappa(H_0)$  — подробнее о старой формуле см. [24.32]), но на практике это, похоже, не имеет значения. Тайже почитайте работу [24.38].

*Методы доверенной области* (см. [24.32]) являются альтернативой линейному поиску для контроля глобальной сходимости алгоритмов. Идея их состоит в предположении, что модель метода (такая как неявная квадратичная модель метода Ньютона) действительна в пределах некоторого малого радиуса, и выборе следующей точки как наилучшей точки в модели с учетом такого ограничения. Принцип похож на поиск по компасу. Несмотря на сходимость при некоторых простых условиях и возможность адаптации радиуса доверия, есть и некоторые проблемы:

- ◆ проблему ограничения радиуса можно решить, запустив вложенный решатель до некоторой приемлемой точности, но это сложнее, чем линейный поиск;
- ◆ в модели Ньютона подход масштабируется только в том случае, если гессиан разрежен и может быть вычислен. В *усеченном алгоритме Ньютона* используются линейные предварительно обусловленные сопряженные градиенты до тех пор, пока не будет обнаружена неположительная определенность.

Масштабируемые алгоритмы линейного поиска, такие как L-BFGS, в целом работают лучше методов доверенной области. Последние, по-видимому, не дают преимуществ при решении уравнений. Но у них нет проблем в работе с неположительно определенным гессианом, что дает преимущество в специфических для приложения случаях, когда есть информация о второй производной, значение  $D$  мало, а гессиан разреженный. Это часто имеет место в задачах наименьших квадратов, где используются специализированные алгоритмы, такие как *алгоритм Левенберга — Марквардта* (см. [24.32]).

Обобщением поиска по компасу является *адаптивный прямой поиск сетки* (Mesh Adaptive Direct Searches, MADS) (см. [24.2]). Это позволяет получить плотный набор

направлений в пределе, если алгоритм застопорился, в обход ограничений координатного спуска и компаса для недифференцируемых  $f$ . Но механика алгоритма слишком неуклюжая, чтобы обеспечить сходимость. Поиск по компасу сходится на дифференцируемых  $f$ , неявно работает на сетке степеней двойки от начального шага (подробное описание теории — в работе [24.23]). Алгоритм MADS очень ловко поддерживает эту сетку. Вместо этого можно просто использовать случайные направления и уменьшать шаг на  $2^{-2n}$  в случае неудачи, в противном случае удваивать его. Технически это несходимость, потому что не налагает никаких достаточных условий спуска (если только не использовать трудномасштабируемую *принудительную функцию* — см. [24.23]), но удобство соизмеримо с MADS.

В задаче глобальной оптимизации я пробовал несколько других метаэвристик, но для протестированных задач они оказались лучше гибридного алгоритма и его компонентов. Приведенные сравнения не являются статистически значимыми, а предназначены лишь для исключения плохих алгоритмов. Правильное сравнение выполнить сложно. В работе [24.12] описано несколько идей о том, что имеет смысл, а что — нет. Все подборки и комбинации основаны на моих собственных экспериментах.

При отсутствии статистических данных стоит отдавать предпочтение простым методам, которые легче для понимания, сходятся и хорошо работают в худшем случае. Методы, не обладающие такими характеристиками, не обязательно плохие, но и сильно лучше они быть не могут. Многие метаэвристики возникают за счет роста популярности, когда предложенный алгоритм показывает себя немного лучше в некотором наборе задач, а затем применяется для решения практических задач в последующих статьях. Поскольку у разработчиков алгоритмов имеется значительный коммерческий интерес, необходимо задавать вопросы:

- ◆ работает ли этот метод лучше на других  $f$ , и правильно ли настроены конкуренты;
- ◆ может ли другой алгоритм так же хорошо или лучше справиться с практическими задачами?

Некоторые общие идеи (в дополнение к тому, что обсуждается в *главе 16. Комбинаторная оптимизация*):

- ◆ многие алгоритмы вдохновлены естественными процессами. Это может привести к получению удачной смещенной выборки чисто случайно, а не потому, что выбор строился на какой-то качественной логике. Такие алгоритмы необходимо упрощать или улучшать с помощью математического анализа и экспериментов, и естественный процесс лишь изредка дает намеки на определенный выбор;
- ◆ из-за NFL и сложности надлежащего тестирования были разработаны многочисленные алгоритмы и предложены различные варианты. Для сокращения поиска можно сделать следующее:
  - рассмотрения заслуживают только алгоритмы, имеющие хорошую производительность и применимость;
  - у алгоритма не должно быть слишком много лишней логики настройки, а их механика должна быть близка к задаче;
  - все параметры, кроме предельного оценочного бюджета, должны иметь правильные значения по умолчанию;

- пробуются только одна версия алгоритма — обычно это базовая версия с наиболее логичными значениями параметров;
- алгоритмы, чувствительные к незначительным изменениям, нестабильны, и их выбор ненадежен.

В работе [24.45] приведено наглядное обсуждение оптимизационных ландшафтов с точки зрения плавной оптимизации.

При работе с неограниченными независимыми случайными данными и марковскими семплерами вопрос состоит в том, насколько толстым должен быть хвост, и ответа на этот вопрос нет. Распределение Парето (см. [24.48]) может дать хвост  $x > 0$  с вероятностью  $O(x^a)$ . Возможно, самый толстый полезный хвост дает значение  $a = 0.3$  или около того, но неясно, каковы его преимущества, потому что требуется учитывать диапазон с плавающей точкой с очень большой вероятностью. Здесь, чтобы получить отрицательный шаг и шаг в диапазоне  $[-1, 1]$ , можно использовать смешанное распределение, если это по какой-либо причине необходимо. Распределение Леви устойчиво, т. е. несколько шагов Леви эквивалентны одному шагу с разными параметрами, что может быть полезно. Но ни один хвост не может быть достаточно толстым для действительно глобального поиска, потому что любое такое распределение имеет определенную медиану (равную 1 в приведенных реализациях). Таким образом, точки внутри гиперсферы с радиусом, равным медиане, так же вероятны, как и точки снаружи, что является локальным уклоном. Таким образом, хотя локальный марковский поиск технически сходится глобально, это не имеет особого смысла, поскольку в любой выборке ему может повезти.

Некоторые конкретные алгоритмы (подробности приведены в ссылках, если интересно, а в приведенных описаниях даны только варианты реализации):

- ♦ *оптимизация роя частиц* не рассматривалась, потому что для нее требуется гораздо больше настроек и параметров, чем кажется разумным. Кроме того, в некоторых исследованиях, таких как [24.11], этот и некоторые другие недавние алгоритмы не превзошли алгоритм дифференциальной эволюции, который 20 лет не менялся, в то время как все другие алгоритмы в исследовании имеют недавние, еще не проверенные предложения по улучшению;
- ♦ *CMA-ES Холецкого*. Алгоритм CMA-ES и его варианты хорошо зарекомендовали себя во многих недавних тестах. Алгоритм Холецкого является самым простым и требует обновления  $O(D^2)$  (большинству других требуется  $O(D^3)$ ). Реализация в точности выполнена по рекомендациям [24.3]. Параметр  $\sigma$  устанавливается в обычную начальную шкалу  $x = \max(1, \|x\|_2)/10$ . Удивительно, но он работал хуже, чем *random-before-local search* (да, я проверял реализацию на корректность). Его вариант использования, по-видимому, представляет собой только шумную локальную минимизацию, где он заменит методы, основанные на Ньютоне.

Выбор представительный, хотя, конечно, не исчерпывающий. Любой хороший алгоритм не должен пытаться искать иголку в стоге сена, потому что это расточительно и вряд ли даст результаты.

Не решен вопрос о том, имеют ли на практике популяционные методы общее преимущество перед методами с одним решением. В настоящий момент кажется, что преимущества нет. Похоже, что популяционные методы просто запускают взаимодействующие семплеры на основе состояний и технически не должны превосходить марковский

поиск и некоторые из его очевидных расширений, таких как выполнение нескольких шагов. Также возможен коллапс популяции и ненужные оценки некачественных членов популяции (выборки независимых случайных данных должны быть менее расточительными). Кажется, что в популяции можно задействовать память, но это полезно только тогда, когда она используется правильно. Сходимость методов с одним решением обычно легче доказать. Для популяционных методов коллапс иногда приводит к полной стагнации алгоритма.

Алгоритм NFL обходит многие интересные вопросы. Например, всегда ли лучше использовать распределения с толстым хвостом? Нет, если глобальный минимум находится недалеко. Но в этом случае потеря эффективности невелика, а когда она большая, выборка с толстым хвостом сработает лучше. Функции  $f$ , в которых нельзя использовать какие-либо отклонения, называются *вводящими в заблуждение*, и для таких функций от любого алгоритма не стоит ждать хорошей производительности. Например, при оптимизации функции в 2D путем просмотра графика обычно не думают, что функция может иметь сингулярность между пикселями, хотя это возможно. Возможность встретить вводящую в заблуждение  $f$  увеличивает потребность в гибридных алгоритмах, в которых распространенные случаи ошибок обрабатываются.

Похоже, что наиболее полезным результатом метаэвристического исследования является поиск лучших способов исследования. Результаты исследования в дальнейшем могут использоваться в алгоритмах выбора. Возможные открытия:

- ◆ оператор мутации DE для глобальной выборки;
- ◆ выборка Леви для семплинга с толстым хвостом;
- ◆ упрощенная версия многих алгоритмов.

Почти у каждой известной метаэвристики есть предлагаемое улучшение, которое, однако, не обязательно считается улучшением. Разработка алгоритмов, которые хорошо работают только на стандартных наборах тестов, приводит к избыточной сложности алгоритма.

Оптимизация *метамодели* является многообещающим методом. Попробуйте найти многомерную интерполяцию проблемы и оптимизируйте ее, чтобы отыскать следующую точку выборки. В двух измерениях хорошим подходом является *Кригинг/гауссовский процесс/байесовская оптимизация*, но для нее нужно много вычислительных ресурсов, а на больших значениях  $D$  она не используется. Для очень дорогих  $f$ , таких как моделирование автомобильных аварий или длительные симуляции, этот метод, безусловно, предпочтителен. Метамоделью может быть любой метод регрессии (т. е. случайный лес, а не Кригинг), и ее можно оптимизировать с помощью методов «черного ящика», описанных в этой главе. Поскольку метамодель, как правило, начинается с нескольких случайных/узловых точек, бюджет вычислений должен превышать небольшое количество оценок.

Если  $f$  не является полностью «черным ящиком» и содержит некоторую полезную информацию, например верхнюю границу константы Липшица, задача глобальной оптимизации решается точно с использованием методов ветвей и границ (см. [24.26]). В некоторых случаях можно эвристически оценить константу Липшица, например, используя статистику экстремального порядка для вывода границ минимального и максимального  $f$ -значений. В работе [24.35] приведено много руководств по внедрению.

Для симплекс-метода использование алгебры разреженных матриц и динамическая поддержка LUP реализуются сложно, но существенно ускоряют вычисления (см. [24.16]). В результате получаются «слишком техничные» алгоритмы, потому что для получения хорошей производительности нужны хитрые приемы.

Для решения линейных задач можно использовать *методы внутренних точек*. Симплекс пересекает края допустимой области, а эти методы проходят через внутреннюю часть. Это занимает больше времени на итерацию, но они сходятся за число итераций  $O(\text{полином})$ , обычно постоянное на практике.

## 24.17. Советы по дополнительной подготовке

- ◆ Работает ли поиск по золотому сечению на дискретных  $f$ ? Какие модификации необходимы и как можно доказать его правильность и завершение?
- ◆ В одномерном случае реализуйте метод конечных разностей Ньютона, в котором повторно для вычисления второй производной используются оценки  $f$ . Сравните эффективность с эффективностью других методов.
- ◆ Реализуйте метод искусственной переменной для задачи линейного программирования.
- ◆ Повысьте эффективность симплекс-метода линейного программирования, обновив матричную факторизацию, вместо ее пересчета на каждой итерации. Используйте плотную матричную алгебру.
- ◆ Исследуйте и реализуйте некоторые распространенные методы внутренних точек для линейного программирования.
- ◆ Исследуйте и реализуйте некоторые распространенные методы внутренних точек и другие методы нелинейного программирования.
- ◆ Изучите метод MADS из работы [24.2] в качестве альтернативы компасу. Научите метод работать с ограничениями координат.
- ◆ Исследуйте основанные на моделях алгоритмы без производных (см. [24.2 и 24.14], чтобы познакомиться с концепцией). Из экспериментов, описанных в работе [24.19], следует, что *алгоритм NEWUOA* Пауэлла является наиболее многообещающим. Он строится постепенно из нескольких оценок как квадратичная модель, но из комментариев, приведенных в работе [24.14], следуют некоторые улучшения.
- ◆ Для марковского поиска вместо шага по Леви можно попробовать обычный шаг аналогичного масштаба. Как это повлияет на производительность?
- ◆ Научите локальный поиск изменять значение  $x$  по ссылке для эффективности при работе с несколькими шагами и большим  $n$ .
- ◆ Исследуйте и внедрите вариант имитации отжига из GenSA (см. [24.50]), который хорошо работает согласно [24.30].
- ◆ Ознакомьтесь с другими вариантами CMA-ES. Алгоритм IPOP-CMA-ES считается среди них эталоном. Попробуйте также ILS с реализованной здесь версией, которая по сути является локальным поиском. Особенно интересны версии с ограниченной памятью, вдохновленные L-BFGS (см. [24.44] и ссылки в этой работе). Эта область все еще развивается.

- ◆ Исследуйте и внедрите байесовскую оптимизацию. Полученный алгоритм испытайте для выбора параметра машинного обучения вместо используемого в настоящее время метода оптимизации.
- ◆ Исследуйте и сравните существующие методы оптимизации шума. Используйте существующий тестовый набор с относительно низким уровнем генерируемого шума. Алгоритм случайного поиска может быть хорошим эталоном.
- ◆ Возьмите алгоритм дифференциальной эволюции и генетический алгоритм скрещивания и попробуйте смешать их в пропорции 90–10%, а не по 50%. Повышает ли это производительность за счет сохранения хороших строительных блоков?
- ◆ При небольших бюджетах оценки лучше сравнивать производительность с начальным значением, что позволяет узнать снижение в процентах, возможно, по логарифмической шкале. Для этого можно выполнить сравнение производительности. Стали ли выводы более информативны?

## 24.18. Список рекомендуемой литературы

- 24.1. Andrei N. (2017). Continuous Nonlinear Optimization for Engineering Applications in GAMS Technology. Springer.
- 24.2. Audet C., & Hare W. (2017). Derivative-Free and Blackbox Optimization. Springer.
- 24.3. Bäck T., Foussette C., & Krause P. (2013). Contemporary Evolution Strategies. Springer.
- 24.4. Bagirov A., Karimtsa N., & Mäkelä M. M. (2014). Introduction to Nonsmooth Optimization: Theory, Practice and Software. Springer.
- 24.5. Beck A. (2017). First-Order Methods in Optimization. SIAM.
- 24.6. Bénichou O., Loverdo C., Moreau M., & Voituriez R. (2011). Intermittent search strategies. *Reviews of Modern Physics*, 83(1), 81.
- 24.7. Bergstra J., & Bengio Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb), 281–305.
- 24.8. Bornemann F., Laurie D., Wagon S., & Waldvogel J. (2004). The SIAM 100-digit Challenge: a Study in Highaccuracy Numerical Computing. SIAM.
- 24.9. Bousquet O., Gelly S., Kurach K., Teytaud O., & Vincent D. (2017). Critical Hyper-Parameters: No Random, No Cry. arXiv preprint arXiv:1706.03200.
- 24.10. Boyd S., & Vandenberghe L. (2004). Convex Optimization. Cambridge University Press.
- 24.11. Civicioglu P., & Besdok E. (2013). A conceptual comparison of the Cuckoo-search, particle swarm optimization, differential evolution and artificial bee colony algorithms. *Artificial intelligence review*, 1–32.
- 24.12. Clerc M. (2015). Guided Randomness in Optimization. Wiley.
- 24.13. Clerc M. (2019). Iterative Optimizers: Difficulty Measures and Benchmarks. Wiley.
- 24.14. Conn A. R., Scheinberg K., & Vicente L. N. (2009). Introduction to Derivative-Free Optimization. SIAM.
- 24.15. Dennis Jr J. E., & Schnabel R. B. (1996). Numerical Methods for Unconstrained Optimization and Nonlinear Equations. SIAM.
- 24.16. Griva I., Nash S. G., & Sofer A. (2009). Linear and Nonlinear Optimization. SIAM.
- 24.17. Hager W. W., & Zhang H. (2005). A new conjugate gradient method with guaranteed descent and an efficient line search. *SIAM Journal on optimization*, 16(1), 170–192.

- 24.18. Hager W. W., & Zhang H. (2013). The limited memory conjugate gradient method. *SIAM Journal on Optimization*, 23(4), 2150–2168.
- 24.19. Hansen N., Auger A., Ros R., Finck S., & Pošík P. (2010, July). Comparing results of 31 algorithms from the black-box optimization benchmarking BBOb-2009. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation* (pp. 1689–1696).
- 24.20. Hastie T., Tibshirani R., & Wainwright M. (2015). *Statistical Learning with Sparsity: The Lasso and Generalizations*. CRC Press.
- 24.21. Jamil M., & Yang X. S. (2013). A literature survey of benchmark functions for global optimization problems. *International Journal of Mathematical Modeling and Numerical Optimization*, 4(2), 150–194.
- 24.22. Kar Mitsa N. (2015). Diagonal bundle method for nonsmooth sparse optimization. *Journal of Optimization Theory and Applications*, 166(3), 889–905.
- 24.23. Kolda T. G., Lewis R. M., & Torczon V. (2003). Optimization by direct search: New perspectives on some classical and modern methods. *SIAM review*, 45(3), 385–482.
- 24.24. Lewis A. S., & Overton M. L. (2013). Nonsmooth optimization via quasi-Newton methods. *Mathematical Programming*, 1–29.
- 24.25. Liu D. C., & Nocedal J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1–3), 503–528.
- 24.26. Locatelli M., & Schoen F. (2013). *Global Optimization: Theory, Algorithms, and Applications*. SIAM.
- 24.27. Maros I. (2002). *Computational Techniques of the Simplex Method*. Springer.
- 24.28. Moré J. J., Garbow B. S., & Hillstom K. E. (1981). Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, 7(1), 17–41.
- 24.29. Moré J. J., & Thuente D. J. (1994). Line search algorithms with guaranteed sufficient decrease. *ACM Transactions on Mathematical Software (TOMS)*, 20(3), 286–307.
- 24.30. Nash J. C. (2014). *Nonlinear Parameter Optimization Using R Tools*. Wiley.
- 24.31. Nesterov Y. (2018). *Lectures on Convex Optimization*. Springer.
- 24.32. Nocedal J., Wright S. (2006). *Numerical Optimization*. Springer.
- 24.33. PAN P (2014). *Linear Programming Computation*. Springer.
- 24.34. Penot J. P. (2012). *Calculus Without Derivatives*. Springer.
- 24.35. Pintér J. D. (1996). *Global Optimization in Action: Continuous and Lipschitz Optimization: Algorithms, Implementations and Applications*. Springer.
- 24.36. Press W. H., Teukolsky S. A., Vetterling W. T., & Flannery B. P. (2007). *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge University Press.
- 24.37. Price K., Storn R. M., & Lampinen J. A. (2005). *Differential Evolution: A Practical Approach to Global Optimization*. Springer.
- 24.38. Pytlak R. (2008). *Conjugate Gradient Algorithms in Nonconvex Optimization*. Springer.
- 24.39. Simon D. (2013). *Evolutionary Optimization Algorithms*. Wiley.
- 24.40. Skajaa A. (2010). Limited memory BFGS for nonsmooth optimization. Courant Institute of Mathematical Science, New York, Master's thesis.
- 24.41. Spall J. C. (2003). *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. Wiley.
- 24.42. StackOverflow (2018). Approximate order statistics for normal random variables. <https://stats.stackexchange.com/questions/9001/approximate-order-statistics-for-normal-random-variables>. Accessed January 6, 2018.
- 24.43. Terlaky T., Anjos M. F., & Ahmed S. (Eds). (2017). *Advances and Trends in Optimization with Engineering Applications*. SIAM.

- 24.44. Varelas K., Auger A., Brockhoff D., Hansen N., ElHara O. A., Semet Y., Kassib R. & Barbaresco F. (2018). A comparative study of large-scale variants of CMA-ES. In International Conference on Parallel Problem Solving from Nature (pp. 3–15). Springer.
- 24.45. Weise T. (2011). Global Optimization Algorithms: Theory and Application, 3rd ed. Downloadable from <http://www.it-weise.de/projects/bookNew.pdf>.
- 24.46. Wikipedia (2013). Stochastic approximation. [http://en.wikipedia.org/wiki/Stochastic\\_approximation](http://en.wikipedia.org/wiki/Stochastic_approximation). Accessed May 18, 2013.
- 24.47. Wikipedia (2017a). Unimodality. <https://en.wikipedia.org/wiki/Unimodality>. Accessed December 10, 2017.
- 24.48. Wikipedia (2017b). Pareto distribution. [https://en.wikipedia.org/wiki/Pareto\\_distribution](https://en.wikipedia.org/wiki/Pareto_distribution). Accessed December 10, 2017.
- 24.49. Wikipedia (2018). Order statistic. [https://en.wikipedia.org/wiki/Order\\_statistic](https://en.wikipedia.org/wiki/Order_statistic). Accessed January 11, 2018.
- 24.50. Xiang Y., Gubian S., Suomela B., & Hoeng J. (2013). Generalized Simulated Annealing for Global Optimization: The GenSA Package. R Journal, 5(1).
- 24.51. Zhigljavsky A., & Zilinskas A. (2007). Stochastic Global Optimization. Springer



## 25. Введение в машинное обучение

Га-га-га

*Амелия Кедик, 1 год, когда смотрит на игрушечного пингвина, а ранее видела лишь гусей*

### 25.1. Введение

Мы рассмотрим здесь общие темы, характерные для большинства задач машинного обучения — например, подготовку данных,. Обсуждается также соответствующая теория. Обязательным условием является хорошее понимание статистики.

Основная идея заключается в том, что машинное обучение — это не что-то автоматическое. Предположим, что некоторый паттерн является шумом, если не доказано обратное. В большинстве областей, где данные лишь незначительно связаны с требуемой информацией, получение информации из данных — это лучшее, на что можно рассчитывать. Кроме того, производительность развернутой обучающей системы, скорее всего, будет гораздо ниже, чем у тестовой, поскольку данные накапливаются медленно, а пользователи пытаются обмануть систему. Тем не менее, когда в данных присутствуют устойчивые паттерны, а алгоритм правильно изучает их, результат может оказаться ценным.

### 25.2. Что такое машинное обучение?

Представьте, что вы хотите взять ипотеку. Вы ловите на улице банкира, здороваетесь с ним и просите его дать вам ипотечный кредит. Может ли он дать вам его? Нет, потому что он ничего не знает о вашей ситуации.

Тогда вы идете в банк и приносите с собой кое-какие документы:

- ♦ ваш кредитный рейтинг = 680;
- ♦ ваша сумма первоначального взноса = 30%;
- ♦ ежемесячная плата на содержание дома  $\approx 1000$  долларов США.

Это здорово, но что банк делает с этими данными? Подходят они ему или нет? Должен же быть какой-то надежный эталон, на котором основывается решение о выдаче ипотеки.

Предположим теперь, что банк купил базу данных прошлых ипотечных заявок с экспертными решениями банкира об одобрении или отклонении. Что теперь? Самая простая идея заключается в поиске ближайшего соседа: просканировать базу данных, найти «самую близкую» к вашей ситуации и принять решение на ее основе. «Близость» высчитывается по некоторой функции расстояния, такой как евклидова, на данных,

масштабированных на диапазон  $[0, 1]$ , чтобы не сравнивать яблоки с апельсинами (обсуждается позже в этой главе).

Но если решение будет отрицательным, а по закону решения по заявкам на ипотеку должны быть полностью прозрачными, такой подход не годится. Другой подход состоит в том, чтобы составить дерево решений на основе данных (обсуждается в *главе 26. Машинное обучение: классификация*), например:

- ◆ если кредитный рейтинг  $< 700$ :
  - если первоначальный взнос  $< 50\%$ , отклонить;
  - в противном случае одобрить;
- ◆ иначе:
  - если первоначальный взнос  $< 20\%$ , отклонить;
  - в противном случае одобрить.

*Машинное обучение автоматически выводит такую структуру решений из данных.* В результате вы получаете решения, обычно называемые *прогнозами*, по точкам данных, таким как отдельные заявки на ипотеку.

## 25.3. Математическое обучение

Нужно получить предиктор  $f$ , который в определенных ситуациях принимает правильные решения, представленный в виде объектов в некотором *пространстве признаков*  $X$ .  $X$  обычно является векторным пространством, но не обязательно. Для эффективного обучения подойдет и метрическое пространство, в котором можно получить расстояния между объектами. В векторном пространстве каждый объект представляет собой вектор *признаков*, выбранных на основе знаний предметной области. Обучение находит  $f$ , которое *разбивает*  $X$  на некоторое, возможно бесконечное, количество разделов, присваивая всем  $x$  идентификатор раздела  $ID$ , который может быть дискретным или непрерывным. Предполагается, что  $x$  содержит всю полезную информацию об объекте. Например, чтобы обнаружить одиночку в командном шутере, в качестве характеристик можно использовать количество встреченных врагов и исследованный % игрового пространства и считать игрока, который встретил 0 врагов и исследовал  $< 5\%$  пространства, одиночкой. Здесь  $ID \in \{\text{«да»}, \text{«нет»}\}$ . Предположим, что  $x \sim$  распределение  $P$ , а  $ID \sim$  условное распределение  $P|x$ . В зависимости от задачи последние могут быть непрерывными, дискретными или дельта-дираковскими. Последний случай — это детерминированное обучение, где для любого  $x$  существует уникальный идентификатор.

Вместо явного определения  $f$  (что может быть практически невозможно, например в задаче поиска лица на фотографии), алгоритм  $A$  видит конечное множество  $S$  из  $n$  примеров  $z_i$ , из которых он, как мы надеемся, узнает  $f$ , которое *обобщает* невидимое  $z$ . Для любого  $z_i$  существуют  $x_i$  и *информация о подсказках*  $y_i \in Y$ . Для любого  $A$  необходимо понимать свою стратегию разделения. Для задач *прогнозирования*  $ID_i = y_i =$  выборка из  $P|x_i$ , но  $y_i$  может быть произвольным или не существовать для некоторых или всех  $i$ . Функция  $y$  зависит от  $x$ ,  $ID$  и, возможно, другой информации.

Пусть требуется измерить эффективность обучения с точки зрения  $x$  и  $ID$ , имея только  $x$  и  $y$ . Функция *потерь*  $L(z) = L(x, y)$  измеряет ошибки  $f$  на  $z$ . Для безупречной работы

алгоритма функция  $L$  должна возвращать минимально возможное конечное значение, обычно 0. Риск  $R_f(P) = E_z[L_f(z)]$  измеряет ошибку обобщения  $f$ . Обычно предполагается, что все  $z$  одинаково важны.  $Z$  и  $L$  определяют учебную задачу, а  $P$  — задачу внутри нее.  $A$  обычно предназначен для минимизации риска для конкретной задачи, выполнение которой и является целью обучения. Типичные задачи:

- ◆ *классификация*:  $ID = y = \text{метка класса} \in [0, k - 1]$ . Предполагая, что правильные решения дают прибыль 0, а неправильные стоят 1,  $Lf(x, y) = (f(x) \neq y)$ , где  $R_f = E[\% \text{ неправильно классифицированных } z]$ ;
- ◆ *регрессия*:  $ID = y$  являются вещественными. Обычно  $Lf(x, y) = (f(x) - y)^2$ , т. е. предположим квадратичную стоимость;
- ◆ *кластеризация* — то же, что и классификация, но без подсказок, т. е.  $y = x$ . Часто используют функцию потерь  $L_f(x, y) = \text{расстояние}(f(x), x)$  до группы подобных  $x$ ;
- ◆ *частичное обучение с учителем* — подобно кластеризации, но в некоторых тестах есть дополнительная информация, которая помогает улучшить изученный раздел.  $L$  зависит от того, есть ли подсказка;
- ◆ *обучение с усилением значения-функции* — классификация или регрессия, когда подсказка присваивается последовательности решений. Например, в играх вроде шахмат существует логика, которая подсказывает, какой ход является хорошим, и если в конце такая логика выигрывает или проигрывает, это и становится подсказкой.

Например, можно распознать написанную от руки цифру по ее изображению, представленному массивом ячеек  $8 \times 8$  с оттенками серого цвета  $\in [0, 16]$  (рис. 25.1).

Приведенный далее набор данных о цветах ириса (рис. 25.2) содержит размеры чашелистиков и лепестков ириса, на основании которых нужно решить, какой это тип ириса.



Рис. 25.1. Примеры цифровых данных

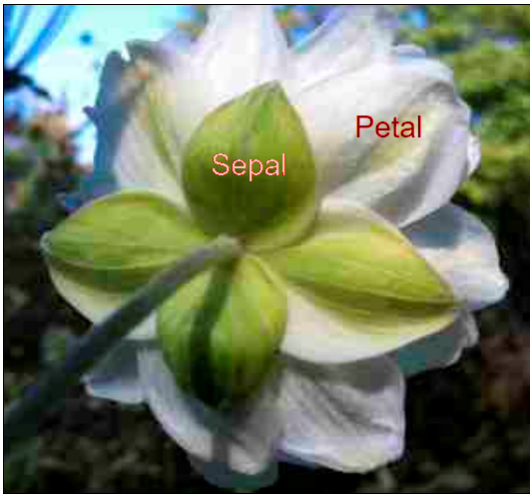


Рис. 25.2. Чашелистик (Sepal) похож на лепесток (Petal), но находится снизу и обычно зеленого цвета

Тип	Длина чашелистика	Ширина чашелистика	Длина лепестка	Ширина лепестка
Setosa	5,1	3,5	1,4	0,2
Setosa	5,4	3,9	1,7	0,4
Versicolor	6,4	3,2	4,5	1,5
Versicolor	5,6	2,9	3,6	1,3
Virginica	6,2	3,4	5,4	2,3
Virginica	7,2	3,6	6,1	2,5

Прогнозирование использования ресурсов  $A$  на основе его явных и неявных входных данных является типичной задачей регрессии. Вот пример промышленной задачи прогнозирования эффективности отопительной нагрузки на основе следующих данных:

Относительная компактность	Площадь поверхности	Площадь стены	Площадь крыши	Общая высота	Ориентация	Площадь остекления	Распределение площади остекления	Тепловая нагрузка
0,98	514,5	294	110,25	7	2	0	0	15,55
0,9	563,5	318,5	122,5	7	2	0,1	1	29,03
0,86	588	294	147	7	3	0,1	2	27,02
0,79	637	343	147	7	4	0,1	3	36,97
0,76	661,5	416,5	122,5	7	5	0,1	4	32,31
0,69	735	294	220,5	3,5	2	0,1	5	11,21

В репозитории UCI (см. [25.3]) можно найти эти и многие другие наборы данных в удобном для чтения формате. В реальном мире сбор и организация данных для каж-

дой задачи выполняются по-разному: путем анализа различных файлов журналов и/или запросов к базам данных. Такие данные обычно нуждаются в последующей очистке. Также не следует глубоко вчитываться в данные, но лучше просматривать их, прежде чем делать выводы. Главное при этом — не пытаться узнать о данных что-то лишнее!

Данные лучше всего представляются интерфейсом доступа типа модель-представление-контроллер, который позволяет создать конвейер преобразования данных, чтобы избежать чрезмерного копирования:

```
template<typename X, typename Y> struct InMemoryData
{
    Vector<pair<X, Y> > data;
    typedef X X_TYPE;
    typedef Y Y_TYPE;
    typedef X const& X_RET;
    InMemoryData() {}
    template<typename DATA> InMemoryData(DATA const& theData):
        data(theData.getSize())
    {
        for(int i = 0; i < data.getSize(); ++i)
        {
            data[i].first = theData.getX(i);
            data[i].second = theData.getY(i);
        }
    }
    void addZ(X const& x, Y const& y){data.append(make_pair(x, y));}
    X_RET getX(int i) const
    {
        assert(i >= 0 && i < data.getSize());
        return data[i].first;
    }
    double getX(int i, int feature) const
    {
        assert(i >= 0 && i < data.getSize() && feature >= 0 &&
            feature < data[i].first.getSize());
        return data[i].first[feature];
    }
    Y_TYPE const& getY(int i) const
    {
        assert(i >= 0 && i < data.getSize());
        return data[i].second;
    }
    int getSize() const{return data.getSize();}
};
```

Для векторных данных необходимо иметь возможность получить измерение, которое работает с любым средством доступа к данным:

```
template<typename DATA> int getD(DATA const& data)
{
    assert(data.getSize() > 0);
    return data.getX(0).getSize();
}
```



Рис. 25.3. Типичный конвейер обработки данных (конкретные преобразования рассмотрим позже)

Иногда конвейер может углубляться, что делает доступ неэффективным (рис. 25.3).

Таким образом, перед применением  $A$  данные можно буферизовать. Буфер наиболее оправдан для  $A$ , которые уже используют много памяти, поэтому дополнительные данные из-за буфера не влияют на варианты использования:

```

template<typename LEARNER, typename BUFFER, typename PARAMS =
    EMPTY> class BufferLearner
{
    LEARNER model;
public:
    template<typename DATA> BufferLearner(DATA const& data,
        PARAMS const& p = PARAMS()): model(data, p) {}
    typename BUFFER::Y_TYPE predict(typename BUFFER::X_TYPE const& x) const
    {return model.predict(x);}
};
  
```

Хотя машинное обучение не позволяет решить любую задачу на свете, но во многих задачах оно очень успешно, в частности (см. [25.34]):

- ◆ распознавание лица — реализовано в большинстве смартфонов. Быстро и достаточно надежно идентифицирует лица;
- ◆ повторная идентификация — восстановление отредактированной личной информации в опубликованных записях. Например, большинство людей можно однозначно идентифицировать по почтовому индексу, дате рождения и полу. Одному губернатору, опубликовавшему анонимные медицинские карты, прислали по почте его собственные данные.

Обучение направлено на открытие новых знаний. Когда известна задача, нужно:

1. Получить данные.
2. Очистить их, если это необходимо.
3. Запустить на них  $A$ , чтобы получить  $f$ .
4. Развернуть/использовать  $f$ .

Пункт (4) является наиболее важной частью этого процесса, хотя сначала так не кажется. Окончательная система должна уметь работать во многих условиях и не подвергаться чрезмерному влиянию, скажем, незначительной предвзятости в данных обучения. Необходимо провести проверочные эксперименты для надлежащего тестирования. Также хорошей идеей является выполнение некоторого анализа чувствительности: внесите произвольные или даже нелогичные изменения во входные данные, чтобы увидеть, как система реагирует на изменения, и тем самым узнать больше о ее устойчивости. Один из способов получить такие данные — смешать измерения существующих данных. Ничего из отмеченного не обсуждается в книгах по машинному обучению, но, как и в случае со всем программным обеспечением, необходимо тщательное развертывание.

## 25.4. Прогнозирование и структурный вывод

И статистика, и машинное обучение пытаются что-то оценить. Возможно, основное их различие состоит в том, что в первом случае упор делается на изучение параметров модели, а во втором — на получение модели с хорошими прогностическими характеристиками.

В некоторых случаях, таких как линейная регрессия, значения параметров выводятся из обычных статистических соображений — типа максимального правдоподобия. Но затем модель используется для прогнозирования. Таким образом, прогностический вывод в каком-то смысле более общий, чем статистический: минимизация риска — это четко определенная задача, в отличие от максимальной вероятности, которая требует выполнения некоторых условий. В принципе, можно напрямую найти параметры путем минимизации риска, и многие теории машинного обучения исследуют, как этот способ будет работать.

## 25.5. Оценка риска предиктора

Пусть  $F$  — это пространство функций, отображающих  $X$  в идентификаторы, зависящие от задачи, определяющее все возможные  $f$ .  $\forall f \in F$ ,  $R_f$  хорошо определена, но неизвестна, т. к. неизвестен  $P$ . Поскольку единственной информацией является случайная выборка  $S$ , в лучшем случае можно узнать, что  $\exists$  ошибка  $\varepsilon$  и вероятность  $p$  такие, что с вероятностью  $\geq 1 - p$  происходит одно из следующего:

- ◆  $R_f \leq \varepsilon$ :  $f$ , вероятно, приблизительно верна ( $PAC(\varepsilon, p)$ );
- ◆  $R_f \in [\varepsilon_-, \varepsilon_+]$ : получаем двустороннюю уверенность.

Если такая граница известна, а  $\varepsilon$  достаточно мало, при  $1 - p \approx 95\%$  обучение прошло успешно. Нельзя гарантировать  $p = \varepsilon = 0$ , если  $X$  не конечно, и вместо  $S$  рассматриваются  $Z$ , а обучение становится не нужно, потому что достаточно просто запоминания. Поэтому нужно быть осторожным при развертывании  $f$  как части критической системы. Например, вы не можете знать, что будет делать обученный беспилотный автомобиль, если перед ним вдруг выпрыгнет животное, а сзади окажется мотоцикл, — ведь такой случай точно не был частью обучения. Во многих случаях требуется мнение человека. Например, устаревший знак дорожных работ может привести к тому, что беспилотный автомобиль снизит скорость и смутит водителей, едущих позади него. Еще одной проблемой могут быть конфликтующие дорожные знаки, когда один из них плохо виден.

Вывод общего из частного называется *индукцией*, которая не так хорошо понята, как *дедукция*, т. е. вывод частного из общего. Проблема в том, что истинное частное не подразумевает истинное общее, т. е. обратный принцип не работает. Иногда возникают редкие и непредсказуемые события — например, сбой сети из-за злонамеренной активности. Итак,  $\exists$  истинное правильной  $f$ , существует лишь достаточно хорошее приближение. В квантово-механическом мире почти все явления природы содержат некоторый шум. Неспособность получить риск, равный 0 или почти 0 (например,  $10^{-4}$ ), является проблемой для многих задач, потому что во многих случаях хорошего значения риска, такого как 1% (при условии классификации), недостаточно.

Для набора примеров *эмпирический риск*  $R_{f,n} = \frac{1}{n} \sum L_f(z_i)$ . Для его вычисления требуется  $O(n)$  вызовов  $f$ :

```
template<typename Y, typename DATA, typename LEARNER> Vector<pair<Y, Y> >
    evaluateLearner(LEARNER const& l, DATA const& test)
{
    Vector<pair<Y, Y> > result;
    for(int i = 0; i < test.getSize(); ++i)
        result.append(pair<Y, Y>(test.getY(i), l.predict(test.getX(i))));
    return result;
}
```

Когда  $z_i$  — независимые случайные данные,  $L_f(z_i)$  — это независимые случайные функционалы с некоторым неизвестным распределением. Вы можете использовать CLT для получения доверительных границ. Результат действителен только асимптотически и не дает правильной границы РАС, но на практике используется часто. Если  $L \in [0, 1]$ , используйте метод Хеффдинга или эмпирический метод Бернштейна, чтобы получить границы РАС.

Если  $f$  не найдется случайным образом после просмотра  $S$ ,  $L_f(z_i)$  не является независимой и случайной, а результирующие оценки чрезмерно оптимистичны из-за многократного тестирования. Если некоторая  $f$  хорошо изучает  $S$ , но плохо справляется с другими примерами, это называется *переобучением*. Например, имея  $n$  телефонных номеров и кредитных карт, можно подобрать многочлен, который определяет номер кредитной карты по номеру телефона и работает для любой  $z_i$ . На любом конечном наборе данных существует несчетное множество  $f$  с нулевым риском. Переобучение случается и с людьми — например, кандидаты на собеседовании часто пишут код, который работает только для заданных примеров. Но важно получить именно обобщение, потому что  $f$  будет использоваться для новых, ранее неизвестных данных.

Одним из решений является *метод удержания*: зарезервировать часть данных для оценки риска и получить оценку и ограничение РАС из нее, используя неравенство Хеффдинга или CLT (для асимптотической оценки). Обычно 80% данных задействуются для обучения, а остальные 20% — для тестирования. Реализация использует перестановку для индексации данных и детерминированной перестановки, чтобы нарушить любой возможный порядок сортировки. Например, можно получить данные из SQL-запроса, отсортированного по какому-то ключу базы данных:

```
template<typename DATA> struct PermutedData
{
    DATA const& data;
    Vector<int> permutation;
    typedef typename DATA::X_TYPE X_TYPE;
    typedef typename DATA::Y_TYPE Y_TYPE;
    typedef typename DATA::X_RET X_RET;
    PermutedData(DATA const& theData): data(theData) {}
    int getSize()const{return permutation.getSize();}
    void addIndex(int i){permutation.append(i);}
    void checkI(int i)const
    {
```



```

        assert(i >= 0 && i < permutation.getSize() &&
               permutation[i] >= 0 && permutation[i] < data.getSize());
    }
    X_RET getX(int i) const
    {
        checkI(i);
        return data.getX(permutation[i]);
    }
    double getX(int i, int feature) const
    {
        checkI(i);
        return data.getX(permutation[i], feature);
    }
    Y_TYPE const& getY(int i) const
    {
        checkI(i);
        return data.getY(permutation[i]);
    }
};

template<typename DATA> pair<PermutedData<DATA>, PermutedData<DATA> >
createTrainingTestSetsDetPerm(DATA const& data,
double relativeTestSize = 0.8)
{
    int n = data.getSize(), m = n * relativeTestSize;
    assert(m > 0 && m < n);
    pair<PermutedData<DATA>, PermutedData<DATA> > result(data, data);
    Vector<int> perm(n);
    for(int i = 0; i < n; ++i) perm[i] = i;
    permuteDeterministically(perm.getArray(), n);
    for(int i = 0; i < n; ++i)
    {
        if(i < m) result.first.addIndex(perm[i]);
        else result.second.addIndex(perm[i]);
    }
    return result;
}

```

Любая незначительная корректировка математической модели может сделать недействительными ее предположения и весь последующий анализ. Это особенно проблематично для машинного обучения, где из-за невозможности анализа реальных моделей используются упрощенные модели. Незначительные изменения вряд ли сильно повлияют на выводы, но строгая валидность исчезнет. Нарушение допущения о независимости и случайности, вероятно, является наиболее распространенной проблемой, поскольку данные почти всегда не являются полностью независимыми. Но на эвристику можно полагаться, когда в противном случае не получается вообще ничего путного. Во многих случаях неформально предполагается какая-то устойчивость, т. е. что результаты при слегка нарушенных допущениях отличаются в лучшем случае незначительно.

## 25.6. Источники риска

Выбор  $L$  позволяет определить *оптимальный механизм обучения Байеса*, который знает  $P$ , и для любого  $x$  выбирает  $ID = \arg\max_z (\Pr(y(ID)|x))$ . Например, для предсказания необъективного подбрасывания монеты с 60%-ной вероятностью выпадения орла и  $L = (\text{ошибка} : 1 : 0)$ ,  $R_{\text{оВ}} = 0,4$ . Это решение является оптимальным, поэтому  $R_{\text{оВ}} = E_z[\min_f L_f(z)]$ . Поскольку  $P$  обычно неизвестно,  $\text{oВ}$  полезно только теоретически:

- ◆ для любой  $f$   $R_{\text{оВ}} \leq R_f$ ;
- ◆ в искусственных случаях, когда значение  $P$  известно, оно дает представление о поведении других  $A$ .

Пусть  $\min_f L_f(z) = 0$ . Тогда для непересекающихся задач, где каждому соответствует один  $ID$ ,  $P|x$  помещает всю вероятность в одну точку или на дискретном интервале, а  $R_{\text{оВ}} = 0$ . В задачах с пересечением есть случайная составляющая, поэтому выбранные признаки не дают достаточно информации для того, чтобы всегда принимать правильные решения. Нужно больше информативных признаков.

Общая стратегия поиска хорошего  $A$  состоит в том, чтобы свести поиск  $f$  к оптимизации, т. е. оптимизировать по некоторому *пространству поиска* или *классу моделей*  $G \subseteq F$ . Обычно  $G$  намного меньше, чем  $F$ , потому что  $F$  несчетно, но можно рассматривать только  $G = \{f, \text{ для представления которых не требуется слишком много памяти}\}$ . Ограничение точности чисел (рис. 25.4) в таком случае не проблема, поскольку  $f$ , отличающиеся только в пределах точности, статистически неразличимы.  $A = G +$  стратегия выбора  $f \in G$  путем минимизации  $R_f$  на основе  $S$ .

$R_f \leq$  суммы:

- ◆ *ошибки информативности признака*  $R_{\text{оВ}} - 0$ . Из-за отсутствия информативных функций;
- ◆ *ошибка приближения (аппроксимации)*  $\min_{h \in G} R_h - R_{\text{оВ}}$ . Из-за того, что  $G$  не используется с лучшим  $h$ ;

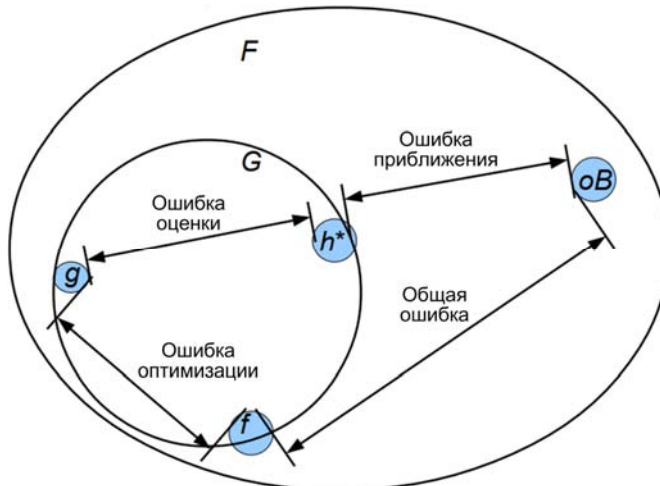


Рис. 25.4. Ошибки, вызванные неизбежными ограничениями

- ♦ *ошибка оценки*  $R_g - \min_{h \in G} R_h$ , для  $g = \operatorname{argmin}_{h \in G} \text{SearchObjective}(h, n)$ . Из-за незнания  $Z$ ;
- ♦ *ошибка оптимизации*  $R_f - R_g$ , для  $f = \operatorname{приблизительно} \operatorname{argmin}_{h \in G} \text{SearchObjective}(h, n)$ . Из-за приближительной оптимизации (точная оптимизация обычно сложна).

Предполагая, что идеальные функции и ошибка оптимизации = 0, нас интересуют ошибки приближения (аппроксимации) и оценки. Последнее может быть ограничено через  $n$ , т. е.  $\forall f \in G \left| R_f - R_{f,n} \right| \leq B(n, f, p)$  с вероятностью  $\geq 1 - p$  такой, что  $\forall \varepsilon > 0$ , и  $p > 0 \exists n(\varepsilon, f, p)$  такие, что  $B \leq \varepsilon$ ;  $B$  не зависит от  $P$ . Если  $B$  одинаково для любых  $f$ ,  $G$  имеет *равномерную сходимость* и  $B(n, f, p) = B(n, C(G), p)$ , где  $C$  измеряет *способность*  $G$  к переобучению. Границы конечной выборки выполняются одновременно для любой  $f \in G$ , в частности для любой  $f$ , возвращаемой  $A$ . Ограничение емкости предотвращает переобучение, исключая  $f$ , способные точно моделировать шум в  $S$ , как в примере с кредитной картой.

Эвристика  $C$  — это количество оцениваемых параметров. Она не дает общие границы, но обычно близка к  $C$ , которая это делает. Например, пример с кредитной картой не работает, если  $S =$  многочлены степени  $< n$ . Хёффдинг дает равномерную сходимость для одноэлементных  $S = \{f\}$  в задачах с ограниченными  $L$ .

*Принцип Ватника* предлагает решить более простую задачу напрямую — вместо того, чтобы сначала решить более сложную промежуточную. Идея состоит в том, чтобы минимизировать ошибку оценки.

## 25.7. Контроль сложности

Использование  $R_{f,n}$  в качестве цели поиска называется *минимизацией эмпирического риска* (Empirical Risk Minimization, ERM). Если  $G$  имеет равномерную сходимость,  $B$  одинаково для любых  $f \in G$ , и найденный вариант будет иметь конечно-выборочную оценку. ERM полезна только для нескольких особых случаев, таких как линейная регрессия, поскольку в этом случае для получения малого  $B$  нужно малое  $C$  или очень большое  $n$ . Равномерная сходимость определяет достаточную *сложность выборки*  $S$ , т. е. для любого  $\varepsilon > 0$  и  $p > 0$  может найти такое  $n$ , что ошибка оценивания  $< \varepsilon$ . Чтобы обеспечить эффективное обучение, необходимо  $n = O(\text{полиномиальное})$ . Но обычно  $G$  неизвестна, поэтому использование сложности выборки по существу ограничивается линейной регрессией.

Можно построить  $G$  с оценкой конечной выборки. Рассмотрим  $G = \cup H_i$  и соответствующую  $w_i \in (0, 1)$ , выбранную до просмотра данных, так что  $\sum w_i \leq 1$ , и для любого  $H_i$  получаем равномерную сходимость с оценкой  $B$ . Из-за многократного тестирования условие  $R_f \leq B$  не выполняется, но по поправке Бонферрони  $\forall f \in S \left| R_f - R_{f,n} \right| \leq B(n, C(H_i(h)), w_i(f) p)$  с вероятностью  $\geq 1 - p$  (см. [25.29]).

Если у  $G$  есть границы конечной выборки, задача, где целью поиска является  $R_{f,n} + B(n, f, p)$ , называется *минимизацией структурного риска* (Structural Risk Minimization, SRM). Обычно она начинается с рассмотрения более простых моделей и останавливается, когда для наилучшего найденного  $f$  и всех  $h$  еще не рассмотренных

$R_{f,n} + B(n, f, p) \leq B(n, h, p)$ . Пусть  $g = \min_{h \in S} R_h$ . Поскольку поиск выбрал  $f$ , а не  $g$ ,  $R_{f,n} + B(n, f, p) \leq R_{g,n} + B(n, g, p)$  и, следовательно,  $R_{f,n} - R_{g,n} \leq B(n, g, p) - B(n, f, p)$ . Тогда ошибка оценки:

$$\begin{aligned} R_f - R_g &\leq (R_{f,n} + B(n, f, p)) - (R_{g,n} - B(n, g, p)) = \\ &= R_{f,n} - R_{g,n} + B(n, f, p) + B(n, g, p) \leq 2B(n, g, p), \end{aligned}$$

что является оценкой конечной выборки.

Из доказательства видно, что нужна двусторонняя граница, потому что, если есть только верхняя граница,  $R_{h,n}$  может быть отделена от  $R_h$  и не давать информации о  $g$ . В SRM различие между поиском структуры модели и оптимизацией ее параметров теряется. Даже если известно, что истинная модель сложна, из-за отсутствия достаточного количества данных для ее оценки придется согласиться на более простую модель.

Значения  $w_i$  содержит априорную информацию, но не являются значимыми. Например,

для  $H_i$  в порядке возрастания  $C(H_i(h))$  можно использовать  $w_i = \frac{6}{\pi(i+1)^2}$ , которое мед-

ленно уменьшается и в сумме дает 1 в пределе. Для получения конечного ответа можно использовать любое  $p$  вроде 0,05 или 0,01 и прекратить поиск после того, как границы воспрепятствуют существованию лучшего  $f$  в  $H_i$  с более высокой пропускной способностью. Перед просмотром данных необходимо выбрать  $p$ ,  $S$  и  $w_i$ . Но обычно выбирают не все значения, и в этом случае результат является только эвристическим.

Предположим, что  $H_i$  состоит из одной гипотезы  $h$  и  $L \in [0, M]$ . Подставим  $w_i p$  в двустороннее неравенство Хёффдинга. Тогда получаем

$$|R_f - R_{f,n}| \leq M \sqrt{\frac{\ln(1/w_i) + \ln(2/p)}{2n}}$$

с вероятностью  $\geq 1 - p$ . Так можно выполнить SRM по любой счетной гипотезе после присвоения весов. В частности, пусть каждое  $h$  представлено в двоичном виде с помощью некоторого универсального кода без префиксов, такого как гамма (см. главу 15. Сжатие), используя размер  $|h|$ . Тогда для  $w_i = 2^{-|h|}$ , по неравенству Крафта  $\sum w_i \leq 1$ . Это приводит к минимизации риска Оккама (Occam Risk Inimization, ORM), т. е. поиска:

$$f = \arg \min_{h \in G} R_{h,n} + M \sqrt{\frac{|h| \ln(2) + \ln(2/p)}{2n}}.$$

Для кодирования вещественного числа используйте «достаточно хорошую» точность  $O(1/\sqrt{n})$ , основанную на типичной дисперсии многих оценок, предсказанных границей Крамера — Рао (см. главу 21. Вычислительная статистика). Последнее более разумно, потому что даже в лучшем случае получения среднего значения существует оценка с примерно такой же точностью. Чем экономичнее код, тем лучше оценка.

Незначительная проблема ORM заключается в том, что масштабирование  $y$  для изменения  $M$  не сильно влияет на код модели. Необходимость определить значение  $G$  перед просмотром данных не является проблемой, потому что даже определение структуры данных, способной представлять  $f$ , определяет код без префиксов. Метод ORM очень общий и применяется к любой задаче с ограниченным  $L$ , поэтому обычно легко найти и

определить подходящую  $G$ . Но использование информации, специфичной для задачи, может дать гораздо лучшие оценки для достаточно сложных  $h$  — нижние границы ORM (обычно очень расплывчатые) не препятствуют этому. Так что на практике значение метод ORM заключается в том, чтобы показать недостатки дизайна, препятствующие обобщению. Например, для лучшего обобщения не надо задавать ненужную точность параметров.

Философское понятие *бритва Оккама* советует не делать вещи более сложными, чем это необходимо, т. е. простое объяснение с большей вероятностью будет правильным, чем сложное, и ORM следует этому правилу. Выгоды:

- ◆ простота полезна сама по себе — например, простую модель проще понять и изучить;
- ◆ существуют гораздо более сложные модели, поэтому вероятность случайного нахождения сложной модели, которая соответствует имеющимся данным, выше. То есть выбор простого  $G$  ограничивает переобучение, уменьшая «количество»  $f$ , которые можно попробовать. В качестве альтернативы, имея данные и две модели, которые хорошо подходят к ним, можно выбрать более сложную модель с точки зрения выбора параметров.

## 25.8. Ошибка аппроксимации

$G$  не всегда способно вычислить разбиение  $oB$ , которое для конкретной задачи может быть *сколь угодно сложным* по Колмогорову, что определяется невычислимым наименьшим числом битов, которое может его представить (см. главу 15. *Сжатие*), или сложным по некоторой эвристической метрике. Таким образом, для указания раздела может потребоваться произвольное количество примеров. Возможны определенные сложности:

- ◆ некоторые  $z_i \in S$  могут оказаться *выпадающими*, когда трудно отличить шум подсказок (некоторые  $y_i$  неверны) от достоверной информации. Сложные модели должны уметь обрабатывать локальные исправления  $X$  с различным поведением, особенно если такие исправления поддерживаются достаточным количеством примеров. Шум мешает анализировать близлежащие примеры с разными подсказками. Обычно при наличии достаточной *поддержки примеров* предполагается, что пятно локально отличается, в противном случае это выпадающее значение. Шум наиболее проблематичен в небольших локальных пятнах, чем в более крупных, потому что большие пятна сглаживают шум (рис. 25.5);
- ◆ *проклятие размерности* — у данных в векторном пространстве обучение усложняется по мере увеличения  $D$ . Если  $x$  является вектором двоичных переменных, все двумерные возможности равновероятны, и даже получение достаточного количества данных для оценки является проблемой, поскольку конкретный  $x$  ничего не говорит о других, например, для многомерной операции  $\text{xor}$ . В более общем смысле функция может состоять из множества локальных областей, каждая из которых ведет себя по-своему, и их количество обычно увеличивается с ростом  $D$ . Возможно, лучшим объяснением того, почему теоретические границы не создают проблем, является *гипотеза коллектора*, согласно которой в практических наборах данных  $Z$ -часть с достаточной поддержкой намного меньше  $Z$ . Это помогает контролировать ошибку оцен-

ки, сосредотачивая данные на важных областях и игнорируя маловероятные, и этого зачастую достаточно для достижения удовлетворительной производительности.

В целом безнадежно пытаться найти точное лучшее разбиение, по крайней мере лучшее, чем асимптотическое. В лучшем случае его можно аппроксимировать, и чем выразительнее  $G$ , тем меньше ошибка аппроксимации. Поэтому обычно выбирают наиболее выразительные  $G$ , где имеет место удовлетворительный контроль ошибки оценки. Поскольку любые обучающие данные содержатся в гиперсфере конечного радиуса, прогнозирование чего-либо вне ее не имеет смысла. Но вероятность встретить такой  $x$  экспоненциально мала по  $n$ .

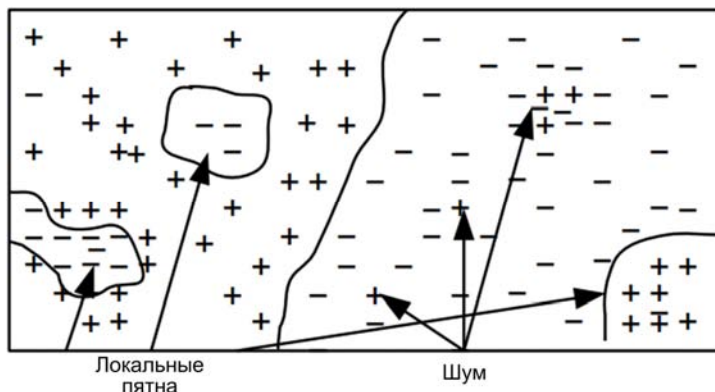


Рис. 25.5. Различные типы примеров

Не существует конечно-выборочной оценки ошибки аппроксимации, по крайней мере, без некоторых предположений — например, предположения об ограниченном диапазоне или о том, что разбиение имеет липшицевую непрерывность или другую регулярность (т. е. подобные  $x$  приводят к подобным  $y$ ). Например, существует фигура, которая больше всего напоминает и  $1$ , и  $7$ , но на практике предполагается, что такая форма вряд ли появится. Человек сказал бы: «Не знаю, может быть и  $1$ , а может и  $7$ ». Кроме того, в таких играх, как шахматы, гроссмейстеры обладают интуицией только в отношении небольшого подмножества всех возможных позиций. Сложность разделения является основной причиной ошибки аппроксимации. Один из способов измерить его для той или иной задачи — посмотреть, насколько сложна логика разбиения конкретного  $A$  с достаточно низким уровнем риска.

$A$  является согласованным для задачи, если  $R_{\mathcal{L}}(A, n) \rightarrow R_{\text{ов}}$  при  $n \rightarrow \infty$ , и универсально согласованным, если это выполняется для любой подзадачи той или иной задачи (см. [25.8]). Это отличается от статистической согласованности, которая требует только сходимости к предельному параметру выбранной модели без учета ошибки ее аппроксимации. Некоторые примечания:

- ◆ многие  $A$  универсально непротиворечивы;
- ◆ для задач, с которыми простые разбиения справляются плохо, универсально согласованное  $A$  требует сколь угодно больших  $n$  и объема памяти для представления  $f$ ;
- ◆ усиление определения за счет оценок конечной выборки приводит к невыполнимо-му запросу, поскольку для некоторых задач требуются произвольно сложные раз-

бления. В общем случае теоремы об *отсутствии бесплатных обедов* (No Free Lunch, NFL) не дают доказательств (см., например, главу 26. *Машинное обучение: классификация*).

Другие (более известные) теоремы NFL для задач прогнозирования исходят из того, что существуют задачи, в которых конечные обучающие данные вредны и бесполезны, и во всех таких задачах ни одно  $A$  не выигрывает. Основная идея доказательства (см. [25.22]) состоит в том, что если обучение является детерминированным, а  $X$  и  $Y$  ограничены, можно установить равномерное распределение для возможных  $P$  так, что каждое  $P$  будет равновероятно. Тогда для любых  $A$  и конечном процессе обучения, если обучающие данные исключены из расчета риска,  $E_P R_f(A, \text{данные}) = E_{x \notin \text{данные}} E_P L(f(x), y)$ , поскольку при равномерном распределении  $P$  при заданном  $x$  все  $y$  равновероятны, и  $E_P L(f(x), y) = E_P L(f(x), \text{равномерно случайное } y)$ . В задаче классификации последнее равно  $1/k$ , т. е. ожидаемому риску случайного предположения.

Так:

- ◆ невозможно выполнить обучение для всех задач сразу;
- ◆ поскольку некоторые  $A$  справляются с некоторыми проблемами лучше, чем другие, они должны хуже справляться с другими проблемами;
- ◆ на практике для обобщения необходимо допущение хотя бы некоторой неявной предвзятости.

Более общие теоремы NFL (см. [25.35]) приводят к тем же выводам. Классификация — это самая простая задача обучения как с точки зрения информации-подсказки, так и с точки зрения диапазона  $y$ . Таким образом, теоремы NFL справедливы и для более общих задач (см. [25.31]). Эта логика должна применяться и к другим выводам о невозможности или вычислительной сложности. NFL не противоречит универсальной непротиворечивости, которая допускает произвольное количество обучающих данных. Самое проблематичное предположение NFL состоит в том, что обучение пытается решить все возможные проблемы. Некоторые интерпретации:

- ◆ бессмысленно сравнивать разные  $A$ , предполагая, что некоторые из них лучше других. Только частные предположения позволяют различать  $A$  из разных задач. Эксперименты говорят, что разные типы задач отдают предпочтение разным обучающимся, поэтому некоторая частичная NFL должна быть верной в том смысле, что допущения конкретного  $A$  являются более подходящими для одних задач, чем для других. Например, линейная функция может лучше работать в одних задачах и хуже в других. Но некоторые  $A$  неизменно лучше справляются с большинством проверенных задач, хотя вполне возможно, что с другими задачами это будет не так;
- ◆ чтобы избежать NFL, нужно иметь знания предметной области в отношении конкретной задачи. Знание предметной области часто очень помогает, например, в задачах обработки изображений и естественного языка. Поскольку необходимо выбрать как минимум класс модели, из которого следует выбирать конкретную модель, правильность этого выбора определяется предметной областью;
- ◆ не ожидайте, что задачи, в которых обучающие данные вредны, будут решены. В этом случае информация-подсказка для  $S$  не связана с информацией о  $z \notin S$ . NFL не работает, если нужно найти решение хоть для какой-то  $P$ , поэтому ее можно иг-

норировать. То есть требуются некоторые предположения, например о хороших предположениях. Например, большинство  $A$  имеют отклонения, предполагая, что  $z$  каким-то образом связаны. Приведенные ранее интерпретации могут иметь место лишь частично.

Конкретный  $A$  не обязательно должен быть согласованным, даже если лежащее в его основе представление знаний допускает согласованность. Если согласованности нет, лучше всего подойдет высокая мощность аппроксимации.

Границы конечной выборки для ошибки оценки могут быть слишком большими при небольшом фиксированном  $n$ , когда множество  $G$  достаточно выразительно, чтобы иметь небольшую ошибку аппроксимации. Например,  $G =$  многочлены степени 1000 имеет конечное число параметров и должно иметь равномерную сходимость для многих задач, но для получения полезных оценок  $n$  должно быть очень большим. Таким образом, SRM и ORM предпочтительнее ERM.

Выбор структуры данных представления знаний обычно определяет  $G$  и предлагает некоторую стратегию поиска. Например, использование гиперплоскости означает, что  $G =$  множество всех гиперплоскостей, а естественный поиск — это оптимизация параметров. *Поскольку требуется выбрать некоторое представление знаний, надо сделать хоть какие-то предположения.*

Никакой принцип индукции не позволяет всегда получать хороший  $A$ . Теоретические границы обучающего множества дают только некоторые указания, а не полезные оценки ошибок. Большинство  $A$  были созданы ad hoc:

1. Выбрать представление знаний, которое не препятствует согласованности и является экономичным с точки зрения простоты и уменьшения ошибок оценки.
2. Некоторым образом контролировать ошибку оценки, либо за счет простоты, устойчивости и отсутствия многократного тестирования, либо за счет сочетания этих вещей.
3. Учитывать другие ограничения — например, алгоритму может потребоваться удобство для человека или масштабируемость, и для достижения таких ограничений понадобится отказаться от некоторого риска.

Ни один из лучших  $A$  в точной реализованной форме не дает строгих границ конечной выборки, а известные границы, за исключением незначительных нарушений, оказываются неопределенными. Если известен хороший  $A$ , неясно, существует ли для проблемы лучший вариант. Например, эллипсы должны быть сложными для человеческой интуиции, потому что потребовались столетия, прежде чем Кеплер открыл эллиптические орбиты. Каждые 10 лет или около того обнаруживается по крайней мере один очень хороший алгоритм. Чаще всего на практике выбор  $A$  является произвольным и не требует тщательной логики.

## 25.9. Устойчивость

Интуитивно устойчивый  $A$  выводит одинаковые  $f$  из разных наборов данных (см. [25.24]). Пусть обучающие наборы  $T_1$  и  $T_2$  с  $n$  независимых случайных примеров отличаются только одним примером.  $A$  равномерно  $b$ -устойчив, если задана  $f$  из обучения на  $T_1$ , и  $h$  на  $T_2$ ,  $\forall z, \forall f, |L_f(z) - L_h(z)| \leq b$ .



Теорема ([25.24]): если  $L \leq M$  и  $A$   $b$ -стабилен, любое  $T$  состоит из  $n$  независимых случайных точек данных, если  $A$ , обученный на  $T$ , производит  $f$ , тогда

$$R_f \leq R_{f,n} + b + (2nb + M) \sqrt{\frac{\ln(1/p)}{2n}} \text{ с вероятностью } \geq 1 - p.$$

Таким образом, если  $b = O(1/n)$ , как в случае с несколькими алгоритмами (некоторые из них обсуждаются в последующих главах), ошибка оценивания составляет  $O(1/\sqrt{n})$ . Это очень сильный результат, учитывая его общность. К сожалению, в настоящее время значение  $b$  известно только для нескольких полезных  $A$ . Тем не менее это подчеркивает важность устойчивости. В частности, нестабильные  $A$  не могут оценить свои параметры с разумной точностью и, как ожидается, будут иметь высокую ошибку оценки.

## 25.10. Оценка риска стратегии обучения

Вы можете оценить риск  $f$ , используя задержку. Но часто требуется оценить риск  $A$  (см. [25.33, 25.9]) по:

1.  $P$  при использовании только определенных  $S$  — для определенных решений, которые применяются на основе данных во время обучения, таких как параметры настройки;
2.  $P$  — для сравнения нескольких  $A$  независимо от  $S$ , но с фиксированным  $n$ , потому что обычно вы обучаете на некотором наборе данных размера  $n$  и после этого используете  $f$ .
3. Несколько  $P$  — для сравнения множества  $A$  в целом и принятия решения о том, какой  $A$  использовать для нового набора данных.

Пункт (3) мы рассмотрим позже, а (1) является частным случаем (2). Для (2) важно понимать, что должно быть независимым, и каковы источники отклонений:

- ◆  $A$  может быть рандомизированным;
- ◆ обучающие данные должны быть независимыми и случайными;
- ◆ тестовые данные должны быть связаны друг с другом и не зависеть от обучающих данных.

В идеале хочется знать  $P$ , и вы можете  $m$  раз выполнить обучение на случайной выборке размером  $n$  и проверять полученное  $f$  на случайно выбранном примере. Это создает средние объективные оценки производительности для статистического вывода. Чтобы сделать так на практике, нужен набор данных размером  $m(n + 1)$ , который не обеспечивает эффективного использования данных, но не может работать лучше, не вводя отклонений. Каждый процесс обучения можно запустить несколько раз со случайным порядком примеров и усреднить результаты теста. Это позволяет уменьшить дисперсию из-за собственной рандомизации  $A$ , и зависимость от порядка примера по-прежнему дает независимую случайную выборку размера  $m$  средних значений партии.

Для пункта (1) нужен дополнительный независимый набор данных размера  $m$ . Затем нужно  $m$  раз обучить  $A$  на исходных данных в случайном порядке и оценить результат на одном примере, получив  $m$  независимых случайных несмещенных оценок. Это похоже на введение задержки — за исключением того, что обучение выполняется заново для любого тестового примера. Если  $A$  не имеет собственной рандомизации и не зави-

сит от порядка примера, оба набора дадут один и тот же результат, при этом задержка будет существенно более эффективной и, в нашем случае, предпочтительным выбором.

*Перекрестная проверка:* разбить данные на  $k$  равных подмножеств,  $k$  раз обучить  $A$  на  $k - 1$  подмножествах и протестировать оставшееся подмножество, возвращая средний риск в качестве оценки производительности:

Этап	Использование данных				
1	Обучение	Обучение	Обучение	Обучение	Проверка
2	Обучение	Обучение	Обучение	Проверка	Обучение
3	Обучение	Обучение	Проверка	Обучение	Обучение
4	Обучение	Проверка	Обучение	Обучение	Обучение
5	Проверка	Обучение	Обучение	Обучение	Обучение

Затем  $f$  обучается на всех данных. Таким образом, каждый пример проверяется один раз (с некоторым округлением, когда  $k$  не является множителем  $n$ , но для обычных малых  $k$  это значения не имеет). Чаще всего значение  $k = 10$  дает наилучшую оценку, но значения 5 или 20 столь же хороши (см. [25.19]). Наилучшее значение зависит от приложения (см. [25.2]), здесь для эффективности по умолчанию  $k = 5$ :

```
template<typename LEARNER, typename Y, typename DATA, typename PARAM>
Vector<pair<Y, Y> > crossValidateGeneral(PARAM const& p,
DATA const& data, int nFolds = 5)
{
    assert(nFolds > 1 && nFolds <= data.getSize());
    Vector<pair<Y, Y> > result;
    int testSize = data.getSize()/nFolds; // отброшенные округлением
                                         // данные используются для обучения
    for(int i = 0; i < nFolds; ++i)
    {
        PermutedData<DATA> trainData(data), testData(data);
        int testStart = i * testSize, testStop = (i + 1) * testSize;
        for(int j = 0; j < data.getSize(); ++j)
        {
            if(testStart <= j && j < testStop) testData.addIndex(j);
            else trainData.addIndex(j);
        }
        LEARNER l(trainData, p);
        result.appendVector(evaluateLearner<Y>(l, testData));
    }
    return result;
}
```

Понимание многих аспектов перекрестной проверки является открытой проблемой из-за отсутствия независимости:

- ♦ обучающие наборы перекрываются в 60 или 80% данных, в зависимости от  $k$ ;
- ♦ если любые два этапа содержат очень разные данные с точки зрения  $A$ , во всех прогонах обучающие и тестовые данные могут сильно различаться.

Другие и сопутствующие проблемы:

- ◆  $k$  повторений может быть недостаточно для устранения дисперсии из-за случайности в  $A$ ;
- ◆ оценка дана для  $A$ , обученном на  $n(k - 1)/k$  примерах, и обычно пессимистична для  $A$ , обученного на  $n$  примерах, потому что  $A$  работает лучше при увеличении числа данных;
- ◆ не существует объективной оценки дисперсии оценки риска перекрестной проверки (см. [25.4]). Кроме того, хотя это и не доказано, вполне вероятно, что полезной верхней границы для этой оценки не существует.

Некоторые способы улучшить перекрестную проверку:

- ◆ стратификация по  $y$  — выполняется просто для дискретных  $Y$ . Она дает более точные оценки, где это применимо (см. [25.19]);
- ◆ детерминированное скремблирование примерного порядка с помощью генератора случайных чисел с фиксированным начальным числом обычно делает разные свертки более похожими, например когда данные сортируются по какому-либо свойству, и этого, вероятно, достаточно, когда стратификация неприменима;
- ◆ повторяйте перекрестную проверку 10 или 20 раз, выполняя случайные перестановки данных, усредняйте тестовые баллы каждого экземпляра и используйте средние значения партии в качестве образцов для эвристического расчета дисперсии. Повторение устранил дисперсию из-за случайности  $A$  и, что более важно, позволит уйти от выбора конкретных данных из набора:

```
template<typename PARAM, typename Y, typename DATA, typename LEARNER>
Vector<pair<Y, Y> > repeatedCVGeneral(PARAM const& p, DATA const& data,
    int nFolds = 5, int nRepeats = 5)
{
    Vector<pair<Y, Y> > result;
    PermutedData<DATA> pData(data);
    for(int i = 0; i < data.getSize(); ++i) pData.addIndex(i);
    for(int i = 0; i < nRepeats; ++i)
    {
        GlobalRNG().randomPermutation(pData.permutation.getArray(),
            data.getSize());
        result.appendVector(crossValidateGeneral<PARAM, Y>(p, data, nFolds));
    }
    return result;
}
```

- ◆ можно рассмотреть возможность переключения на задержку, когда  $n$  велико (возможно,  $> 10^6$ ). При большом  $n$ , если  $A$  достаточно устойчив, результаты будут аналогичными, но вы сэкономите немало времени. Перед переключением можно постепенно уменьшать  $k$  до 2.

Для случая (1) перекрестная проверка и ее улучшенные версии дают почти независимую случайную выборку, которая эвристически считается таковой. Для повышения эффективности обычно используют стратификацию или детерминированное скремблирование, хотя повторение, вероятно, окажется более точным. Также усреднение в повторении может оказаться неудобным для вычисления некоторых других метрик.

Поэтому повторение выполняется только в том случае, если вам нужна более точная оценка или известно, что  $A$  рандомизирован.

Вы можете получить необъективную оценку даже неочевидными способами. Например, если  $f$  является лучшей в тестовом наборе, ее оценка на том же тестовом наборе слишком оптимистична, потому что часть его наилучшей производительности является случайной и предпочтительной конкретно для этого тестового набора. Даже при проверке (обсуждаемой далее в этой главе) переобучение может привести к выбору субоптимальных параметров. Функция  $f$ , обученная с использованием  $A$  с заданным предполагаемым риском, может не иметь такого же риска. Если  $A$  имеет большую дисперсию,  $R_f$  будет соответственным образом изменяться. Точная оценка  $R_f(A, n)$  — это все, на что можно надеяться. Для онлайн-обучения нужно несколько разных алгоритмов.

Для случая (3):

- ◆ в NFL сравнения могут быть не вполне корректны, поскольку лучший в целом  $A$  не обязательно должен быть лучшим для любого набора данных. Конкретные  $A$  лучше всего справляются с определенными типами задач, особенно если имеются полезные знания предметной области. Например, задачи распознавания цифр и распознавания цветка ириса очень различаются. Использование знания предметной области часто позволяет получить лучшие результаты. К примеру, в задаче распознавания цифр можно попробовать поворачивать изображения и извлечь из этого дополнительные полезные данные;
- ◆ нужно выбрать некоторую метрику и рассчитать ее для любой задачи и  $A$ . Метрикой обычно является  $R_f$ ,  $n$ , теоретическая верхняя граница или что-то еще. Если  $A$  рандомизирован, запустите его несколько раз и возьмите среднее значение;
- ◆ поскольку разные задачи могут иметь разную сложность, чтобы сделать их сопоставимыми, нужно преобразовать метрики в кривые оценки или ранги (см. главу 21. *Вычислительная статистика*). Когда метрикой является риск, лучше всего подходят криволинейные оценки, поскольку  $R \geq 0$ . Это предполагает, что преобразованные оценки являются независимыми и случайными и не имеют отклонений;
- ◆ для любого  $A$  вычислить среднюю преобразованную метрику и использовать ее для сравнения.

Это можно сделать для нескольких метрик, таких как риск и затраты ресурсов, с помощью кривых или рангового преобразования. Но для метрик вроде времени выполнения ранговое преобразование, скорее всего, является единственным разумным решением, потому что самый медленный  $A$  может быть намного медленнее, чем самый быстрый, и масштабирование не имеет большого смысла. Значения  $D$  и  $n$  тоже играют серьезную роль, но непонятно, как их учитывать при масштабировании, поскольку время выполнения на разных задачах различается неслучайным образом.

Малые наборы данных столь же важны, как и большие. Выбор набора данных также имеет значение, потому что легко найти наборы данных, в которых конкретный  $A$  работает хорошо. Во избежание смещения лучше всего использовать разные наборы данных, желательно из разных областей и разной степени сложности. Кроме того, для статистической значимости тоже необходимо достаточное количество наборов данных.

Для сравнения независимых выборок по (1), (2) и (3) можно использовать следующие метрики (см. главу 21. *Вычислительная статистика*):

- ◆ проверка гипотез:
  - знаковый тест для двух  $A$ ;
  - критерий Немени, если алгоритмов много, т. к. он может сравнивать все пары, включая лучший  $A$ , с остальными;
- ◆ начальная загрузка — эффективна для многих типов сравнений, в частности для нахождения процента хорошей производительности для  $A$ . Она также более гибкая и позволяет, к примеру, сравнивать несколько метрик одновременно желаемым способом.

При оценке  $f$  и  $A$  с использованием хорошо известных наборов данных, таких как данные UCI, имейте в виду, что многие  $A$  были созданы именно под эти алгоритмы. Вероятно, что было опробовано множество алгоритмов и конфигураций параметров, а неудачные были отброшены, что привело к возникновению смещения. Разумным, но несовершенным решением является использование большего количества наборов данных, особенно новых. В то же время трудно правильно обновлять существующие исследования новыми наборами данных или  $A$ , потому что обычно проводится некоторое тестирование.

## 25.11. Принятие решений и выбор модели

Для  $A$ , у которых есть параметры, часть обучающих данных резервируют для *проверки*, чтобы правильно оптимизировать эти параметры. *Параметром* называют все, что ограничивает рассматриваемый набор предикторов — например, выбор стратегии. В статистике *модель* — это набор предикторов. Например, линейная модель — это набор всех гиперплоскостей, которая соответствует данным для создания *предиктора*. Не путайте это с простым случаем, где модель аналогична предиктору (хотя во многих источниках эти понятия являются синонимами). *Класс моделей* — это набор схожих моделей (как, к примеру, набор полиномов  $\leq$  степени 3). Такие модели можно также называть *подмоделями*.

Идея оптимизации параметров аналогична оценке риска — для хорошего обобщения оптимизация параметров должна учитывать разные данные. Простейшей эффективной стратегией оптимизации нескольких параметров является экспоненциальный поиск по сетке диапазонов (см. главу 24. *Численная оптимизация*) с 5-кратной оценкой риска перекрестной проверки. Перекрестная проверка особенно удобна, поскольку ошибки в оценках различий у разных  $A$  обычно намного меньше, чем ошибки в оценках производительности (см. [25.19]). Выбор модели выполнить проще, чем оценку производительности, поскольку первое можно свести ко второму. Можно также использовать результаты  $n$  тестов, принять их независимыми и случайными и вычислять их дисперсию, после чего выбрать простейший  $A$  в пределах одного стандартного отклонения от наилучшего  $A$ . Впрочем, точность такого расчета является эвристической. Для этого правила и перекрестной проверки стандартная ошибка рассчитывается на основе выборки размером  $k$ , т. е. при допущении независимых этапов (см. [25.17]). После выбора параметров нужно повторно выполнить обучение на всех данных.

Выбор параметров не влияет на границы пропускной способности конечной выборки, но из-за многократного тестирования обычно возникает ошибка оценки. Таким образом, для параметров, к которым  $A$  не очень чувствителен, лучше иметь разумные зна-

чения по умолчанию, что также позволит сэкономить время выполнения. По возможности при выборе структуры и параметров используйте логику (основанную на каких-то имеющихся знаниях), а не перебирайте множество вариантов. Например, при выборе из диапазона параметров, который подходит для большинства областей, используйте поиск по сетке. Как правило, такие диапазоны делаются немного шире, чем необходимо, потому что плохой выбор значения для некоторой задачи хуже, чем незначительная ошибка оценки и неэффективность в большинстве задач. По возможности для определенных областей используйте настраиваемые диапазоны.

Ограничение оптимизации позволяет уменьшить ошибки оценки. Контроль сложности и предотвращение многократного тестирования являются основными инструментами контроля ошибок оценки. В случае обучения с учителем используйте более прямые методы, которые обсуждаются в последующих главах.

Чтобы получить точную оценку риска после выбора параметра, необходимо снова оценить выбор на новых данных, используемых только при вычислении окончательного предиктора. Тем не менее, несмотря на то, что оценка риска выбранной конфигурации параметров на выбранных данных имеет смещение, различия между кандидатами должны быть намного меньше, так что сам выбор должен оказаться достоверным.

Можно обобщить эту методику до *вложенного выбора модели*, который применяется при выборе нескольких наборов параметров. Например, можно использовать перекрестную проверку для выбора модели, но  $A$  может выполнить свою собственную перекрестную проверку на имеющихся данных и подобрать себе параметры. Вложенный выбор, как правило, реже приводит к переобучению, чем выбор из многомерного набора, но требует большего количества данных.

Иногда алгоритму не требуется никаких параметров, поэтому приведенный далее пример работает со стандартным API, который предполагает интерфейс с одним параметром (возможно, сложный объект):

```
template<typename LEARNER, typename Y, typename X = NUMERIC_X>
struct NoParamsLearner
{
    LEARNER model;
    template<typename DATA> NoParamsLearner(DATA const& data,
        EMPTY const& p): model(data) {}
    Y predict(X const& x) const { return model.predict(x); }
    double evaluate(X const& x) const { return model.evaluate(x); }
};
```

Для некоторых учебных задач невозможно определить хорошую  $L$ , поэтому такие методы, как перекрестная проверка, не работают. Обычно используются специализированные методы выбора параметров.

Для пользователя базовое правило исследования состоит в том, чтобы не рассматривать никакие  $A$ , зависящие от параметров, если параметры нельзя выбрать автоматически, за исключением случаев, когда для выбора достаточно знания предметной области (а это редкость).

## 25.12. Общие стратегии выбора модели

Было предложено много эвристических принципов индукции. Некоторые из них согласованы, т. е. способны производить согласованные  $A$  в общих случаях, но это лишь говорит о качестве конструирования. Большинство так или иначе формализуют бритву Оккама:

- ♦ выбор модели на основе проверки гипотез аналогичен выбору на основе проверки, но вместо проверки на отдельных данных выполняются парные проверки гипотез. Мы принимаем простейшую модель за нулевую и используем что-то вроде знакового теста, переходя к более сложным моделям. Обычно этот метод работает намного быстрее, чем проверка, но из-за многократного тестирования в нем делается много выводов из одних и тех же данных. Однако достоверность 95% не имеет большого значения для хорошей производительности в будущем, так что этот метод часто выбирает слишком простые модели и используется редко;
- ♦ *регуляризация* — «исправление» SRM, используя  $aB$  вместо  $B$ , где  $a \in [0, 1]$  и определяется перекрестной проверкой. Таким образом, данные решают, как сильно штрафовать алгоритм за сложность. Дальнейшее смягчение заключается в использовании цели поиска  $= R_{h,n} + aC(h)$  для некоторой эвристики  $C$ , такой как  $|h|$ , или количество параметров, где  $a \in [0, \infty)$ . Многие  $A$  имеют некоторую форму регуляризации, тогда как SRM/ORM в основном концептуальны. Хорошая регуляризация обеспечивает асимптотическую простоту. Основная цель SRM — показать, что поиск учащегося методом полного перебора работает. Регуляризация — это практический аспект. Поэтому откажитесь от идеи контроля ошибок оценки SRM в наихудшем случае;
- ♦ принцип *минимальной длины описания* (Minimum Description Length, MDL) — свести обучение к сжатию, т. е.  $f = \operatorname{argmin}_h (|h| + |\gamma(h)|)$ , используя некоторый код для обоих (см. [25.15]). Предполагается, что если данные можно сжать, значит, вы уже что-то о них знаете, т. е. вам нужно меньше информации для исправления ошибок  $h$ , чем для получения данных. Не существует известного неравенства конечной выборки, которое бы подкрепило этот принцип, а полезность кода, полученного из примеров, безусловно, зависит от  $n$ . Этот подход кажется столь же правильным, как и стремление к простоте, и в некоторых случаях последовательным (см. [25.15]). В отличие от ORM, MDL применяется, даже если  $L$  неограничена. *Сырой MDL* использует специальный код для  $|h|$  и  $|\gamma(h)|$ . Можно закодировать  $|h|$ , как и в ORM. Для  $|\gamma(h)|$  кодируются только различия между предсказанным и фактическим  $y$ , чего достаточно для восстановления последнего. В некоторых случаях математика упрощается, потому что аддитивные члены, постоянно действующие в модели, не имеют значения. В ORM можно использовать универсальные коды, например, для реального кодирования ошибок и их признаков. Знаки занимают по 1 биту и выпадают из расчетов как константы. Если предположить, что ошибки следуют распределению с PDF  $p$  и что ошибки дискретизированы до некоторого крошечного интервала  $d$ , для кодирования требуется около  $-\lg(dp(\text{ошибка}))$  битов, поэтому  $d$  отбрасывается как постоянный аддитивный член, а  $-\lg(p)$  остается. Например, для нормального  $p$  со средним значением 0 после некоторых упрощений  $|\gamma(h)|$  — это просто ошибка  $L_2$ . В задаче классификации кодирование усложняется: с  $k$  классами предсказание либо правильное, либо допускает одну из  $k(k-1)$  ошибок. Используя единый код для них и подсчитывая количество возможностей, вы получите многочлен, поэтому с  $m$  пра-

вильными ответами и  $e_i$  неправильными для ошибки  $i$  нужно  $\lg\left(\frac{n!}{m! \prod e_i!}\right)$  битов,

чтобы исправить ошибки. Механизм несколько громоздкий, поэтому при ограниченных  $L$  ORM выигрывает у MDL. Обычно оптимальным вариантом является подсчет количества возможностей и использование универсальных кодов на результате. Сырой MDL интуитивно привлекателен, но процесс разработки кодов для общих  $A$  сложен, а штраф за сложность кажется слишком большим, поэтому его практическое использование по существу ограничивается определением штрафов для регуляризации;

- ♦ *жадное разделение пространства* — создать некоторую многомерную структуру данных и рекурсивно разбивать  $X$ , используя простое  $h$  для каждого разбиения. Механизм рассматривает только небольшую часть  $f$  на каждом шаге, что уменьшает количество рассматриваемых  $h$  и, следовательно, возможность переобучения.

## 25.13. Смещение, дисперсия и бэггинг

Схема «Аппроксимация — оценка — оптимизация» — это не единственная полезная декомпозиция. Интуитивно  $A$  может ошибаться из-за:

- ♦ *смещения* — предпочтения одних функциональных отношений другим в силу имеющихся знаний. В отличие от ошибки аппроксимации, смещение учитывает при использовании локального поиска все решения, такие как ошибка оптимизации;
- ♦ *дисперсия* — получение разных  $f$  из разных случайных  $S$  одинакового размера из-за невозможности эффективно оценить параметры, в основном вследствие переобучения. В отличие от ошибки оценки, она учитывает часть ошибки оптимизации и другие случайности в решениях.

Смещение подобно частичной слепоте, а дисперсия подобна галлюцинациям — для ясности зрения нужно свести к минимуму и то и другое. Для их измерения нужно использовать функцию  $L$  в рамках задачи. Пусть  $p$  — случайная предикторная переменная и ее *оптимальная комбинация*  $C(p) = \operatorname{argmin}_m E_p[L(p, m)]$ . Например, если  $L$  — потери регрессии  $L_2$ ,  $C(r)$  — среднее значение, для потерь  $L_1$  — медиана, а для бинарных потерь классификации — мода. Обратите внимание, что  $\text{oB}(x) = C(y|x)$ , т. е. оВ может генерировать произвольное количество выборок из  $y|x$  и комбинировать их.

Поскольку мы работаем с набором данных размера  $n$ , потребуется выборка  $S$  размера  $n$ , а следующим лучшим вариантом является *главный предиктор*  $M(x) = C(f_S(x))$ , т. е. комбинирование прогнозов обученных  $f$  на случайной выборке  $S$ . Помимо оптимальности оптимальной комбинации, главный предиктор больше ничем не полезен. Но зато это хорошая точка привязки, вокруг которой дисперсия невелика. В случае равномерной устойчивости точка привязки не используется. Затем нужно определить следующее (см. [25.11]):

- ♦  $\text{Смещение}(x) = L(\text{oB}(x), M(x))$  — эффективность основного предиктора относительно оВ;
- ♦  $\text{Дисперсия}(x) = E_S[L(f_S(x), M(x))]$  — стоимость отличий производительности от основного предиктора;
- ♦  $\text{Шум}(x) = E_S[L(\text{oB}(x), y(x))]$  — шума не избежать, т. е.  $E_x[\text{noise}(x)] = R_{\text{oB}}$ .



Теорема (см. [25.11]): для любого  $x$  и метрики  $L$ ,  $L(x, y) \leq \text{шум}(x) + \text{смещение}(x) + \text{дисперсия}(x)$ . Для потери  $L_2$  в регрессии имеем равенство. Это оправдывает определения смещения и дисперсии, которые являются обобщением классических статистических определений для регрессии. Нужно получить  $A$ , где оба значения малы. Дисперсию можно оценить с помощью *бэггинга*, который имитирует основной предиктор с использованием повторной выборки начальной загрузки:

1.  $T$  раз для некоторого  $T$ , например, 300:
2. Создать повторную выборку  $S$  размера  $n$  из набора данных.
3. Обучить  $A$  на ней, чтобы получить  $f$ .
4. Чтобы предсказать  $x$ , сформировать главный предиктор из всех  $f$  и вернуть его ответ.

Чтобы оценить ожидаемые смещение и дисперсию  $x$ , используйте выборки, полученные путем запуска основного предиктора на  $S$ , и для конкретного примера задействуйте в обучении только  $f$ , которые не использовались. Для оценки смещения, если  $\sigma V$  неизвестно, используйте потерю основного предиктора как комбинированную меру смещения + шума. Поскольку задача имеет фиксированный шум для любого  $A$ , это по-прежнему позволяет обнаруживать различия смещения между различными  $A$ .

Механизм бэггинга считается точным, но несовершенным. Он вводит случайность, используя образцы начальной загрузки для обучения и готовые образцы для оценки. Оба варианта отличаются от использования настоящих случайных выборок.

Алгоритм  $A$  с достаточно высокой дисперсией считается *неустойчивым*. Многие  $A$  были изучены с использованием бэггинга и подобных методов, и были сделаны некоторые интересные выводы (см. [25.11, 25.32]). Например, смещение и дисперсия различаются у разных вариантов выбора параметров и других решений. Обычно приходится достигать компромисса, т. е. потерять одно, чтобы получить другое.

Из-за отличной производительности бэггинг, где  $A$  = дерево решений (см. главу 26. *Машинное обучение: классификация*), сам по себе является хорошим  $A$  для задачи классификации, хотя вместо него используется случайный лес (та же глава). Из-за начальной загрузки бэггинг немного увеличивает смещение  $A$  и значительно снижает дисперсию, поэтому его использование обычно идет на пользу  $A$  с высокой дисперсией. Возможность формировать задачу главного предиктора для любой задачи делает бэггинг и его улучшения очень расширяемыми.  $A$  является *несмещенным* тогда и только тогда, когда для любого  $x$   $\text{bias}(x) = 0$ . Важным выводом является то, что дерево решений имеет тенденцию к слабому смещению и неустойчивости.

Интересным свойством такого ансамбля *рандомизации* является тот факт, что объединение большего количества базовых учеников не приводит к переобучению, вопреки рассуждениям, основанным на сложности. Хотя это формально доказано для частного случая (см. [25.7] — идея состоит в том, что для классификации любого класса пропорция голосов сходится к фиксированному значению), в целом это верно. Полуформальное доказательство: конкретные  $m$  приводят к конкретному неизвестному общему риску, а близкие  $m$  должны приводить к аналогичным значениям риска. Рассмотрим риск оптимальной комбинации  $R(p) = \min_m E_p[L(p, m)]$  для ансамбля. Предположим, что  $L$  ограничена. Тогда при наличии  $T$  базовых учеников, заставив любого из них дать произвольный прогноз,  $R$  изменится только на  $\leq L/T$ , независимо от того, изменится ли текущее значение  $\min_m$ . Согласно неравенству МакДиармида (см. [25.24])  $R$  сходится к своему ожидаемому значению экспоненциально быстро по  $T$ . Это значение  $R$  соответ-

ствуется, возможно, бесконечному набору  $m$ . По мере добавления новых базовых учеников  $R$  перестает меняться, но набор сжимается. Для дискретного  $m$  он сходится к одному или нескольким значениям, равным  $R$ . В более общем случае и для непрерывного  $m$  он сходится к счетному множеству окрестностей равных минимумов  $R$ . После некоторого  $T$  количество окрестностей становится фиксированным, а размер каждого из них достаточно мал, чтобы не имело значения, какое  $m$  в конечном итоге будет выбрано. Таким образом, в обоих случаях после конечного  $T$  набор материально различных рассматриваемых  $m$  перестает изменяться, и добавление сколь угодно большего количества базовых учеников, эффект от которых больше, чем от первых  $T$ , не имеет значения. Но здесь есть некоторые технические проблемы — например, для некоторого  $L$  предельных значений может не существовать, но у ограниченных  $L$  они есть, и поскольку мы отбираем одни и те же данные с учетом разумного  $m$ , для разумного  $L$  всегда есть пределы.

Таким образом, у ансамблей рандомизации нет дрейфа, когда добавление большего количества базовых учеников приводит к увеличению переобучения. Интуитивно понятно, что множественное тестирование не выполняется путем объединения, а устойчивость в некоторой степени контролируется косвенно.

Смещение и дисперсия зависят от  $n$  и не являются асимптотическими, в отличие от шума. Часть дисперсии, связанная с  $n$ , будет уменьшаться по мере увеличения  $n$ . Но сравнения смещения и дисперсии по-прежнему имеют смысл для конкретных наборов данных, потому что сделанные выводы, вероятно, будут справедливы для множества  $n$ , хотя, возможно, и не для асимптотического случая. Нет противоречия в том, что  $A$  одновременно и смещен, и согласован, потому что алгоритм может быть смещен, но иметь нулевую дисперсию. Это случай  $k$ -NN (см. главу 26. *Машинное обучение: классификация*, [25.11]). Многие доказательства непротиворечивости на самом деле показывают, что смещение стремится к 0 за счет медленного увеличения  $G$ , так что дисперсия также стремится к 0.

При использовании функций потерь, отличных от  $L_2$ , дисперсия может компенсировать часть смещения. Альтернативное разложение заключается в эффектах смещения и дисперсии (см. [25.18]), которые вместе точно учитывают все потери, а не просто ограничивают их. Но бэггинг не работает для оценки эффектов, и если вы попытаетесь описать взаимодействие смещения и дисперсии, они утратят простой интуитивный смысл.

Структурное смещение и дисперсия — интересная идеологическая концепция. Объединение предикторов объединяет и сглаживает их разбиения. Но это имеет значение только в той мере, в какой  $L$  способна это измерить. Например, даже при полном знании распределения  $Z$  у вас может не быть поддержки примеров в определенных областях, а это означает, что отдельные разделы будут неразличимы. Кроме того, определенные различия в разделах приводят к одним и тем же результатам, например для классификации, поэтому  $L$  не может ловить точные различия даже при наличии необходимой поддержки примеров. Это говорит о том, что в общем случае раздел  $oV$  не уникален. Кроме того, разные разделы могут иметь разную дисперсию, но одинаковый эффект от нее.

Использование ансамблей может уменьшить следующие показатели (см. [25.9]):

- ♦ ошибку аппроксимации — объединение базовых обучающихся расширяет  $G$  за счет включения функций, не принадлежащих  $G$ ;

- ♦ ошибку оценки — сочетание слабо оцененных базовых учащихся может дать хорошо оцененные средние значения;
- ♦ ошибку оптимизации — уменьшает риск того, что одна неудачная эвристическая оптимизация приведет к плохому результату.

## 25.14. Паттерны проектирования

Существуют определенные повторяющиеся решения общих проблем. Приведем примеры таких решений:

- ♦ *ставка на редкость* — сложная модель со многими параметрами вряд ли будет эффективной, поэтому нужно выставить как можно больше неэффективных значений, например нулевых, чтобы они не оказывали влияния на алгоритм и уменьшали сложность (см. [25.17]);
- ♦ выполняйте оптимизацию приблизительно — из-за ошибки оценки разница между приблизительным и оптимальным решениями обычно очень мала. *Ранняя остановка* — это остановка оптимизации после некоторой логической вехи, или когда отслеживаемая ошибка набора проверки начинает увеличиваться. Преждевременный рост говорит о возможных проблемах;
- ♦ *выпуклость* — если оптимизация для определенного  $L$  сложна, т. к. функция не выпуклая, найдите *суррогатную функцию потерь*, которая ограничивает  $L$  и приводит к проблеме выпуклой оптимизации. Из-за ограничений риск решения может быть очень близок к риску оптимального решения исходной задачи;
- ♦ *получите больше данных* — обычно это самый простой и эффективный способ снизить риск. Посредственная модель с большим количеством данных часто превосходит отличные модели с небольшим количеством данных (см. [25.12]). Обучение в этом случае проваливается, особенно для  $x$ , которые сильно отличаются от всех известных примеров. Любая изученная  $f$  является практически неполной, т. е. неприменимой к любой  $x \in X$ , точно так же, как численная интерполяция сильно отличается от экстраполяции;
- ♦ используйте знание предметной области — любая информация, которую  $A$  не нужно изучать, может быть очень полезной. Например, при обработке изображений соседние пиксели можно считать связанными;
- ♦ используйте SGD (см. главу 24. *Численная оптимизация*) со штрафом за регуляризацию — это удобно для моделей с субградиентами, поскольку обучение проводится в реальном времени, эффективно, а недооптимизация ведет к сохранению малой ошибки оценки;
- ♦ доверяйте только проверенным  $A$  — для многих задач было предложено и продолжает предлагаться слишком много  $A$ , и практически невозможно перебрать и сравнить их все. Тут возникает ошибка оценки, заключающаяся в том, что качества каждого  $A$  известны не полностью и их обнаружение является дорогостоящим с точки зрения времени исследователя. Разумная стратегия состоит в том, чтобы рассматривать только те  $A$ , которые экспериментально показали наибольшую производительность при решении множества задач кем-то, кроме создателей, или обладающих другими желательными качествами, такими как интерпретируемость, простота ис-

пользования и настройки параметров, эффективность и качественный дизайн, который оправдывает производительность. Хотя другие  $A$  могут улучшить  $R_f$ , по крайней мере, для конкретных задач, вы можете сосредоточить усилия на подготовке данных. Тем не менее в задачах проблем, имеющих большое экономическое значение, требуется провести обширные исследования.

У всех известных  $A$  есть проблемы, но их трудно исправить, не сломав что-то еще. Конечная цель — развернуть  $f$  как часть системы принятия решений. Есть и другие критерии, которые мы хотим видеть в  $A$ :

- ◆ опыт получения  $f$  с низким уровнем риска для общих наборов данных, особенно похожих на данные из решаемой задачи;
- ◆ разумная оперативность — в идеале обучение занимает максимум несколько дней, а принятие решения происходит мгновенно;
- ◆ возможность работать как «черный ящик» с минимальной помощью от пользователя — любые параметры должны настраиваться автоматически, но разумно, чтобы пользователи предоставляли специфичную для предметной области информацию, такую как приблизительные диапазоны параметров или настраиваемые функции;
- ◆ простота — например, компания Netflix решила не использовать слишком сложного победителя конкурса рекомендаций ([http:// techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html](http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html));
- ◆ интерпретируемость — изученный раздел должен быть понятен человеку, и часто предпочитают  $f$  с простым представлением знаний.

## 25.15. Подготовка данных

Выбор признаков — самый важный момент обучения. Неинформативные или случайные функции, такие как идентификатор записи (в которых легко допустить ошибку, когда данные автоматически считываются из какой-либо базы данных), могут повлиять на  $f$ , поскольку случайно может показаться, что такие данные обладают предсказательной силой. Хотя некоторые области имеют естественные особенности, для достижения наилучших результатов следует конструировать признаки, максимально основанные на знании предметной области. Например, при попытке предсказать уровень здоровья человека лучше вместо роста и веса использовать индекс массы тела, потому что, учитывая вес, модель не обязательно узнает, что рост тоже имеет значение. Кроме того, в таких играх, как шахматы, люди определяют качество позиции, используя количество фигур, мобильность, структуру пешек и т. д., и нецелесообразно изучать их по расположению фигур (по крайней мере, так считалось до недавнего времени — см. [https://www.chessprogramming.org/Deep\\_Learning](https://www.chessprogramming.org/Deep_Learning)).

Ложноинформативные признаки еще более опасны. Например, известен случай, когда военная организация пыталась обучить распознаватель танков и в целях экономии использовала для обучения фотографии, сделанные всего за два дня: с танком и без танка. В один из дней погода была облачная, и алгоритм научился предсказывать наличие танка на фото по цвету неба.

Обычно один из известных хороших  $A$  легко применить к любым данным в поддерживаемом формате. Труднее собрать данные из различных и зачастую плохо поддержи-

ваемых источников, таких как журналы в свободном формате или базы данных, т. к. данные нуждаются в очистке, т. е. удалении недопустимых значений, закодированных здравым смыслом, таких как возраст = 0, пол = «Н/Д» и т. д. Учитывая такие шумные источники и большое  $n$ , создание производных признаков может быть практически невозможным, по крайней мере, без знания предметной области для выполнения очистки. Простая эвристика для обнаружения неверных данных заключается в проверке часто встречающихся значений, поскольку они, скорее всего, будут недействительными.

Как правило,  $x$  является *числовым* (непрерывным или *дискретным*), *порядковым*, *категориальным*, смешанным или произвольным, но с расстоянием или функцией ядра (обсуждается позже). Разница между порядковым и категориальным значением состоит в том, что последний не дает отношения порядка.

Вы можете выполнять преобразования между категориальными и числовыми функциями. Для преобразования категории в число определите двоичную переменную 0/1 для любого значения категории. Это преобразование дает необходимую градиентную или линейную разделимость для многих алгоритмов.

Для преобразования числа в категорию можно определить поддиапазоны, называемые *корзинами*, и для заданного значения функции вернуть соответствующий номер *корзины*. Обычно предполагается, что объект имеет ограниченный диапазон, и помещают  $z \notin S$  или за пределами этого диапазона в корзины конца интервала. Простая стратегия разделения — на корзины равной ширины, например на  $\lg(n)$  корзин одинакового размера:

```
typedef Vector<int> CATEGORICAL_X;
class DiscretizerEqualWidth
{
    ScalerMinMax s;
    int nBins;
    int discretize(double x, int i) const
    {
        x = s.scaleI(x, i);
        if(x < 0) return 0;
        if(x >= 1) return nBins - 1;
        return nBins * x;
    }
public:
    template<typename DATA> DiscretizerEqualWidth(DATA const& data,
        int theNBins = -1): s(data), nBins(theNBins)
    {
        assert(data.getSize() > 1);
        if(nBins == -1) nBins = lgCeiling(data.getSize());
    }
    CATEGORICAL_X operator() (NUMERIC_X const& x) const
    {
        CATEGORICAL_X result;
        for(int i = 0; i < x.getSize(); ++i)
            result.append(discretize(x[i], i));
        return result;
    }
};
```

Для обучения в реальном времени можно использовать 5–10 постоянных корзин. Но при разделении теряется информация, и его лучше избегать, выбирая алгоритм  $A$ , который не нуждается в дискретизации. Чтобы свести к минимуму ошибку оценки, в каждой корзине должно быть достаточное количество примеров, но даже если это так, все равно возникнет ошибка аппроксимации, если поместить разные примеры в одну и ту же корзину.

Если есть сочетание числовых и категориальных функций, обычно лучше преобразовать последние в первые, поскольку при этом информация не теряется. Поэтому обычно представляют данные в виде числового вектора:

```
typedef Vector<double> NUMERIC_X;
```

## 25.16. Масштабирование

Многие алгоритмы отдают равный приоритет функциям при их объединении, поэтому более масштабные алгоритмы могут иметь большее влияние. Только некоторые  $A$  не обращают внимания на масштаб. Одним из способов масштабирования является сопоставление всех значений признаков на один диапазон, обычно  $[0, 1]$ , используя форму-

лу  $scaledX = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$ . Значения  $x_{\max}$  и  $x_{\min}$  рассчитываются по обучающим данным.

Масштабирование выполняется преобразованием в конвейере данных:

```
class ScalerMinMax
{
    NUMERIC_X minX, maxX;
public:
    ScalerMinMax(int D): minX(D, numeric_limits<double>::infinity()),
        maxX(D, -numeric_limits<double>::infinity()){}
    template<typename DATA> ScalerMinMax(DATA const& data)
    {
        assert(data.getSize() > 0);
        minX = maxX = data.getX(0);
        for(int i = 1; i < data.getSize(); ++i) addSample(data.getX(i));
    }
    void addSample(NUMERIC_X const& x)
    {
        assert(minX.getSize() == x.getSize());
        for(int j = 0; j < x.getSize(); ++j)
        {
            minX[j] = min(minX[j], x[j]);
            maxX[j] = max(maxX[j], x[j]);
        }
    }
    double scaleI(double xi, int i) const
    {
        double delta = maxX[i] - minX[i];
        return delta > 0 ? (xi - minX[i])/delta : 0;
    }
}
```

```

NUMERIC_X scale(NUMERIC_X x) const
{
    for(int i = 0; i < x.getSize(); ++i) x[i] = scaleI(x[i], i);
    return x;
}
};

```

Только обучающие данные нормализуются в диапазон  $[0, 1]$ , а другие данные могут отображаться за пределами этого диапазона, но это не должно вызывать проблем.

Еще одним способом масштабирования является *ученичество*, т. е. любой признак, делающий среднее значение равным 0, а дисперсию — 1. В целом этот способ работает хуже, вероятно, потому, что дисперсия как мера масштаба имеет смысл только тогда, когда данные нормальные, но лучше для  $A$ , которым требуется среднее значение 0 (обычно для SGD, т. к. это способствует сходимости):

```

class ScalerMQ
{
    Vector<IncrementalStatistics> ic;
public:
    ScalerMQ(int D): ic(D) {}
    template<typename DATA> ScalerMQ(DATA const& data): ic(getD(data))
    {for(int i = 0; i < data.getSize(); ++i) addSample(data.getX(i));}
    void addSample(NUMERIC_X const& x)
    {for(int j = 0; j < x.getSize(); ++j) ic[j].addValue(x[j]);}
    double scaleI(double xi, int i) const
    {
        double q = ic[i].stdev();
        return q > 0 ? (xi - ic[i].getMean())/q : 0;
    }
    NUMERIC_X scale(NUMERIC_X x) const
    {
        for(int i = 0; i < x.getSize(); ++i) x[i] = scaleI(x[i], i);
        return x;
    }
};

```

Масштабирование может выполняться в реальном времени с динамическим обновлением соответствующих параметров. Масштабы будут подстраиваться под данные, и первые несколько  $z_i \in S$  могут оказаться плохо масштабированы, но это не должно вызывать проблем.

Масштабирование — это не бесплатный обед. Обычно оно повышает производительность, но теряет информацию, если масштабы признаков зависят друг от друга. Например, соседние пиксели в изображениях связаны, и наличие светлого пиксела рядом с темным сигнализирует о контрасте. Требуется время  $O(nD)$  для масштабирования  $S$ .

У обоих описанных популярных методов масштабирования есть очевидные проблемы, которые часто игнорируются:

- ♦ в случае масштабирования диапазона граничные оценки не являются статистически устойчивыми и меняются в зависимости от данных. Было бы лучше использовать 5-й и 95-й процентиля как 0 и 1 соответственно, потому что они сходятся и имеют

более низкую дисперсию оценки. Для их эффективного вычисления можно использовать две кучи;

- ♦ в ученичестве дисперсия зависит от формы. Например, однородные и биномиальные данные имеют одинаковый диапазон, но разные отклонения.

Практический стандарт, по-видимому, заключается в использовании масштабирования диапазона, если  $A$  не нуждается в ученичестве.

## 25.17. Обработка пропущенных значений

Собранные векторные данные могут быть неполными, если некоторые  $x$  содержат нулевые значения, — например, респонденты опроса могли не заполнить некоторые пункты. Несмотря на то что на  $oV$  это не влияет, в применимых на практике  $A$  эта проблема имеет место. Можно отбросить выборки с пропущенными значениями, если  $n$  велико. Но исключение примеров с отсутствующими значениями может привести к смещению, если значения *выпадают не случайно*, а по некоторому шаблону (см. [25.14]).

Другой вариант — отбрасывание функций с отсутствующими значениями. Это эффективно, если в основном отсутствуют значения нескольких конкретных признаков, — например, если респонденты не хотят в чем-то признаваться.

При малых  $n$  имеет смысл использовать информацию из неполных примеров. Некоторые решения:

- ♦ использовать среднее значение или медиану значений из других примеров — самый простой вариант, но он может привести к смещению;
- ♦ задействовать значение ближайшего соседа  $x$ , рассчитанное с учетом расстояния, которое использует только признаки без пропущенных значений, — вероятно, наиболее практичная стратегия замены;
- ♦ предварительно научиться угадывать пропущенные значения — более сложная в настройке, но, вероятно, наименее предвзятая стратегия замены;
- ♦ использовать  $A$ , который может обрабатывать отсутствующие функции и не теряет информацию. Однако такой  $A$  не обязательно является хорошим выбором. Поскольку при удалении примеров с отсутствующими значениями удаляется слишком много, это кажется лучшим решением.

Факт отсутствия значения сам по себе является полезной информацией. Например, вы можете поставить точный диагноз, основываясь только на названиях анализов, запрошенных врачом.

Еще одна проблема заключается в прогнозировании на основе выборки с пропущенными значениями. Даже если работа с пропущенными значениями позволяет получить ответ, оценки производительности теряют точность. Возможно, потребуется поддерживать логику замены на основе обучающих данных.

## 25.18. Выбор признаков

Создание полезных признаков требует человеческого понимания, но алгоритм может автоматически удалять бесполезные функции и сохранять лучшее подмножество. Это позволяет достичь нескольких целей:



- ◆ получить простоту и связанные с ней преимущества. Упрощение иногда достигается за счет немного повышенного риска;
- ◆ сократить затраты на сбор данных — нет необходимости восстанавливать удаленные функции, например проводя для пациентов дополнительные медицинские тесты;
- ◆ повышение эффективности: меньше функций = меньше вычислений.

С шумными признаками работать сложнее, потому что  $A$  попытается использовать шум. Например, если имеется функция множества бросков монеты, некоторые из них могут показаться полезными и будут сохранены, что приведет к усложнению модели и снижению ее качества.

Признак является (см. [25.20]):

- ◆ *сильнореlevantным*, если его удаление из любого подмножества уменьшает  $R_{об}$ ;
- ◆ *слаборелеvantным*, если его удаление из части подмножеств уменьшает  $R_{об}$ ;
- ◆ *нерелеvantным*, если удаление его из любого подмножества не уменьшает  $R_{об}$ .

Оптимальные подмножества состоят из всех сильно релевантных и, возможно, некоторых слаборелевантных признаков и не содержат нерелевантных признаков. Например, в данных о цветках ириса каждая переменная сама по себе обеспечивает достаточно точную классификацию, поэтому для идеальной классификации могут не потребоваться все четыре. Нужны слаборелевантные признаки, потому что любая переменная из нескольких может дать необходимую информацию. Например, для изучения функции `hog` ни одна из переменных по отдельности не полезна, но полезно знать обе.

Снижение риска путем выбора признаков обычно ничего не дает. Существуют подмножества  $2^D$ -функций, и каждое из них может быть лучшим. Попытка определить, какое из них требует резервирования некоторых проверочных данных, увеличивает ошибку оценки, потому что подмножество, найденное на сокращенных данных, не обязательно должно быть лучшим для всех данных, поскольку как полезные, так и бесполезные признаки могут таковыми не выглядеть. Теоретически, чтобы утверждать, что подмножество из  $k$  признаков является лучшим, необходимо учитывать все подмножества из  $2^k$  признаков (см. [25.8]). Хороший  $A$  имеет встроенный контроль переобучения, который мало что выигрывает от отдельного выбора признаков. *Реальная цель* — *сократить количество используемых функций с минимальным увеличением риска*. Поэтому обычно используются все признаки, включенные экспертом в предметной области, а выбор делается только в том случае, если эффективность и стоимость сбора данных являются проблемой. Хорошей метрикой успеха является процент сохраненных функций.

*Встроенные методы* используют  $A$  со встроенным выбором функций. Некоторые  $A$  обычно очень эффективны и не используют некоторые признаки, придают большое значение некоторым признакам или явно стараются не использовать некоторые признаки. Они специфичны для конкретной задачи — например, деревья решений/регрессии и методы, основанные на регуляризации  $L_1$  (обсуждаемые позже в соответствующих главах). Они обучают и выбирают признаки одновременно, избегая ошибок оценки и неэффективности поиска бесплатно. Но такие  $A$  обычно не самые эффективные (иначе проблема выбора признаков была бы решена). Выбирайте встроенные методы, когда  $n$  слишком мало для использования проверочного набора или слишком велико для эффективности других методов.

Выбор признаков реализован как фильтр в конвейере данных:

```
class FeatureSelector
{
public:
    Vector<int> fMap;
public:
    FeatureSelector(Bitset<> const& selection)
    {
        for(int i = 0; i < selection.getSize(); ++i)
            if(selection[i]) fMap.append(i);
    }
    NUMERIC_X select(NUMERIC_X const& x) const
    {
        NUMERIC_X result;
        for(int i = 0; i < fMap.getSize(); ++i) result.append(x[fMap[i]]);
        return result;
    }
    double select(NUMERIC_X const& x, int feature) const
    {
        assert(feature >= 0 && feature < fMap.getSize());
        return x[fMap[feature]];
    }
};

template<typename DATA> struct FSData
{
    DATA const& data;
    FeatureSelector const & f;
    typedef typename DATA::X_TYPE X_TYPE;
    typedef typename DATA::Y_TYPE Y_TYPE;
    typedef X_TYPE X_RET;
    FSData(DATA const& theData, FeatureSelector const & theF): data(theData),
        f(theF) {}
    int getSize() const {return data.getSize();}
    X_RET getX(int i) const {return f.select(data.getX(i));}
    double getX(int i, int feature) const
        {return f.select(data.getX(i), feature);}
    Y_TYPE const& getY(int i) const {return data.getY(i);}
};

template<typename LEARNER> struct FeatureSubsetLearner
{
    FeatureSelector f;
    LEARNER l;
public:
    template<typename DATA> FeatureSubsetLearner(DATA const& data,
        Bitset<> const& selection): f(selection), l(FSData<DATA>(data, f)) {}
    int predict(NUMERIC_X const& x) const {return l.predict(f.select(x));}
};
```

Для реализации удобно использовать один алгоритм, чтобы найти несколько хороших подмножеств и упорядочить их по количеству используемых функций, а другой — чтобы выбрать признак с наименьшим риском. Типичный проверочный поиск выбирает наиболее эффективное подмножество, разрешая связи в пользу меньших:

```
template<typename RISK_FUNCUTOR> Bitset<> pickBestSubset(
    RISK_FUNCUTOR const &r, Vector<Bitset<> >const& subsets)
{return valMinFunc(subsets.getArray(), subsets.getSize(), r);}
```

Реальная цель — сократить количество используемых признаков, а не повысить производительность. Итак, вы можете выбирать наименьшие подмножества с риском  $\leq$  риска полного подмножества. В случае некоторой дисперсии в базе  $A$  результирующее множественное тестирование предпочитает гораздо более простые подмножества, компенсируя ошибку оценки и получая некоторую эффективность:

```
template<typename RISK_FUNCUTOR> Bitset<> pickBestSubsetGreedy(
    RISK_FUNCUTOR const &r, Vector<Bitset<> >const& subsets)
{
    int best = subsets.getSize() - 1;
    double fullRisk = r(subsets[best]);
    for(int i = 0; i < best; ++i) if(r(subsets[i]) <= fullRisk) best = i;
    return subsets[best];
}
```

Подмножество со всеми признаками является эталоном, включенным в алгоритм создания подмножества, и используется часто. Например, для распознавания цифр. Хотя некоторые области изображения более важны, каждый пиксел несет собственную информацию, так что только некоторые из них могут быть ненужными, но невозможность их удаления не является ошибкой, — все пикселы могут быть полезны. Но если база  $A$  имеет высокую дисперсию, риск полного подмножества может быть оценен неправильно, что приведет к выбору слишком простых подмножеств. Рассмотрите повторную перекрестную проверку только для полного подмножества.

Если подмножеств много, чтобы сократить время вычислений и ошибку оценки, можно выполнить подвыборку ранжированных подмножеств с помощью поиска по сетке. Выполнив это многократно, можно увеличить перспективные диапазоны (здесь не реализовано):

```
Vector<Bitset<> > subSampleSubsets(Vector<Bitset<> >const& subsets,
    int limit)
{
    assert(subsets.getSize() > 0 && limit > 0);
    Vector<Bitset<> > result;
    int skip = ceiling(subsets.getSize(), limit);
    for(int i = subsets.getSize() - 1; i >= 0; i -= skip)
        result.append(subsets[i]);
    result.reverse();
    return result;
}
```

*Методы-обертки* — используйте один или несколько  $A$  для поиска в некоторых подмножествах, задействовав проверочный набор для оценки риска. Существует небольшая проблема: обертки склонны выбирать признаки, которые хороши только для используемой базы  $A$  (см. [25.16]). Но поскольку обычно для повторного обучения на всех данных и в найденном подмножестве используется один и тот же  $A$ , это не имеет значения. Кроме того, из-за ошибки оценки невозможно выбрать лучшие подмножества, чтобы удовлетворить всех возможных учащихся, потому что каждый из них может использовать разные признаки, путаться из-за работы других и т. д.

Полное перечисление возможно до  $D \leq 12$ , в зависимости от  $n$  и базы  $A$ . Генерация происходит в порядке размера:

```
struct SubsetLengthComparator
{
    bool operator()(Bitset<> const& lhs, Bitset<> const& rhs) const
    {return lhs.popCount() < rhs.popCount();}
    bool isEqual(Bitset<> const& lhs, Bitset<> const& rhs) const
    {return lhs.popCount() == rhs.popCount();}
};

Vector<Bitset<> > selectFeaturesAllSubsets(int D)
{
    assert(D <= 20); // вычислительная безопасность
    int n = pow(2, D) - 1;
    Vector<Bitset<> > result(n, Bitset<>(D));
    for(int i = 0; i < n; ++i)
    {
        int rank = i + 1;
        for(int j = 0; rank > 0; ++j, rank /= 2)
            if(rank % 2) result[i].set(j);
    }
    quickSort(result.getArray(), 0, result.getSize() - 1,
        SubsetLengthComparator());
    return result;
}
```

*Прямой поиск* начинает работу без каких-либо признаков и жадно добавляет самые полезные, пока не достигнет производительности полного подмножества. Ему нужны оценки  $O(D^2)$ , поэтому он полезен, возможно, для  $D \leq 40$ :

```
template<typename RISK_FUNCTOR>
Bitset<> selectFeaturesForwardGreedy(RISK_FUNCTOR const &r, int D)
{
    Bitset<> resultI(D);
    resultI.setAll();
    double fullRisk = r(resultI);
    resultI.setAll(0);
    for(int i = 0; i < D; ++i)
    {
        double bestRisk;
        int bestJ = -1;
        for(int j = 0; j < D; ++j) if(!resultI[j])
        {
            if(i == D - 1)
            {
                bestJ = j;
                break;
            }
            resultI.set(j, true);
            double risk = r(resultI);
            resultI.set(j, false);
        }
    }
}
```

```

        if(bestJ == -1 || risk < bestRisk)
        {
            bestRisk = risk;
            bestJ = j;
        }
    }
    resultI.set(bestJ, true);
    if(r(resultI) <= fullRisk) return resultI;
}
resultI.setAll();
return resultI;
}

```

При использовании с оВ прямой поиск находит максимальное подмножество, которое нельзя увеличить для повышения производительности. Но он не может найти xor-подобные подмножества, где отдельные признаки не кажутся полезными. Тем не менее это обычно не проблема для используемых на практике данных, особенно потому, что в конечном итоге полезные подмножества обнаруживаются.

Гораздо более масштабируемый подход заключается в том, чтобы рассмотреть возможность предсказания каждой функции в отдельности и создать подмножества, добавляя по одной функции за раз от самого низкого до самого высокого риска. При большом  $D$  это позволит получить, по крайней мере, приблизительное ранжирование подмножеств. Объединив это с подвыборкой, можно достичь эффективности при оценке на следующем этапе:

```

template<typename RISK_FUNCTOR> Vector<Bitset<>> selectFeatures1F(
    RISK_FUNCTOR const &r, int D)
{
    Vector<Bitset<>> selections;
    Vector<double> risks(D);
    for(int i = 0; i < D; ++i)
    {
        Bitset<> temp(D);
        temp.set(i);
        risks[i] = r(temp);
    }
    Vector<int> indices(D);
    for(int i = 0; i < D; ++i) indices[i] = i;
    IndexComparator<double> c(risks.getArray());
    quickSort(indices.getArray(), 0, D - 1, c);
    Bitset<> resultI(D);
    for(int i = 0; i < D; ++i)
    {
        resultI.set(indices[i]);
        selections.append(resultI);
    }
    return selections;
}

```

Этот код игнорирует взаимодействие между функциями и плохо справляется с проблемами типа xor. В общем случае полезность признака нельзя определить с помощью простого теста. Например, знак Зодиака может быть полезен при определении характе-

ра человека, только если человек пытается соответствовать ему. Проблема `xor` создает множество таких примеров. Тем не менее большинство задач сильно отличаются от `xor`, и поиск по одному признаку оказывается возможным. Обычно это хорошо для удаления ненужных функций (см. [25.16]).

Общая стратегия использования оберток:

- ◆ полное перечисление, если  $D \leq 12$ ;
- ◆ жадный прямой поиск, если  $D \leq 40$ ;
- ◆ поиск по одному признаку с подвыборкой в противном случае.

Грубым обоснованием для 12 и 40 является управление во время выполнения. Предполагая, что база  $A$  работает за время  $O(D)$ , полный перебор и прямой поиск соответственно занимают время  $O(D2^D)$  и  $O(D^3)$ , что для выбранных чисел почти одно и то же:

```
template<typename RISK_FUNCTOR> Bitset<> selectFeaturesSmart(
    RISK_FUNCTOR const& r, int D, int subsampleLimit = 20)
{
    if(D <= 12) return pickBestSubset(r, selectFeaturesAllSubsets(D));
    else if(D <= 40) return selectFeaturesForwardGreedy(r, D);
    else return pickBestSubsetGreedy(r, subSampleSubsets(
        selectFeatures1F(r, D), subsampleLimit));
}
```

## 25.19. Ядра

Ядра позволяют эффективно добавлять признаки, которые являются комбинацией других признаков. Это делается с помощью функции отображения признаков  $F$ . К примеру  $F$ , равная идентичности, соответствует отсутствию отображения, а простая  $F$  может включать в себя все пары произведений признаков.

Такое отображение позволяет резко увеличить  $D$  и сделать обучение невозможным с точки зрения вычислений, поэтому обычно это делается лишь тогда, когда применяется *трюк с ядром*, т. е. когда  $A$  использует только скалярные произведения чего-то с  $x$ . Например, для линейной регрессии  $wx$  для некоторого вектора весов  $w$  становится  $F(w)F(x) = K(w, x)$ , где  $K$  — функция *ядра*, специфичная для  $F$ .  $K$  вычисляется напрямую, обычно за время  $O(D)$ , без предварительного сопоставления с расширенным пространством. Это позволяет увеличить размерность расширенного пространства до  $\infty$ .

По определению любая функция  $F$ , соответствующая скалярному произведению в расширенном пространстве, является корректной  $K$ , но не всякая функция  $K$  является допустимым ядром. Теорема (см. [26.3]):  $K$  действительно тогда и только тогда, когда матрица  $M$  размером  $n \times n$  для всех примеров такая, что  $M[i][j] = K_{ij}$ , является *симметричной и положительно определенной* (Symmetric and Positive Definite, SPSPD), т. е.  $M[i][j] = M[j][i]$  и  $\forall u \in \mathbb{R}^n uMu \geq 0$ . Доказательство:  $M$  является SPSPD  $\rightarrow \exists$  матрица  $B$ , такая что  $M = B^T B \rightarrow F(x_i) = i$ -й столбец  $B$  является картой признаков  $\rightarrow K$  действительно.

Также

$$K_{ij} = F(x_i)F(x_j) \rightarrow K_{ij} = F(x_j)F(x_i)$$

и

$$uMu = \sum u_i u_j F(x_i) F(x_j) = \left\| \sum u_i F(x_i) \right\|^2 \geq 0 \rightarrow M$$

являются симметричными и положительно определенными.

Таким образом, каждое действительное  $K$  генерирует вектор числовых признаков некоторой размерности для любого  $x$ , при этом  $x$  не обязательно должен быть вектором. Поскольку  $K$  является скалярным произведением таких векторов, это функционал гильбертова пространства, т.е. билинейность. С точки зрения понимания  $K$  измеряет сходство. В частности, расстояние  $L_2$  в расширенном пространстве хорошо определено, поскольку при использовании билинейности расстояние<sup>2</sup> =  $\|F(x_i) - F(x_j)\|^2 = K_{ii} - 2K_{ij} + K_{jj}$ .

$M$  содержит всю доступную информацию об обучающих данных. Преобразование в  $M$  ведет к потере информации. Например, поскольку скалярное произведение измеряет косинус примера относительно начала координат, любая информация о вращении теряется. Выбор  $K$  представляет собой предварительное знание предметной области о данных, и ни одно из них не подходит для всех типов данных. Для числовых векторов полезное  $K$  включает *линейный* (простое скалярное произведение) и *гауссовский* (также называемый *радиальным*) базис, определяемый формулой

$$K(x_1, x_2) = \exp\left(-\frac{\|x_1 - x_2\|}{\sigma}\right),$$

где  $\sigma$  — параметр ширины, аналогичный стандартному отклонению. Последнее может представлять собой любую непрерывную границу раздела и обычно является предпочтительным методом из-за хорошей экспериментальной производительности для классификации с помощью SVM (обсуждается в главе 26. *Машинное обучение: классификация*, см. также [26.9]):

```
struct GaussianKernel
{
    double a;
    GaussianKernel(double theA): a(theA) {}
    double operator() (NUMERIC_X const& x, NUMERIC_X const& y) const
    { // внимание! При излишне высоких значениях x - y
      //в результате получится 0
      NUMERIC_X temp = x - y;
      return exp(-a * dotProduct(temp, temp));
    }
};
```

Существуют полезные  $K$  для других типов данных, таких как строки и графики. Использование ядер позволяет изучать не векторные данные напрямую. Механизм SVM (см. главу 26. *Машинное обучение: классификация*) особым образом использует ядра напрямую. В целом *теорема о представителе* упрощает использование ядер (см. [25.24].): пусть  $g$  — неубывающая функция,  $H$  — гильбертово пространство, соответствующее любому  $K$ , а  $R$  — любая функция риска. Тогда существуют вещественные числа  $a_i$  такие, что  $f(x) = \sum a_i K(x_i, x)$  является решением уравнения.

## 25.20. Обучение в реальном времени

Многим высокопроизводительным  $A$  требуется слишком много времени или памяти для больших  $n$ . Требуется получить  $A$ , которому не нужны все данные перед обучением и который сможет изучать примеры по одному.

SGD часто является предпочтительным алгоритмом оптимизации. В большинстве случаев необходимо перепроверить его первоначальную начальную скорость, иначе она может быстро упасть до  $\infty$ . Градиенты чувствительны к масштабу, и использование той же начальной скорости, например 1, работает только в особых случаях, в основном для классификации, когда градиенты не могут быть слишком большими. Кроме того:

- ◆ SGD обладает некоторой автоматической устойчивостью, т. к. параметры изначально равны 0 или некоторым другим значениям низкой сложности;
- ◆ SGD естественным образом обрабатывает отсутствующие данные — в уравнении обновления опускаются все отсутствующие значения;
- ◆ ошибка оценки представляет собой гораздо меньшую проблему, потому что увидеть пример один раз обычно недостаточно для переобучения.

Для настройки параметров используется *гонка*. Этот метод работает для параметров, которые можно настроить в автономном режиме с помощью поиска по сетке. Любая рассматриваемая комбинация параметров создает  $A$  в реальном времени. После того как обучение почти завершено, оставшиеся данные используются для проверки, чтобы выбрать наилучшую  $f$ . В случае настройки в реальном времени, если обучение никогда не заканчивается, используйте *предварительную ошибку* — сначала оцените пример, а затем извлеките из него уроки, сохраняя скользящее среднее. При прогнозировании используется наилучшая на текущий момент  $f$ , и более ранние оценки отбрасываются для оценки  $R_f$ . Такие простые оценки, вероятно, недостаточно точны для оценки производительности, но они позволяют выбрать наилучшие или почти наилучшие параметры. Также через некоторое время могут выпасть плохо работающие гонщики (здесь не реализовано):

```
struct BinaryLoss
{
    double operator() (int predicted, int actual) const
    { return predicted != actual; }
};

template<typename LEARNER, typename PARAMS = EMPTY, typename Y = int,
        typename LOSS = BinaryLoss, typename X = NUMERIC_X> class RaceLearner
{
    Vector<LEARNER> learners;
    Vector<double> losses;
    LOSS l;
public:
    RaceLearner(Vector<PARAMS> const& p): losses(p.getSize(), 0)
    { for(int i = 0; i < p.getSize(); ++i) learners.append(LEARNER(p[i])); }
    void learn(X const& x, Y y)
    {
        for(int i = 0; i < learners.getSize(); ++i)
        {
            losses[i] += l(learners[i].predict(x), y);
```



```

        learners[i].learn(x, y);
    }
}
Y predict(NUMERIC_X const& x) const
{
    return learners[argMin(losses.getArray(), losses.getSize())].
        predict(x);
}
};

```

Этот метод, например, применяется к линейному SVM.

## 25.21. Работа с неекторными данными

Преобразование некоторых типов данных в векторную форму распространено во многих областях, но может быть громоздким. например, для следующих данных:

- ◆ текст — попробуйте *модель мешка слов*, где есть вселенная из  $D$  слов и подсчеты для встречающихся слов;
- ◆ аудио — обычно в качестве признаков используют частоты преобразования Фурье;
- ◆ изображение — используйте пиксели, возможно, уменьшая цветовой диапазон.

У текста при векторном преобразовании теряется такая информация, как отношения между позициями слов. *Неекторные данные* — это любые данные, которые нелегко преобразовать в векторную форму.

В большинстве предметных областей необходимо провести некоторую инженерию знаний, т. е. формально описать знания предметной области с помощью *онтологии*, т. е. объектной модели, описывающей происходящее. Например, можно формализовать географию, определив регионы, страны, города и т. д. вместе с соответствующими отношениями сдерживания. Как только предметная область формализована, становится проще определять функции с использованием общих шаблонов. Например, для шахмат игровое состояние обычно представляется комбинацией следующего:

- ◆ разница в материалах из-за присвоения значения каждому типу фигуры;
- ◆ позиционные факторы, такие как безопасность короля, мобильность, контроль центра, пешечная структура и т. д. Без знания предметной области единственным признаком является расположение фигур.

Вместо векторного преобразования можно использовать функции расстояния и ядра. Реализация этого принципа для различных  $A$  обсуждается в более поздних главах. Первое обычно проще, но, если дано расстояние  $d$ , можно использовать ядро  $e^{-d}$  или  $\frac{1}{d+1}$ .

## 25.22. Масштабное обучение

Основной проблемой является эффективность — хороший в общем случае  $A$  обычно занимает слишком много времени. Вы можете сделать следующее:

- ◆ использовать  $A$ , выполняющийся в реальном времени, — это работает, но такие алгоритмы, как правило, не самые точные;

- ♦ отобрать данные для создания меньшего, но все же репрезентативного  $S$ , а остальные использовать для проверки и оценки, — обычно это хорошая стратегия, потому что многие  $A$  эффективны при оценке и могут использовать для этого все оставшиеся данные. Но обучение со всеми данными должно давать более точные модели. Выборка хорошо работает с ансамблями (см. главу 26. *Машинное обучение: классификация*). Например, можно изменить случайный лес, чтобы использовать  $\leq m$  (для некоторого разумного  $m$ , такого как  $10^4$ ) примеров для мешков, или разбить данные на независимые фрагменты — иногда это называется *методом программной алхимии*;
- ♦ хранить данные на диске — работает только с алгоритмами  $A$ , которые масштабируются в  $n$  и обеспечивают эффективный доступ к кешу данных. Другие работают медленно из-за произвольного доступа и/или сверхлинейного времени выполнения. Тем не менее использование буфера может помочь. Например, использование переставленных данных напрямую вызывает операции ввода/вывода — чтобы их избежать, используйте сортировку во внешней памяти для вычисления буфера перестановки. Случайная или детерминированная перестановка также может быть эффективно выполнена таким образом:

```
template<typename X, typename Y> struct DiskData
{
    EMVector<pair<X, Y> > data;
    DiskData(string const& filename): data(filename) {}
    template<typename DATA> DiskData(DATA const& theData,
        string const& filename): data(filename)
    {
        for(int i = 0; i < theData.getSize(); ++i)
            addZ(theData.getX(i), theData.getY(i));
    }
    void addZ(X const& x, Y const& y){data.append(make_pair(x, y));}
    typedef X X_TYPE;
    typedef Y Y_TYPE;
    typedef X const& X_RET;
    X_RET getX(int i)const
    {
        assert(i >= 0 && i < data.getSize());
        return data[i].first;
    }
    double getX(int i, int feature)const
    {
        assert(i >= 0 && i < data.getSize() &&
            feature >= 0 && feature < data[i].first.getSize());
        return data[i].first[feature];
    }
    Y_TYPE const& getY(int i)const
    {
        assert(i >= 0 && i < data.getSize());
        return data[i].second;
    }
    int getSize()const{return data.getSize();}
};
```

- ♦ явно обрабатывать разреженные векторы признаков с большим количеством нулей. Например, линейный SVM SGD может обрабатывать разреженные модели, такие как мешок слов, используя карты вместо векторов. Это дает значительное увеличение эффективности.

*Большие данные* и другие страшные слова касаются крупномасштабного обучения. В них нет ничего мистического, потому что наиболее известный подход в больших данных заключается в использовании описанных ранее методов и кластеров параллельных компьютеров. Проблема заключается в том, что стандартный метод декомпозиции данных параллельного программирования обычно невозможен, потому что большинству  $A$  необходимо иметь доступ ко всем данным, если только небольших подвыборок оказывается недостаточно. В настоящее время исследования больших данных занимают следующими задачами:

- ♦ эффективные системы хранения, такие как Hadoop и Spark, уже позволяют работать с гораздо большими наборами данных;
- ♦ масштабируемые алгоритмы, вычисляющие приемлемые приближительные ответы;
- ♦ методы для больших  $D$  — больших данных по-прежнему недостаточно, чтобы преодолеть проклятие размерности и возникающие в результате ошибки оценки, несмотря на периодические заявления о том, что эта задача уже решена.

## 25.23. Выводы

У каждого  $A$  свои проблемы. Если у вас есть вычислительные ресурсы, обычно используется несколько  $A$ , а перекрестная проверка позволяет выбрать подходящий. Машинное обучение — это лишь часть интеллектуального анализа данных. Конечная цель состоит в том, чтобы извлечь пользу из данных, а машинные алгоритмы — это всего лишь инструменты. Важным аспектом, который является скорее искусством, чем методом, является поиск хорошего набора данных, задавание правильных вопросов и определение того, получают ли хорошо обоснованные выводы. В работе [25.30] приведен интересный общий обзор, а в работе [25.27] описано несколько подробных тематических исследований. У случайных данных, исходя из рассуждений о несжимаемости, вероятность возникновения хитрого паттерна мала, поэтому нужны данные, которые показывают стабильные паттерны.

Человеку свойственно видеть закономерности в случайных данных и убеждаться в их правильности, поддаваться влиянию личных предубеждений и идеологий или делать внешне мотивированные, но на самом деле неточные прогнозы. Регулярно делать экстремальные прогнозы — это рационально, потому что они иногда оказываются верными и дают много пользы, а если оказываются неверными, то это не страшно («эксперты» не признаются, что им просто повезло найти решение). В профессиональном спорте и других областях прогнозирования больших денег решающим фактором часто является наличие большего количества специфичной для предметной области информации, чем у конкурентов, что позволяет в среднем получать немного лучшие результаты.

Во многих повседневных задачах, для которых анализ данных полезен, много полезного анализа можно выполнить с помощью здравого смысла и простых методов классической статистики, таких как линейная и логистическая регрессия.

## 25.24. Примечания по реализации

Идея конвейера данных оригинальна, хотя и очевидна. Алгоритмы одинаково смотрят на данные и могут считывать их по запросу даже с диска, что делает возможным обучение даже на очень больших наборах данных.

Алгоритмы выбора функций было трудно выбрать из-за множества возможных вариантов и некоторых рекомендаций относительно того, какие из них являются лучшими. Поэтому я выбрал самые простые алгоритмы и те, чья хорошая производительность легко объяснима.

Другие алгоритмы предварительной обработки данных следуют описаниям в учебниках.

## 25.25. Комментарии

Концепция информации-подсказки является новой, но полезной, поскольку  $A$  должен принимать  $x$  в качестве входных данных и возвращать в качестве выходных данных некоторый идентификатор.

Название ORM новое. Добавлю путаницы: его также называют MDL (см. [25.29]), хотя это не сводит к минимуму описание данных. Сама граница является *границей бритвы Оккама* и аналогична *РАС-байесовским границам* с  $\Pr(h) = 2^{-|h|}$  (см. [25.23]).

Остерегайтесь терминологии — в классической математике имеет место равномерная сходимость для последовательности, равномерная сходимость по вероятности для оценки, а здесь — равномерная сходимость для  $G$ .

$G$  является *РАС-агностическим и обучаемым* (см. [25.29]), если не существует  $A$  такого, что  $\forall \epsilon > 0$  и  $p > 0$ , и существует  $n$  такое, что  $A$  возвращает  $\text{PAC}(R_{\text{ов}} + \epsilon, p)f$ . Этот вывод определяет привлекательный класс сложности, но в остальном бесполезен. Первоначальная *обучаемость РАС* менее общая и предполагает бинарную классификацию и  $R_{\text{ов}} = 0$ . Метка класса 1 формирует *понятие*  $\subset X$ , и задача состоит в том, чтобы идентифицировать его. Это менее общий метод, чем наложение распределения на  $X$  и  $Y$ . Еще одно предположение состоит в том, что, поскольку вся компьютерная информация является бинарной с ограниченной памятью,  $G$  состоит из конечного числа булевых функций. Это позволяет доказать, что некоторые логические задачи обучения можно обучить за полиномиальное время, а некоторые — нет. РАС важен только по историческим причинам, потому что его создание вызвало значительный интерес к статистической теории обучения.

Но понятие «концепция» интуитивно полезно для классификации, потому что именно так люди учатся. Эта идея не приносит пользы, потому что оценка раздела является менее общей проблемой, чем оценка понятий для любого класса. Но у концепций есть преимущество: они позволяют понять, например, что определенное изображение цифры вообще не является цифрой и не принадлежит ни к какому классу. Технически с этим можно справиться, создав класс «не знаю», но так делают редко. Для сравнения: обычная классификация требует полного разделения  $X$ , хотя некоторые его части не будут соответствовать  $x$  с немалой вероятностью или  $x$ , допустимой для предметной области. Знание количества классов  $k$  также является полезным знанием предметной области.

Существует несколько похожих друг на друга определений устойчивости, т. е. вы можете ограничить  $E[R_f - R_{f,n}]$  и использовать неравенство Маркова для вывода из этого хвостовой границы (см. [25.29]).

Статистические правила выбора моделей, такие как AIC и его варианты, эффективны для некоторых статистических моделей. Здесь лучшей считается модель, где минимальный  $AIC = \text{количество параметров} - \text{логарифмическая вероятность}$ . Недостатки:

- ◆ применяется только к моделям с четко определенным числом параметров и логарифмической вероятностью;
- ◆ при малых  $n$  нужны исправления типа  $AIC_c$ ;
- ◆ предполагается, что все рассмотренные модели достаточно эффективны;
- ◆ правила асимптотичны, и для их теоретической эффективности часто требуются особые условия.

MDL не имеет этих проблем и более интуитивно понятен. То же самое касается и байесовских методов, потому что их проще обсуждать с точки зрения универсальных кодов, чем конкретных распределений. Эти методы:

- ◆ выбор байесовской модели:  $\Pr(h|y) = O(\Pr(h)\Pr(y|h))$ .  $\Pr(h)$  — это *априорная вероятность* того, что модель  $h$  относительно хорошо объясняет данные, прежде чем увидеть их, а  $\Pr(y|h)$  — это вероятность увидеть данные, если бы они были разумно объяснены с помощью  $h$ , т. е. любые ошибки считаются шумом. Например, при работе с действительными  $y$  считается, что  $y$  остатков нормальное распределение. Тогда  $f = \operatorname{argmax}_h \Pr(h|y)$  является *максимальной апостериорной* (Maximum A Posteriori, MAP) моделью. Обычно единственное предварительное знание — предпочтение более простых моделей, но это происходит автоматически, потому что правильное априорное распределение не может быть однородным в неограниченном диапазоне и поэтому должно делать некоторые допущения, которые стремятся к простоте. То есть без достаточного количества данных можно было бы также предположить, что параметр  $\approx 0$ . Хотя произвольный выбор априорного значения может показаться неправильным, он имеет стабилизирующий эффект и уменьшает дисперсию. Интерпретация априорных значений несколько сложна, потому что почти всегда истинная функция  $\notin G$ . Априорное значение указывает только на относительную уверенность в качестве рассматриваемых моделей, и здесь парадоксов нет. Конечно, априорное распределение не должно быть догматическим (например, дельта-функция) — апостериорное распределение должно допускать любые данные хотя бы с небольшой вероятностью. Итак, априорное значение =  $\Pr(\text{структура})\Pr(\text{параметры})$ . Самая большая проблема с MAP заключается в том, что необходимо решить, насколько верить в априорную вероятность, т. е. выбрать начальные отклонения параметров. Регуляризация решает эту проблему с помощью перекрестной проверки, поэтому она предпочтительнее. Большинство форм регуляризации эквивалентны той или иной форме MAP, например штраф  $L_2$  для коэффициентов означает гауссовский априор, а  $L_1$  — лапласов;
- ◆ байесовский *апостериорный вывод* без выбора модели — аналогичен MAP, за исключением  $f = E[h] = \int_h \Pr(h|y)$ . Идея состоит в том, чтобы избежать привязки к конкретному значению  $h$ , что обычно крайне маловероятно, и сделать вывод напрямую. Это соответствует *принципу Этикара* о сохранении всех хороших объяснений. Инте-

грал обычно аппроксимируется с помощью МСМС, что особенно удобно, поскольку апостериорное значение известно с точностью до константы (см. главу 6. *Генерация случайных чисел*). Регуляризация считается неявной, если все модели имеют вероятность  $> 0$ , но в простой модели вероятность, которой нельзя пренебречь, присваивается меньшему количеству данных, чем в более сложной. Таким образом, наиболее вероятная модель зависит от данных, а достаточное количество данных означает, что предпочтительна не слишком сложная и не слишком простая модель, и чем больше данных, тем более сложная модель может быть предпочтительна. Но сложность не всегда соответствует тому, какую модель выбирают данные (см. [25.25]).

Примерно к 2000 году в теории обучения был достигнут значительный прогресс благодаря применению *неравенств концентрации* и *сложности Радемахера* (см. [25.24]), которые при совместном использовании дают полезные границы конечной выборки.

О перекрестной проверке многое известно, а многое нет (см. [25.2]). Отсутствие независимости от кратности не предотвратило неправильное использование в статистических тестах. Например, для сравнения двух  $A$  иногда рекомендуется усреднить результаты каждой кратности, получив  $k$  предположительно одинаковых нормальных оценок, и применить парный  $t$ -критерий. Эвристическая оценка дисперсии (см. [25.26]) может быть достаточно точной, но не дает никаких гарантий и занижает дисперсию в простой задаче (см. [26.36]). Скрытые зависимости создают много проблем. Например, двойная перекрестная проверка может казаться независимой, но на самом деле зависит от подобию, потому что экземпляры в одном множестве слишком похожи или слишком отличаются от экземпляров в другом относительно случайной выборки. Интересный вопрос для перекрестной проверки заключается в том, улучшает ли детерминированная перестановка стратифицированную версию. Логика состоит в том, что, несмотря на стратификацию, данные могут иметь некоторый порядок сортировки.

*Повторная задержка* случайным образом разбивает данные на тестовые и обучающие наборы одинакового размера много раз.

Повторная перекрестная проверка с тем же количеством обучений имеет меньшую дисперсию.

В случае перекрестной проверки, предельный случай  $k = n - 1$ , называемый *перекрестной проверкой с исключением одного* (Leave-one-out Cross-validation, LOOCV), привлекателен тем, что дает оценку обучения почти на всех  $n$  примерах и не требует повторения (за исключением случаев, когда  $A$  рандомизирован или зависит от порядка примера), но обычно этот метод неэффективен:

- ◆ требуется значительно больше времени, чем на повторную перекрестную проверку;
- ◆ из-за использования тестовых примеров из одного и того же набора данных вместо независимых экземпляры с задержанными экземплярами либо очень похожи, либо отличаются от экземпляров обучающего набора, что приводит к высокой дисперсии оценки для многих  $A$ . Для регулярной перекрестной проверки в наборе тестов, скорее всего, будет равное число похожих и непохожих экземпляров. Интуитивно понятно, что меньшее значение  $k$  быстрее и дает меньшую дисперсию оценки, а большее уменьшает смещение оценки. LOOCV имеет высокую дисперсию по отношению к небольшим изменениям  $S$ . Если  $A$  не является случайным или не зависит от порядка примера, LOOCV становится детерминированным, поскольку дает тот же результат на тех же данных, но выбор  $S$  по-прежнему придает ему случайность.

Для небольших наборов данных и очень стабильного  $A$  предпочтительным методом может быть LOOCV. Но регулярная перекрестная проверка работает почти так же хорошо и используется чаще всего.

Естественная статистическая альтернатива перекрестной проверке — использование начальной загрузки для многократного обучения на  $n$  повторных выборках и проверки на всех данных. Но в этом случае 63,2% обучающих данных будут в тестовом наборе, что приведет к слишком оптимистичной оценке риска. Тестирование только данных, не отобранных для обучения, устраняет это смещение, но переоценивает риск, поскольку обучающий набор содержит около 0,632 данных. По всей видимости, начальная загрузка — это не подходящий метод, поскольку ее предположение о том, что распределение функционала выборки  $\approx$  распределению повторных выборок, сомнительно из-за сильного влияния повторяющихся данных.

Было предложено множество декомпозиций смещения и дисперсии, которые использовались для различных выводов (см. [25.18]). Несмотря на технические различия, все они достаточно похожи, так что общие выводы, сделанные одним, применимы к другому, особенно в отношении стабильности и низкого смещения.

При выборе параметров теоретический вопрос заключается в том, является ли использование стратегии выбора последовательным (см. [25.8]). На практике это, как правило, не имеет значения, потому что в пределе применяется согласованность, а многими смещениями можно пренебречь при достаточно большом  $n$ .

Для сырого алгоритма MDL использование распределения Коши для ошибок даст логарифмическую ошибку, но это больше похоже на сжатие и на универсальный гаммакод. Кроме того, для действительного  $y$  можно дискретизировать ошибки как  $\lceil \text{error}/d \rceil$  для некоторого небольшого интервала дискретизации  $d$  и напрямую кодировать с помощью гаммы, но это громоздкий метод. *Точный MDL* — это предположительно улучшенная версия MDL, особенно с точки зрения использования оптимальных кодов (см. [25.15]). Этот метод сложен, и получение его кодов нетривиально даже для простых параметрических моделей, таких как линейная регрессия (результат аналогичен BIC), а для непараметрических моделей он работает.

Более дорогой, но интересный способ масштабирования — запомнить  $S$  и использовать его для преобразования значений в ранги, которые не зависят от масштаба.

Еще один часто используемый метод создания корзин — *равная частота*. При этом диапазоны корзин определяются таким образом, чтобы количество точек данных в каждой корзине было примерно одинаковым. Таким образом, каждый интервал содержит данные. Число корзин должно быть таким, чтобы в каждом было достаточно примеров для точной оценки, поэтому используйте число  $\sqrt{n}$ , что предположительно дает хороший баланс между ошибками приближения и оценки. Этот метод, кажется, немного превосходит другие методы создания корзин (см. [25.14]). Интересная идея состоит в том, чтобы в каждой ячейке было  $\geq 30$  предметов.

Существует много других способов заполнения пропущенных значений (см. [25.14]), но все они сомнительны, т. к. вносят смещения. Еще один метод предварительной обработки, который обычно не используется, — это выбор экземпляра либо из соображений эффективности, либо из соображений устранения шума. Не существует неэвристических способов определения плохих экземпляров, а хороший  $A$  пригоден для извлечения правильной информации из каждого экземпляра.

Для малых  $D$  обучение можно улучшить за счет человеческого понимания. Один из методов состоит в том, чтобы нанести на график все пары переменных, включая подсказку, и посмотреть, имеется ли в данных визуально различимый паттерн. Кроме того, можно построить каждую функцию по отдельности и увидеть ее распределение.

В некоторых случаях признаки получаются бесполезными:

- ◆ если подмножество признаков полностью линейно коррелировано, ни один из них не может быть сильно релевантным, и вы можете удалить все, кроме одного. На практике идеальная корреляция маловероятна, но можно использовать, возможно,  $|\text{корреляция}| \geq 0,95$ . Можно вычислить матрицу корреляции, выделить достаточно коррелированные записи и жадно выбирать переменные, коррелирующие с большинством других переменных, чтобы минимизировать их общее количество, но это требует больших вычислительных ресурсов;
- ◆ если у признака примерно нулевая дисперсия, он нерелевантен, и его можно удалить.

Как правило, их не стоит проверять, потому что для вычисления корреляционной матрицы требуется время и пространство  $O(D^2)$ , а признаки редко имеют  $\approx$  околонулевую вариантность. Также существует некоторая ошибка оценки — например, сильно коррелированные признаки в  $S$  не обязательно должны быть таковыми вообще. Это усугубляется многократным тестированием, потому что некоторые функции могут быть сильно коррелированы случайно.

В задаче выбора признаков существует также *обратный поиск*. Он начинается с подмножества, содержащего все признаки, и жадно удаляет наименее полезные признаки по одному. При использовании с оВ он находит минимальное подмножество, которое нельзя уменьшить без увеличения риска. Но если в начале имеется много слабореlevantных признаков, этот алгоритм может легко удалить хорошие признаки из-за их кажущейся малой выгоды. Однако для типичных наборов данных это не кажется проблемой, а «улучшения» алгоритма, такие как использование оценок отдельных признаков для разрешения связей, дают худшие результаты. У жадного прямого поиска существуют те же риски, но он работает намного быстрее и находит меньшие подмножества. *Двунаправленный поиск* объединяет результаты как прямого, так и обратного поиска, но работает медленнее, чем эти два, а риски производит такие же.

Можно попытаться улучшить поиск по одному признаку, приняв во внимание взаимодействие функций. Для этого нужно оценить  $A$  для любых пар признаков  $i$  и  $j$ , начиная с пустого подмножества  $B$ , жадно добавлять в него признак  $\arg \min_i \left( R_i - \sum_{j \in B} \frac{R_{ij}}{|B|} \right)$ .

Идея заключается в том, чтобы следующий выбранный признак был наиболее полезен и наименее избыточен. Но этому методу требуется  $O(D^2)$  времени и места, и экспериментально это не окупается с точки зрения количества выбранных функций или риска.

Часто упоминаемый класс алгоритмов выбора признаков — это *методы фильтрации*, которые быстро эвристически оценивают важность признака для его исключения или ранжирования. Методы, которые рассматривают одну переменную за раз, обычно бесполезны, потому что поиск по одному признаку является более общим, позволяет использовать хороший алгоритм  $A$  и дает аналогичную эффективность на множестве хороших  $A$ .



Классическая статистика содержит тесты с одной переменной, в которых признаки рассматриваются по одному. Если есть числовой вектор  $x$ , тесты можно использовать на:

- ◆ числовых значениях  $y$  (ранговая корреляция) — тест выявляет монотонные отношения;
- ◆ категориальных  $y$  (ANOVA) — тест проверяет, являются ли достаточно значительными различия между специфическими для категории средними значениями непрерывных переменных. Невозможно использовать более надежный критерий Фридмана, потому что обычно количество  $z_i$  в разных категориях различно.

Результаты можно использовать для прямого удаления незначительных объектов или создания последовательности подмножеств путем жадного ранжирования их по значимости, как при поиске по одному объекту. Несмотря на привлекательность этой идеи:

- ◆ ни разница между средними, ни ранговая корреляция не означают, что признак полезен для используемого  $A$ , независимо от их существенности;
- ◆ тесты делают предположения, которые могут не выполняться, игнорируя при этом множественное тестирование.

Вы можете создать для пользователя отчет по признакам, содержащий, возможно, диапазон, среднее значение и медиану всех признаков, а также результаты тестов, но этот метод не масштабируется, и обертки делают свою работу быстрее и лучше, чем пользователь (хотя и человек порой может заметить что-то полезное).

Более совершенным и, возможно, самым известным фильтром является  $mRMR$  (см. [25.14]). Подобно поиску обертки по всем парам, он вычисляет взаимную информацию  $MI$  между всеми парами признаков и всеми признаками и  $y$ , а также вычисляет на-

бор подмножеств, жадно добавляя признак  $\arg \min_i \left( MI(x[i], y) - \sum_{j \in S} \frac{MI(x[i], x[j])}{|B|} \right)$ .

Он работает аналогично, но существует и проблема: на сегодняшний день лучший практический способ оценки  $MI$  (см. [25.21]) требует использования 2D-запроса  $k$ -NN, который можно сделать эффективным по времени, но для него требуется  $O(n)$  дополнительного пространства.

*Метод смягчения* (см. [25.14]) работает схожим образом, но ему тоже нужен запрос  $k$ -NN, у него есть те же недостатки, что и у  $mRMR$ , и он менее общий. В случае фильтров в качестве альтернативы выбору подмножества иногда предлагается сохранение некоторого процента лучших функций в надежде, что они содержат лучшее подмножество. Это опасно, потому что можно легко выбрать слишком маленькое подмножество из-за отсутствия обратной связи от  $A$ , т. к. фильтры не знают, как будут использоваться функции.

Из-за отсутствия опубликованных экспериментальных сравнений было предложено много других алгоритмов выбора признаков, но они бесполезны.

*Метод извлечения признаков* (см. [25.14]) отображает все признаки в предположительно более информативное пространство признаков и использует некоторые или все признаки в этом пространстве. Как правило, сначала выполняется масштабирование. Качество обычно зависит от того, насколько хорошо удастся восстановить исходные объекты или насколько хорошо сохраняются такие свойства, как относительные расстояния. Потенциальные преимущества метода:

- ◆ меньшие значения  $D$  и более высокая эффективность;
- ◆ меньшая ошибка оценки из-за приближенного представления данных, которое усредняет шум.

Но как выполнить это качественно — непонятно. У популярных методик есть недостатки:

- ◆ *анализ главных компонент* (Principal Component Analysis, PCA) сохраняет только  $k < D$  векторов, ответственных как минимум за 95% дисперсии  $X$  (где  $k$  — довольно интересная мера сложности данных). Для этого необходимо вычислить матрицу преобразования  $D \times k$  за время  $O(nD^2)$  и до некоторого разумного  $D$ . Но использование направления дисперсии фактически предполагает многомерное нормальное распределение данных, поэтому результат оказывается неэффективен для многопиковых или других ненормальных распределений, потому что PCA фактически теряет информацию, которую хороший  $A$  мог бы использовать. Методы вроде *ядерного PCA* (см. [25.17]) и разреженные варианты PCA (см. [26.29]) стремятся улучшить этот метод, но время выполнения первого составляет  $O(n^3)$ , а второй не помогает интеллектуальным  $A$ ;
- ◆ *метод случайной проекции* вычисляет матрицу случайной проекции (см. [25.34]), которая имеет тенденцию сохранять расстояния между  $x$ . Но большинство  $A$  теряют из-за утраты информации больше, чем выигрывают от сокращения  $D$ , поэтому у этого метода нет хороших вариантов использования.

Преобразования признаков позволяют скрывать ненужные переменные. Выполнение таких преобразований противоречат идее прямого решения искомой проблемы, но они могут потребоваться для смешанного обучения (см. главу 29. *Машинное обучение: другие задачи*). Чтобы оправдать общее использование извлечения признаков, нужны обширные экспериментальные данные.

Существует модель обучения в реальном времени, которая допускает *дрейф*, т. е. изменения распределения по  $Z$  (см. [25.13]). Например, вероятность того, что кто-то попадет в середину мишени при игре в дартс, увеличивается по мере обучения. Эту модель сложно обрабатывать как теоретически, так и вычислительно. Простая практическая стратегия состоит в том, чтобы время от времени перерабатывать модель для защиты от изменений в распределении, но это работает не во всех случаях.

В случае оценки в реальном времени возможны более сложные стратегии (см. [25.13]). Один из простых вариантов — снижение ошибки из предыдущего примера с помощью некоторого небольшого коэффициента для учета прогресса в результате обучения, но неясно, как выбрать надежный коэффициент снижения и чем именно это выгодно. Проблему с дрейфом можно решить, используя скользящее окно некоторого числа последних примеров.

## 25.26. Советы по дополнительной подготовке

- ◆ Имеет ли смысл для организации конвейера данных использовать в C++ наследование вместо шаблонов? Влияет ли это на удобство и производительность?
- ◆ Попробуйте в действии надежную версию масштабирования средней дисперсии с использованием в качестве опорной точки медианы и  $MADN$  (замените расчет

средней дисперсии медианами) или *IQR* (и с 75-го процентиля по 25-й) в качестве масштаба (см. главу 21. *Вычислительная статистика*). Выполните тест с классификацией и SVM.

- ◆ Создайте общий анализатор файлов CVS, чтобы не создавать парсер-анализатор для каждого набора тестовых данных. Он должен работать на большинстве наборов тестов из UCI в «матричном» формате с указанным символом-разделителем. Используйте соответствующую проверку надежности, например проверяйте конечность считываемых чисел.

## 25.27. Список рекомендуемой литературы

- 25.1. Anguita D., Ghelardoni L., Ghio A., & Ridella S. (2013). A survey of old and new results for the test error estimation of a classifier. *Journal of Artificial Intelligence and Soft Computing Research*, 3(4), 229–242.
- 25.2. Arlot S., & Celisse A. (2010). A survey of cross-validation procedures for model selection. *Statistics Surveys*, 4, 40–79.
- 25.3. Bache K. & Lichman M. (2013). UCI Machine Learning Repository [<http://archive.ics.uci.edu/ml>]. University of California. Accessed 10/19/2014.
- 25.4. Bengio Y., & Grandvalet Y. (2004). No unbiased estimator of the variance of k-fold cross-validation. *The Journal of Machine Learning Research*, 5, 1089–1105.
- 25.5. Bolón-Canedo V. (2014). Novel Feature Selection Methods for High Dimensional Data. PhD Thesis.
- 25.6. Blum A., Kalai A., & Langford J. (1999). Beating the hold-out: Bounds for k-fold and progressive cross-validation. In *Proceedings of the Twelfth Annual Conference on Computational Learning Theory* (pp. 203–208). ACM.
- 25.7. Breiman L. (2001). Random forests. *Machine learning*, 45(1), 5–32.
- 25.8. Devroye L., Györfi L., & Lugosi G. (1996). *A Probabilistic Theory of Pattern Recognition*. Springer.
- 25.9. Dietterich T. G. (1998). Approximate statistical tests for comparing supervised classification learning algorithms. *Neural computation*, 10(7), 1895–1923.
- 25.10. Domingos P. (1999). The role of Occam's razor in knowledge discovery. *Data Mining and Knowledge Discovery*, 3(4), 409–425.
- 25.11. Domingos P. (2000). A unified bias-variance decomposition. In *Proceedings of 17th International Conference on Machine Learning*. Morgan Kaufmann (pp. 231–238).
- 25.12. Domingos P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, 55(10), 78–87.
- 25.13. Gama J. (2010). *Knowledge Discovery from Data Streams*. CRC.
- 25.14. García S., Luengo J., & Herrera F. (2014). *Data Preprocessing in Data Mining*. Springer.
- 25.15. Grünwald P. D. (2007). *The Minimum Description Length Principle*. MIT Press.
- 25.16. Guyon I. (2008). Practical feature selection: from correlation to causality. *NATO Science for Peace and Security*, 19, 27–43.
- 25.17. Hastie T., Tibshirani R., & Friedman J. (2009). *The Elements of Statistical Learning*. Springer.
- 25.18. James G. M. (2003). Variance and bias for general loss functions. *Machine Learning*, 51(2), 115–135.
- 25.19. Kohavi R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI (Vol. 14, №. 2, pp. 1137–1145)*.
- 25.20. Kohavi R., & John G. H. (1997). Wrappers for feature subset selection. *Artificial intelligence*, 97(1), 273–324.

- 25.21. Kraskov A., Stögbauer H., & Grassberger P. (2004). Estimating mutual information. *Physical Review E*, 69(6), 066138.
- 25.22. Lattimore T., & Hutter M. (2013). No free lunch versus Occam's razor in supervised learning. In *Algorithmic Probability and Friends. Bayesian Prediction and Artificial Intelligence* (pp. 223–235). Springer.
- 25.23. Langford J. (2002). Quantitatively Tight Sample Complexity Bounds. PhD thesis, Carnegie Mellon.
- 25.24. Mohri M., Rostamizadeh A., & Talwalkar A. (2018). *Foundations of Machine Learning*. MIT Press.
- 25.25. Murray I., & Ghahramani Z. (2005). A note on the evidence and Bayesian Occam's razor.
- 25.26. Nadeau C., & Bengio Y. (2003). Inference for the generalization error. *Machine Learning*, 52(3), 239–281.
- 25.27. Nolan D., & Lang D. T. (2015). *Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving*. CRC.
- 25.28. Rissanen J. (2008). Minimum description length. *Scholarpedia*, 3(8), 6727.
- 25.29. Shalev-Shwartz S., & Ben-David S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.
- 25.30. Skiena S. S. (2017). *The Data Science Design Manual*. Springer.
- 25.31. Steinwart I., & Christmann A. (2008). *Support Vector Machines*. Springer.
- 25.32. Valentini G., & Dietterich T. G. (2004). Bias-variance analysis of support vector machines for the development of SVM-based ensemble methods. *The Journal of Machine Learning Research*, 5, 725–775.
- 25.33. Vanwinkelen G., & Blockeel H. (2014). Look before you leap: some insights into learner evaluation with cross-validation. In *JMLR: Workshop and Conference Proceedings* (pp. 1–17).
- 25.34. Witten I. H., Frank E., & Hall M. A. (2016). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.
- 25.35. Wolpert D. H. (2002). The supervised learning no-free-lunch theorems. In *Soft Computing and Industry* (pp. 25–42). Springer.

## 26. Машинное обучение: классификация

### 26.1. Введение

*Классификация* — основная задача машинного обучения, которая нужна в большинстве исследований и при работе с большинством данных. Для понимания многих алгоритмов  $A$  необходимо базовое знание статистики. Мы подробно обсудим основные алгоритмы, часто с оригинальными вариантами реализации. В меньшей степени внимание будет уделено глубокому обучению.

Для некоторых  $A$  при  $k = 2$  удобнее иметь  $y \in \{-1, 1\}$ . Многие алгоритмы и теоретические результаты заточены на работу с бинарными классификаторами, но такие результаты могут быть применимы к  $k > 2$ , по крайней мере, эвристически, потому что существуют простые методы сведения таких задач к набору бинарных.

Чтобы найти  $k$  из имеющихся данных (предположим, что представлены все):

```
template<typename DATA> int findNClasses(DATA const& data)
{
    int maxClass = -1;
    for(int i = 0; i < data.getSize(); ++i)
        maxClass = max(maxClass, data.getY(i));
    return maxClass + 1;
}
```

### 26.2. Стратификация по метке класса

Некоторые  $A$  чувствительны к порядку примеров. В частности, нам бы хотелось избежать сортировки по  $y$ . Рандомизация данных перед разделением и/или использование стратифицированной выборки обычно позволяет справиться с этой проблемой. Выборки обычно достаточно, и она дает повторяемое разбиение, т. к. является детерминированной:

```
template<typename DATA> pair<PermutedData<DATA>, PermutedData<DATA> >
    createTrainingTestSetsStatified(DATA const& data,
    double relativeTestSize = 0.8)
{
    int n = data.getSize(), m = n * relativeTestSize;
    assert(m > 0 && m < n);
    pair<PermutedData<DATA>, PermutedData<DATA> > result(data, data);
    Vector<int> counts(findNClasses(data), p(n); // p для обратной совместимости
    for(int i = 0; i < n; ++i){++counts[data.getY(i)]; p[i] = i;}
    for(int i = 0; i < counts.getSize(); ++i) counts[i] *= relativeTestSize;
```

```

for(int i = 0; i < p.getSize(); ++i)
{
    int label = data.getY(p[i]);
    if(counts[label]-->0) result.first.addIndex(p[i]);
    else
    {
        result.second.addIndex(p[i]);
        p[i--] = p.lastItem();
        p.removeLast();
    }
}
return result;
}

```

В случае перекрестной проверки без стратификации необходимо рандомизировать данные в случае, если примеры отсортированы по классам, потому что их разбиение на этапы создает существенный дисбаланс классов. Реализация с использованием стратификации сложна. Она вычисляет размеры классов и отслеживает последний задействованный пример для любого класса, а для любого этапа помечает примеры, которые будут взяты для тестирования. Затем из них создается и используется тестовый набор, а исходные данные восстанавливаются. Чтобы разделение и восстановление работали, включенные номера примеров сортируются:

```

template<typename LEARNER, typename DATA, typename PARAMS>
Vector<pair<int, int> > crossValidationStratified(PARAMS const& p,
DATA const& data, int nFolds = 5)
{
    assert(nFolds > 1 && nFolds <= data.getSize());
    int nClasses = findNClasses(data), testSize = 0;
    Vector<int> counts(nClasses, 0), starts(nClasses, 0);
    PermutedData<DATA> pData(data);
    for(int i = 0; i < data.getSize(); ++i)
    {
        pData.addIndex(i);
        ++counts[data.getY(i)];
    }
    for(int i = 0; i < counts.getSize(); ++i)
        counts[i] /= nFolds; // округленная часть идет в обучающие данные
    for(int i = 0; i < counts.getSize(); ++i) testSize += counts[i];
    Vector<pair<int, int> > result;
    for(int i = 0; i < testSize; ++i)
    { // создание списка включенных тестовых примеров в порядке возрастания
        Vector<int> includedCounts(nClasses, 0), includedIndices;
        for(int j = valMin(starts.getArray(), starts.getSize());
            includedIndices.getSize() < testSize; ++j)
        {
            int label = data.getY(j);
            if(starts[label] <= j && includedCounts[label] < counts[label])
            {
                ++includedCounts[label];
                includedIndices.append(j);
            }
        }
    }
}

```

```

        starts[label] = j + 1;
    }
}
PermutedData<DATA> testData(data);
for(int j = testSize - 1; j >= 0; --j)
{
    testData.addIndex(includedIndices[j]);
    pData.permutation[includedIndices[j]] =
        pData.permutation.lastItem();
    pData.permutation.removeLast();
}
result.appendVector(evaluateLearner<int>(LEARNER(pData, p),
    testData));
// помещение тестовых данных в правильные места
if(i == nFolds - 1) break;
for(int j = 0; j < testSize; ++j)
{
    pData.addIndex(includedIndices[j]);
    pData.permutation[includedIndices[j]] =
        testData.permutation[testSize - 1 - j];
}
}
return result;
}

```

## 26.3. Оценка риска

Самая распространенная функция риска  $R_f$  — это ожидаемая ошибка =  $E[\% \text{ ошибочно классифицированных примеров}]$ . Она имеет биномиальное распределение с неизвестным средним значением, поэтому для результатов тестов используется доверительный интервал оценки Уилсона (см. главу 21. *Вычислительная статистика*). *Эмпирическая точность* =  $1 - R_{f, \text{тест}}$ . Для равновероятных классов случайная  $f$  имеет ожидаемую точность  $1/k$ .

Точность чувствительна к *дисбалансу классов*, т. е. данные не обязательно должны поступать из случайной выборки, которая хорошо представляет все классы. Дисбаланс может легко возникнуть из-за человеческой ошибки или неравных затрат на получение данных. Например, если в тестовом наборе цифровых данных было только 90 цифр 0 и 10 цифр 1, функция  $f$ , которая всегда возвращает 0, имеет точность 90%. *Критерий сбалансированной частоты ошибок* (и соответственно *сбалансированная точность*) позволяет одинаково взвешивать каждый класс, предполагая, что примеры в каждом классе являются независимыми и случайными, но данных между классами это не касается. Получается не биномиальное, а неравенство конечной выборки, потому что после нескольких алгебраических манипуляций равные веса классов оказываются эквивалентны использованию обобщенной версии неравенства на среднем от результатов примера, где класс считается в диапазоне от 0 до  $w_i = \frac{n}{n_i k}$ . Например,  $0,75a + 0,25b =$

$= 0,5 (1,5a + 0,5b)$ . Имея эти веса, используем диапазон  $\left[ 0, \sqrt{\sum \frac{w_i^2}{n}} \right]$  и применяем

обычное неравенство. Но этот метод полезен только в качестве диагностической метрики. Все алгоритмы обучения предполагают и часто сильно зависят от независимости и случайности данных. Любые сильные нарушения приведут к тому, что учащийся утратит способность к обобщению, и никакая настройка показателей оценки уже не поможет.

Не так уж много информации о том, насколько доверять каждому прогнозу. Например, в задаче распознавания цветков ириса (см. *разд. 25.3*) сорт *seposita* линейно отличается от других, поэтому прогнозирование *seposita* с меньшей вероятностью будет ошибочным, чем *versicolor* или *virginica*. Никакая  $R$  не может объяснить эту уверенность. Можно считать, что имеет место некоторая условная точность, т. е. точность предсказанного класса. Модели, с очень высокой достоверностью работающие на некоторых классах, тоже полезны, т. к. они могут доверять достаточно уверенным решениям, а остальные отправлять на проверку людям. Например, для обнаружения мошенничества с кредитными картами должна быть возможность автоматически пометать большинство транзакций как неподозрительные и использовать человеческий труд только в случае серьезных махинаций.

Некоторые  $A$  могут дать надежные оценки вероятности для отдельных примеров. Большинство из них этого не делает, потому что оценка вероятностей является более общей проблемой, и результаты, как правило, менее точны, чем прогноз  $u$ .

Если функция  $f$  возвращает только метки, матрица путаницы  $M$  дает полную информацию о производительности, т. к. содержит количество заданных примеров для любой пары меток «прогнозируемая — фактическая». Прогнозируемая метка — это координата строки, а фактическая — столбца. Требуется  $O(k^2)$  пространства для их представления и  $O(n + k^2)$  времени для вычисления  $M$ :

```
Matrix<int> evaluateConfusion(Vector<pair<int, int> > const& testResult,
    int nClasses = -1)
{
    if(nClasses == -1)
    { // вычислить nClasses, если оно неизвестно
        int maxClass = 0;
        for(int i = 0; i < testResult.getSize(); ++i) maxClass =
            max(maxClass, max(testResult[i].first, testResult[i].second));
        nClasses = maxClass + 1;
    }
    Matrix<int> result(nClasses, nClasses);
    for(int i = 0; i < testResult.getSize(); ++i)
        ++result(testResult[i].first, testResult[i].second);
    return result;
}
```

Имея  $M$ , можно вычислить множество полезных метрик, суммируя по своим строкам и столбцам:

- ◆  $t = \sum M[r, c]$  = общее число примеров в тестовом множестве;
- ◆  $a = \frac{\sum_{r \neq c} M[r, c]}{t}$  = точность (accurasy);
- ◆  $t_l = \sum M[r, l]$  = число примеров с меткой  $l$ ;



- ◆  $a_l = \frac{\sum_{r \neq l} M[r, l]}{t_l}$  = точность примеров с меткой  $l$ , также называемая *откликом*;
- ◆  $p_l = \sum M[l, c]$  = число примеров с спрогнозированной меткой  $l$ ;
- ◆  $c_l = \frac{\sum_{r \neq l} M[r, l]}{p_l}$  = уверенность прогноза метки  $l$ , т. е. тоже *точность* (precision).

Эти метрики позволяют определять производные метрики, такие как сбалансированная точность  $= \frac{\sum a_l}{k}$ . Например, рассмотрим бинарный набор данных с одним признаком и границей решения 00100|111. Здесь точность (0) < отклик (0), но точность (1) > отклик (1). При вычислении средних значений NaN вида 0/0 пропускаются. Возможна ситуация  $c_l = \text{NaN}$ , но не  $t_l = \text{NaN}$ , потому что метка всегда должна быть представлена, если только набор данных не является неполным, но тогда классификатор может не присвоить ей никаких примеров.

Доверительные интервалы рассчитываются только для точности и сбалансированной точности. Корректировка множественности не производится, т. е. предполагается, что пользователь выберет только нужные значения. Интервалы для метрик, специфичных для класса, не рассчитываются из-за многократного тестирования (рис. 26.1):

```
struct ClassifierStats
{
    double acc, bac;
    pair<double, double> accConf, bacConf;
    Vector<double> accByClass, confByClass;
    int total;
    ClassifierStats(Matrix<int> const& confusion): total(0)
    { // в строках указаны метки, в столбцах - прогнозы
        Vector<int> confTotal, accTotal;
        int k = confusion.getRows(), nBac = 0, actualK = 0;
        IncrementalStatistics accS, bacSW;
        Vector<IncrementalStatistics> precS(k);
        Vector<double> weights(k);
        for(int r = 0; r < k; ++r)
        {
            int totalR = 0;
            for(int c = 0; c < k; ++c)
            {
                totalR += confusion(r, c);
                weights[r] += confusion(r, c);
                total += confusion(r, c);
            }
            accTotal.append(totalR);
            actualK += (totalR > 0);
        }
        double M = 0;
        for(int r = 0; r < k; ++r)
        {
            weights[r] = total/weights[r]/actualK;
            IncrementalStatistics bacS;
```

```

    for(int c = 0; c < k; ++c)
    {
        int count = confusion(r, c);
        bool correct = r == c;
        while(count-->0)
        {
            accS.addValue(correct);
            basSW.addValue(correct * weights[r]);
            M += weights[r] * weights[r];
            bacS.addValue(correct);
            precS[c].addValue(correct);
        }
        accByClass.append(bacS.getMean());
    }
    M = sqrt(M/total);
    for(int c = 0; c < k; ++c)
    {
        int totalC = 0;
        for(int r = 0; r < k; ++r) totalC += confusion(r, c);
        confTotal.append(totalC);
        confByClass.append(precS[c].getMean());
    }
    acc = accS.getMean();
    accConf = wilsonScoreInterval(acc, accS.n);
    bac = basSW.getMean();
    bacConf = HoefFunc::conf(bac, basSW.n);
    bac *= M;
    bacConf.first *= M;
    bacConf.second *= M;
}
};

```

```

acc * total 1746
total 1798
Accuracy: 0.97107897664071186 95% interval: 0.96206872498450946 0.97799786769029617
Balanced Accuracy: 0.97111191719888246 95% interval: 0.95905336988373613 0.98060008111632324
Accuracy by class:
0.9943820224719101
0.99450549450549453
0.98870056497175141
0.96174863387978138
1
0.96703296703296704
0.97790055248618779
0.93296089385474856
0.94857142857142862
0.94444444444444442
Confidence by class:
0.98882681564245811
0.96276595744680848
1
0.96703296703296704
0.98369565217391308
0.9777777777777775
0.98882681564245811
0.98235294117647054
0.93785310734463279
0.92391304347826086

```

Рис. 26.1. Результаты случайного леса (обсуждается далее в этой главе) для цифровых данных

Возможно, все результаты полезны для описания производительности  $f$  на конкретной задаче. Сбалансированная точность хороша для окончательной оценки в большинстве случаев, потому что она:

- ◆ заставляет  $f$  относиться ко всем случаям одинаково, что может быть важно, даже если примеры находятся в смещенном распределении;
- ◆ равна общей точности с идеально сбалансированными классами;
- ◆ является частным случаем риска при распределении входных данных, которое предполагает, что все метки будут встречаться одинаково часто;
- ◆ не всегда  $\leq$  точности. Когда производительность на классах меньшинств лучше, средняя точность  $>$  точности.

Но сбалансированная точность имеет более высокую дисперсию, чем точность, в случае сильного дисбаланса классов, потому что примеры из второстепенных классов сильно влияют на результат.

Для оптимизации параметров и сравнения  $A$  в качестве критерия лучше использовать точность, потому что ее измерение встроено во многие алгоритмы и позволяет влиять на входное распределение путем повторной выборки или предположения о том, что  $S$  является репрезентативным. В частности, реализация перекрестной проверки использует точность как единую метрику для выбора параметров. Интуитивно понятно, что использование точности отодвигает границы (обсуждаемые далее в этой главе) дальше от большинства, а использование сбалансированной точности — от меньшинства:

```
template<typename LEARNER, typename DATA, typename PARAMS> double
crossValidation(PARAMS const& p, DATA const& data, int nFolds = 5)
{
    return ClassifierStats(evaluateConfusion(
        crossValidationStratified<LEARNER>(p, data, nFolds))).acc;
}

template<typename LEARNER, typename PARAM, typename DATA>
struct SCVRiskFunctor
{
    DATA const& data;
    SCVRiskFunctor(DATA const& theData): data(theData) {}
    double operator() (PARAM const& p) const
    { return 1 - crossValidation<LEARNER>(p, data); }
};
```

Точность и сбалансированная точность не учитывают предикторы, которые имеют среднюю достоверность для любой метки, высокую достоверность для одних меток и низкую достоверность для других. Для окончательной оценки требуется изучить  $M$ .

## 26.4. Сведение мультикласса к двум классам

Существуют два простых способа расширить бинарных учащихся на случай  $k > 2$ . Метод *один против всех* (One Vs One, OVA) работает для  $f$ , выходным результатом которых являются вероятности. Он обучает  $k$  бинарных учеников выводить 1 тогда и только тогда, когда  $y = k$ , а результатом является наибольшая выходная метка.

Метод *один против одного* (One Vs One, OVO) обучает  $O(k^2)$  учащихся для любой комбинации бинарных классификаторов и выбирает класс с наибольшим количеством голосов. Обычно этот метод лучше в следующем:

- ◆ фактический класс все время выигрывает, а остальные случайны, и тогда результат всегда правильный и достаточно надежный, если остальные случайны. Например, при сравнении цифр 2 и 3 алгоритм не будет знать, что делать с 7, и сделает случайный выбор;
- ◆ асимптотическая эффективность — хотя требуется больше итераций обучения, каждая из них намного быстрее из-за решения меньшей по объему задачи и быстрее в целом, т. к. большинство  $A$  суперлинейны;
- ◆ границы решений между любыми двумя классами оказываются проще, а дисбаланс классов создает меньше проблем. Например, в данных о цветках ириса сорт *versicolor*, который находится между *seposa* и *virginica*, не может быть отделен одной линией при использовании OVA, а с помощью OVO — может. Теоретически OVO приводит к меньшей ошибке аппроксимации в худшем случае, чем другие методы (см. [26.18]).

Реализация разделяет данные на два класса, чтобы улучшить обучение алгоритмов реального времени вроде SGD. В конвейере преобразования данных при необходимости создается этап замены меток:

```
template<typename DATA> struct RelabeledData
{
    DATA const& data;
    typedef typename DATA::X_TYPE X_TYPE;
    typedef typename DATA::Y_TYPE Y_TYPE;
    typedef typename DATA::X_RET X_RET;
    Vector<Y_TYPE> labels;
    RelabeledData(DATA const& theData): data(theData) {}
    int getSize() const {return data.getSize();}
    void addLabel(Y_TYPE y) {labels.append(y);}
    void checkI(int i) const
    {
        assert(i >= 0 && i < data.getSize() &&
            labels.getSize() == data.getSize());
    }
    X_RET getX(int i) const
    {
        checkI(i);
        return data.getX(i);
    }
    double getX(int i, int feature) const
    {
        checkI(i);
        return data.getX(i, feature);
    }
    Y_TYPE const& getY(int i) const
    {
        checkI(i);
```

```

        return labels[i];
    }
};

template<typename LEARNER, typename PARAMS = EMPTY, typename X = NUMERIC_X>
class MulticlassLearner
{ // если параметры не переданы, используется значение по умолчанию
    mutable ChainingHashTable<int, LEARNER> binaryLearners;
    int nClasses;
public:
    Vector<LEARNER const*> getLearners() const
    {
        Vector<LEARNER const*> result;
        for (typename ChainingHashTable<int, LEARNER>::Iterator i =
            binaryLearners.begin(); i != binaryLearners.end(); ++i)
            result.append(&i->value);
        return result;
    };
    template<typename DATA> MulticlassLearner(DATA const& data,
        PARAMS const& p = PARAMS()): nClasses(findNClasses(data))
    {
        Vector<Vector<int> > labelIndex(nClasses);
        for (int i = 0; i < data.getSize(); ++i)
            labelIndex[data.getY(i)].append(i);
        for (int j = 0; j < nClasses; ++j) if (labelIndex[j].getSize() > 0)
            for (int k = j + 1; k < nClasses; ++k)
                if (labelIndex[k].getSize() > 0)
                {
                    PermutedData<DATA> twoClassData(data);
                    RelabeledData<PermutedData<DATA> >
                        binaryData(twoClassData);
                    for (int l = 0, m = 0; l < labelIndex[j].getSize() ||
                        m < labelIndex[k].getSize(); ++l, ++m)
                    {
                        if (l < labelIndex[j].getSize())
                        {
                            twoClassData.addIndex(labelIndex[j][l]);
                            binaryData.addLabel(0);
                        }
                        if (m < labelIndex[k].getSize())
                        {
                            twoClassData.addIndex(labelIndex[k][m]);
                            binaryData.addLabel(1);
                        }
                    }
                    binaryLearners.insert(j * nClasses + k,
                        LEARNER(binaryData, p));
                }
    }
};

int predict(X const& x) const
{
    Vector<int> votes(nClasses, 0);

```

```

for(int j = 0; j < nClasses; ++j)
    for(int k = j + 1; k < nClasses; ++k)
    {
        LEARNER* s = binaryLearners.find(j * nClasses + k);
        if(s) ++votes[s->predict(x) ? k : j];
    }
return argMax(votes.getArray(), votes.getSize());
}

int classifyByProbs(X const& x) const
{ // для учащихся, выводящих вероятности,
  // например, нейронных сетей
  Vector<double> votes(nClasses, 0);
  for(int j = 0; j < nClasses; ++j)
      for(int k = j + 1; k < nClasses; ++k)
      {
          LEARNER* s = binaryLearners.find(j * nClasses + k);
          if(s)
          {
              double p = s->evaluate(x);
              votes[k] += p;
              votes[j] += 1 - p;
          }
      }
  return argMax(votes.getArray(), votes.getSize());
}
};

```

## 26.5. Контроль сложности

Хорошей мерой сложности  $C$  для группы  $G$ , состоящей из бинарных классификаторов для векторных данных, является *размерность*  $VC =$  максимальное  $d$  такое, что  $\exists dx$ , устроенные так, что для любого присвоения им бинарных меток существует  $f \in G$ , которые могут разделить (*разрушить*) их (рис. 26.2).

Таким образом, для двумерных линий  $d = 3$ . Для  $D$ -мерных гиперплоскостей  $d = D + 1$ . Например, проблему `xor` можно решить путем разделения по координате  $z$ .

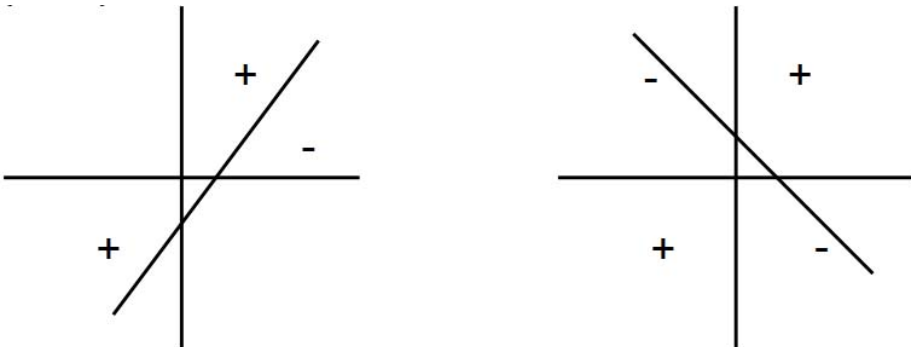


Рис. 26.2. Проблема `xor` — линия может выделить в двумерном пространстве 3 точки, но не 4

Можно ограничить ошибку обобщения функцией от  $R_{f,n}$ ,  $n$  и  $d$ . Теорема (см. [26.43]): пусть  $G$  имеет VC-размерность  $d$ . Тогда с вероятностью  $\geq 1 - p$ :

$$R_f \leq R_{f,n} + \sqrt{\frac{2d \ln(en/d)}{n}} + \sqrt{\frac{-\ln(p)}{2n}}.$$

В этом случае, если  $n/d \rightarrow \infty$  при  $n \rightarrow \infty$ ,  $R_{f,n} \rightarrow R_f$ . Даже гиперплоскости могут быть приспособлены для работы с  $D \approx n$ . Интуиция подсказывает, что, хотя  $f \in G$  может иметь несчетное множество, каждая из них может классифицировать  $n$  примеров конечным числом способов. Оценки Оккама обычно хуже оценок размерности VC, но не всегда — например, для  $G = \{\text{синусоидальные функции}\}$  VC-размерность  $= \infty$ , но описаний довольно мало.

Для SRM можно рассмотреть  $H_i$  с  $i + 1 = d$ , возможно с  $w_i = \frac{6}{(nd)^2}$ , которые в сумме

дают 1. Тогда цель поиска:

$$R_{h,n} + \sqrt{2d \frac{\ln(en/d)}{n}} + \sqrt{\frac{2 \ln(\pi p) - \ln(6p)}{2n}}.$$

Третий член пренебрежимо мал по сравнению со вторым для разумных значений  $p$ . Если знать  $d$ , формулу легко вычислить, но это применимо только к  $k = 2$ , и оценка нечеткая, если  $n$  не достаточно велико, чтобы противостоять пессимизму  $d$ . В частности, выбор модели с использованием теоретических верхних границ, основанных на размерности VC, менее эффективен, чем перекрестная проверка (см. [26.28]).

*Отступ* — это расстояние от пятна примеров определенного класса до границ разделения. Например, в случае 2D-данных и линии, которая их разделяет, отступ — это наименьшее расстояние от любого примера до линии. Интуитивно понятно, что чем больше отступ, тем точнее разделение, поэтому в некоторых случаях можно ограничить риск как функцию отступа, что обычно дает гораздо лучшие оценки, чем использование  $d$  или ORM. Очень сложные  $f$  могут иметь большие запасы и гораздо меньшие границы риска, чем предполагают границы сложности. Многие успешные алгоритмы неявно или явно тем или иным образом максимизируют прибыль. Тем не менее для известного  $A$  отступы не полностью объясняют  $R_f$ . Они эквивалентны более общей декомпозиции смещения — дисперсии (см. [26.20]), но в большинстве случаев более интуитивно понятны.

*Граничные примеры* находятся близко к границам разделения и используются при формировании границ. Шум меток в них является более серьезной проблемой, чем во внутренних примерах, из-за влияния на отступы (см. [26.25]). Невозможно обнаружить шум без достаточной поддержки правильных примеров.

Сложность границы класса является источником ошибки аппроксимации. Она может иметь сколь угодно большую колмогоровскую сложность. Существуют метрики сложности данных (см. [26.45]), но они не кажутся полезными. Например, количество опорных векторов в SVM или размер необработанного дерева решений (оба вопроса обсуждаются далее в этой главе) кажутся более информативными и простыми. Величина линейного нарушения поля SVM является хорошей мерой линейной делимости классов.

Получается невозможный результат, который, однако, предсказуем. Теорема (см. [26.19]): для любого  $A$ , который использует  $n$  примеров для обучения  $f$  и  $\varepsilon > 0$ , существует задача с  $R_{\text{об}} = 0$  такая, что  $R_f \geq 1/2 + \varepsilon$ . Эта теорема об отсутствии бесплатных обедов (No Free Lunch, NFL) подтверждает, что для некоторых задач требуется произвольное количество примеров (это не противоречит согласованности некоторых алгоритмов). Кроме того, это предполагает, что ни один конкретный  $A$  не является лучшим во всех случаях, потому что тот, который плохо подходит для какой-то задачи, может хорошо справиться с другой. Некоторые  $A$  лучше других в большинстве практических случаев, которые допускает эта NFL.

Еще один невозможный результат интересен концептуально, но не имеет значения: теорема (см. [26.43, 26.2]): Пусть  $G$  имеет VC-размерность  $d > 1$ . Тогда для любого  $n$

такого, что  $\varepsilon = \sqrt{\frac{d}{320n}} \leq \frac{1}{64}$ , и любого  $A$ , который возвращает  $f \in G$ , существует задача

такая, что  $\Pr(R_h - \min_{f \in G} R_f > \varepsilon) \geq \frac{1}{64}$ . Это верно даже для конечных  $G$  и  $A$ , которые

запоминают все примеры, потому что примеры выбраны, и существует задача, которая содержит достаточное количество примеров из  $A$  во время обучения, чтобы выполнить оценку. Это не противоречит верхней границе риска — вы можете сделать вероятность большой  $\varepsilon$  (которую можно сделать сколь угодно малой) сколь угодно малой.

Общая проблема, связанная с нижними границами, заключается в том, что они применимы только к рассматриваемой ограниченной модели. Например, сортировка по основанию по-прежнему выполняется за время  $O(n)$ , несмотря на нижнюю границу модели сравнения  $n \lg(n)$ . Эта граница также предполагает, что  $d$  для задачи точно известно, что обычно неверно. Например, для  $G = \{\text{синусоидальная функция}\}$   $d = \infty$ , потому что синус может разбить любое количество точек, но не в том случае, если для представления частоты и амплитуды требуется  $O(1)$  битов. В целом рассмотрение числовых ограничений и границ  $X$  дает более низкое значение  $d$ . Никакие границы не препятствуют тому, чтобы ошибка оценивания равнялась 0 для большинства задач, но нижняя граница интересна с концептуальной точки зрения, поскольку всегда  $d > 0$ , даже если данные, вызывающие ее, маловероятны на практике.

## 26.6. Наивный классификатор Байеса

Пусть  $x$  состоит из независимых категориальных признаков (эквивалентно дискретным признакам с ограниченным диапазоном). По условию независимости  $\Pr(x|y) = \prod_j \Pr(\text{значение признака } j|y)$ , что называется *правдоподобием*. Нужно оценить  $\Pr(\text{значение}|y)$  по (количество примеров со значением  $\in$  классу/количество примеров  $\in$  классу). Эти отсчеты начинаются с 1, чтобы не делить на 0 и работать с априорным значением. Чтобы избежать недополнения, вероятность нужно вычислять как *логарифмическую вероятность*. Это не меняет результатов сравнения, поскольку  $\log$  монотонно увеличивается.

1. Для любого класса и значения функции инициализировать счетчик равным 1.
2. Для любого примера:
3. Для любого значения функции увеличивать счетчик, связанный с ним и  $y$ .



4. При предсказании для любого класса  $i$ :
5. Найти  $LL_i = \sum_j \ln(\text{оценка } \Pr(\text{значение объекта } j | \text{класс } i))$  с использованием счетчиков для класса  $i$ .
6. Вернуть  $\text{argmin}_i(LL_i)$ .

```

class NaiveBayes
{
    struct Feature
    {
        int count;
        LinearProbingHashTable<int, int> valueCounts;
        Feature(): count(0) {}
        void add(int value)
        {
            ++count;
            int* valueCount = valueCounts.find(value);
            if (valueCount) ++*valueCount;
            else valueCounts.insert(value, 1);
        }
        double prob(int value)
        {
            int* valueCount = valueCounts.find(value);
            return (valueCount ? 1 + *valueCount : 1) / (1.0 + count);
        }
    };
    typedef ChainingHashTable<int, Feature> FEATURE_COUNTS;
    typedef ChainingHashTable<int, FEATURE_COUNTS> CLASS_COUNTS;
    mutable CLASS_COUNTS counts;

public:
    typedef Vector<pair<int, int> > SPARSE_CATEGORICAL_X;
    static SPARSE_CATEGORICAL_X convertToSparse(CATEGORICAL_X const& x)
    {
        SPARSE_CATEGORICAL_X result;
        for (int i = 0; i < x.getSize(); ++i)
            result.append(make_pair(i, x[i]));
        return result;
    }
    void learn(SPARSE_CATEGORICAL_X const& x, int label)
    {
        for (int i = 0; i < x.getSize(); ++i)
        {
            FEATURE_COUNTS* classCounts = counts.find(label);
            if (!classCounts) classCounts = &counts.insert(label,
                FEATURE_COUNTS())->value;
            Feature* f = classCounts->find(x[i].first);
            if (!f) f = &classCounts->insert(x[i].first, Feature())->value;
            f->add(x[i].second);
        }
    }
    int predict(SPARSE_CATEGORICAL_X const& x) const
    {
        double maxLL;
    }
}

```

```

    int bestClass = -1;
    for(CLASS_COUNTS::Iterator i = counts.begin(); i != counts.end();
        ++i)
    {
        double ll = 0;
        for(int j = 0; j < x.getSize(); j++)
        {
            Feature* f = i->value.find(x[j].first);
            if(f) ll += log(f->prob(x[j].second));
        }
        if(bestClass == -1 || maxLL < ll)
        {
            maxLL = ll;
            bestClass = i->key;
        }
    }
    return bestClass;
}
};

```

Признаки редко бывают независимыми, и требуется много данных, чтобы в достаточной степени охватить каждую комбинацию признаков, классов и значений. Таким образом, наивный байесовский подход не может конкурировать с лучшими методами, за исключением некоторых проблем. Этот подход важно понимать концептуально, и он дает много преимуществ:

- ◆ обучение  $n$  примеров с  $D$  признаками,  $k$  классами и  $v$  значениями занимает время  $O(nD)$  и пространство  $O(Dkv)$ , а классификация занимает время  $O(kD)$ ;
- ◆ он очень надежен по ORM, потому что член сложности  $O\left(\sqrt{\frac{Dkv}{n}}\right)$  очень мал для больших  $n$ ;
- ◆ обучение выполняется в реальном времени, что позволяет появление новых классов, признаков и значений признаков;
- ◆ естественная обработка разреженных функций и пропущенных значений. Например, это делает наивный байесовский метод выбором для классификации электронной почты как спама.

Числовые данные можно обрабатывать, используя разделение на корзины равной ширины (рис. 26.3).

```

struct NumericalBayes
{
    NaiveBayes model;
    DiscretizerEqualWidth disc;
template<typename DATA> NumericalBayes(DATA const& data): disc(data)
    {
        for(int i = 0; i < data.getSize(); ++i) model.learn(NaiveBayes::
            convertToSparse(disc(data.getX(i)), data.getY(i));
    }
    int predict(NUMERIC_X const& x) const
    {return model.predict(NaiveBayes::convertToSparse(disc(x)));}
};

```



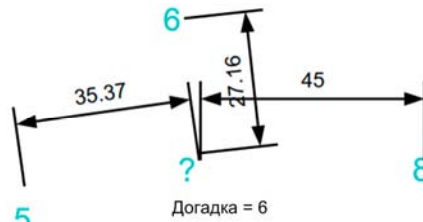


Рис. 26.4. «?» = класс ближайшего примера

В более общем случае можно использовать большинство классов  $k$  (не путать с количеством классов) ближайших соседей ( $k$ -NN) для  $k > 1$ . Если  $X$  — векторное пространство с  $D < \infty$  при  $n \rightarrow \infty$  (см. [26.19]):

- ◆ если  $k \rightarrow \infty$  и  $k/n \rightarrow 0$ ,  $k$ -NN универсально непротиворечиво, то  $k$  должно расти вместе с  $n$ ;
- ◆  $R_{1\text{-NN}} \leq R_{\text{об}}(1 - R_{\text{об}}) \leq 2R_{\text{об}}$  —  $k$ -NN с большим количеством данных имеет некоторую оптимальность 1;
- ◆ для нечетных  $k$ ,  $R_{(k+1)\text{-NN}} = R_{k\text{-NN}} \leq R_{\text{об}} + \frac{1}{\sqrt{ke}}$  используем нечетное  $k$ .

Это не выполняется, если  $D = \infty$  или для общих метрических пространств (см. [26.15]). Масштабирование теоретически не имеет значения, поскольку изменение масштаба эквивалентно изменению  $P$ . Поскольку требуется нечетное  $k$ , значение  $k = 2[\lg(n) / 2 + 1]$  кажется хорошим значением по умолчанию. Оно удовлетворяет условию согласованности и согласуется с обычно рекомендуемым  $k = 5$ .

Для фиксированных  $n$  по мере увеличения  $k$  смещение увеличивается, а дисперсия уменьшается (см. [26.20]), что также поддерживает увеличение  $k$  с  $n$ . Время выполнения запроса  $k$ -NN также увеличивается с  $k$  из-за менее эффективных границ, но это касается только больших значений  $k$ . Алгоритм есть в Сети (хотя и не реализован здесь как таковой):

```
template<typename X = NUMERIC_X, typename INDEX = VpTree<X, int, typename
    EuclideanDistance<X>::Distance> > class KNNClassifier
{
    mutable INDEX instances;
    int n, nClasses;
public:
    KNNClassifier(int theNClasses): nClasses(theNClasses), n(0) {}
    template<typename DATA> KNNClassifier(DATA const& data): n(0),
        nClasses(findNClasses(data))
    {
        for(int i = 0; i < data.getSize(); ++i)
            learn(data.getY(i), data.getX(i));
    }
    void learn(int label, X const& x){instances.insert(x, label); ++n;}
    int predict(X const& x)const
    {
        Vector<typename INDEX::NodeType*> neighbors =
            instances.kNN(x, 2 * int(log(n))/2 + 1);
```

```

Vector<int> votes(nClasses);
for(int i = 0; i < neighbors.getSize(); ++i)
    ++votes[neighbors[i]->value];
return argMax(votes.getArray(), votes.getSize());
}
};

```

Для цифровых данных метод ближайшего соседа является одним из самых точных, обеспечивающих точность 97,96% за очень короткое время.

На обучающих данных 1-NN имеет идеальную точность, но его размерность  $VC = n$  и  $|h|$  равно  $O(n)$ , поэтому ни размерность  $VC$ , ни границы ORM не гарантируют обобщения. Но дальнейшие предположения дают оценку конечной выборки. Теорема (см. [26.51]): пусть  $P(y|x)$  липшицева с константой  $c$ ,  $X$  — это векторное пространство,  $k > 1$ . Тогда для евклидова расстояния:

$$R_{k-NN} \leq \left(1 + \sqrt{\frac{8}{k}}\right) R_{\text{об}} + \frac{6c\sqrt{D+k}}{n^{1/(D+1)}}.$$

Значение  $c$  обычно неизвестно, так что оно не помогает в выборе  $k$ . Но зато возникает проклятие размерности — обучение становится более трудным при больших  $D$ . Если распределение точек равномерное ( $[0, 1]D$ ), распределение расстояний сильно смещено к максимальному расстоянию, поэтому проще выбрать неправильного соседа из-за зашумленных признаков (рис. 26.5). Но на типичных практических данных производи-

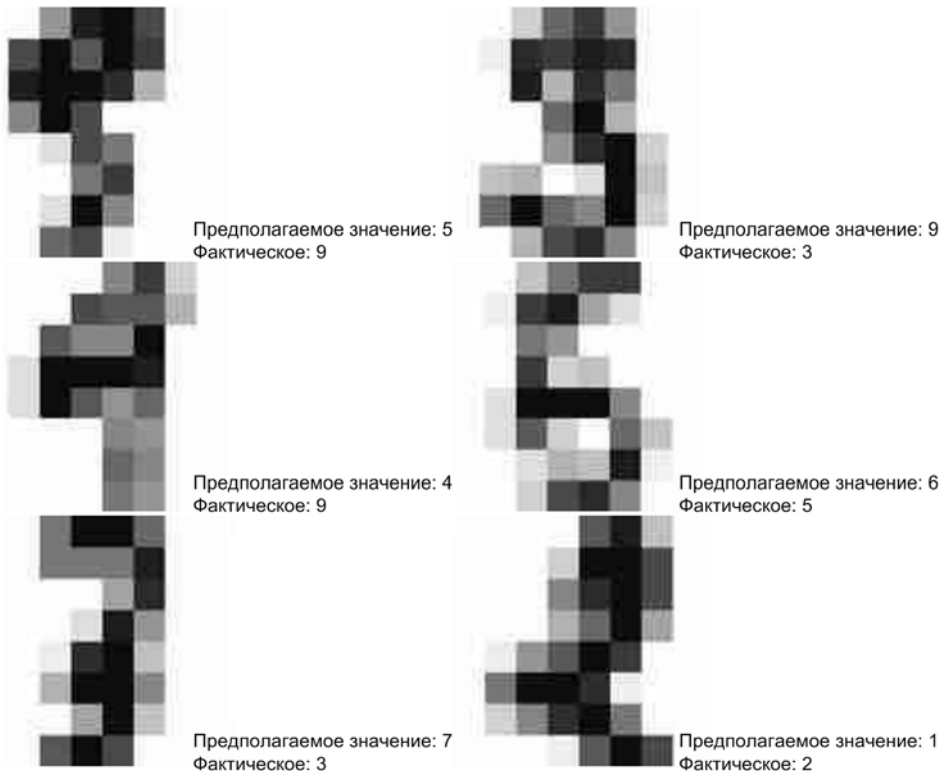


Рис. 26.5. Некоторые ошибки  $k$ -NN

тельность обычно хорошая, потому что базовое измерение этих данных намного меньше, а расстояния в его пространстве признаков не имеют такой же проблемы, потому что распределение по  $X$  далеко не однородно.

На практике  $k$ -NN имеет высокую точность даже при небольшом количестве обучающих данных, но использует слишком много памяти и может быть медленным для больших  $D$ , возможно, недостаточно качественным для приложений реального времени (в начале 1990-х для распознавания цифр  $k$ -NN со специальным расстоянием по касательной был отвергнут в пользу менее эффективной нейронной сети). Таким образом,  $k$ -NN полезен только для неvectorных данных, где можно найти хорошую функцию расстояния, такую как расстояние редактирования между строками. Основываясь на моих экспериментах, дерево VP работает быстро даже для  $n = 10^6$ , поэтому ограничение памяти кажется узким местом.

Несколько иной подход — *ближайшее среднее*: вычислить центроиды каждого класса и присвоить  $x$  классу ближайшего центроида. Он неконкурентен, но интересен, поскольку соответствует  $k$ -средним для кластеризации (см. главу 29. *Машинное обучение: другие задачи*), а его посредственная производительность ( $\approx 84\%$ ) предполагает аналогичную проблему для  $k$ -средних.

## 26.8. Дерево решений

Нужно создать бинарное дерево, в котором узлы просматривают определенные значения признаков, чтобы решить, какую ветвь выбрать, а листья содержат результирующий класс. Похожим образом работает дерево  $k$ -d (см. главу 18. *Вычислительная геометрия*). Последнее обычно использует следующую по порядку функцию со значением случайного примера в качестве разделителя, в то время как первый жадно выбирает лучшую функцию и значение. *Деревья решений легко интерпретируются*, поэтому с неизвестными данными они являются первыми  $A$ , которые можно задействовать для получения интуитивного решения.

Создание оптимального дерева, в котором используется любая разумная мера погрешности, является NP-полным (см. [26.34]). Практическое построение жадно идет сверху вниз, выбирая лучший признак и его значение в качестве точки разделения для корня и рекурсивного создания потомков.

Хорошим критерием разделения является энтропия  $= \sum H(p_i)$ , где  $p_i = \%$  примеров класса  $i$  ( $H(p) = \text{plg}(p)$ , см. главу 15. *Сжатие*). Минимизируйте общую энтропию после разделения, равную количеству левых дочерних примеров  $\times$  энтропия (левые дочерние примеры) + количество правых дочерних примеров  $\times$  энтропия (правые дочерние примеры). При одинаково точном разбиении энтропия благоприятствует тем, где в результирующих узлах одни классы более сконцентрированы, чем другие, поэтому лучше использовать ее вместо точности. Интуитивно понятно, что с каждым разбиением энтропия получает больше информации о метках, поскольку может лучше их сжимать. В конце концов, все примеры в листе имеют одну и ту же метку, энтропия  $= 0$ , и тогда разделение прекращается.

Для числового признака имеем  $n - 1$  возможных разбиений в корне. Эффективным способом выбора лучшего является сортировка значений и рассмотрение всех разбиений по одному слева направо, обновление после каждого расщепления левой и правой таб-

лиц счета для любого класса и повторное вычисление энтропии из них. Рассмотрение всех расщеплений в корне занимает время  $O(n \lg(n)D)$ .

Глубина в худшем случае равна  $n$ , дерево, достигающее этой глубины, переобучается, и его построение займет гораздо больше времени, поэтому глубину нужно ограничить. Наилучшая глубина случая  $= \lg(nClasses)$  из-за схожести примеров. Типичная глубина умеренно эффективного дерева, вероятно, основана на том факте, что большая часть данных остается в одном большом блоке, а при каждом разделении отделяется небольшая часть  $a$ . Здесь глубина  $= \frac{\lg(n)}{\lg(1/a)}$  и, поскольку максимальная глубина сильно

влияет на время выполнения, а не на сложность, ограничение, равное 50, кажется надежным. Меньшие значения не слишком улучшат время выполнения, а большие не улучшат обобщение. Ограничение глубины также делает ненужным использование стека вместо рекурсии. Для некоторых задач требуется большая глубина, например для  $D$ -мерного `хог` требуется полное бинарное дерево глубины  $D$ , но значимые данные обычно не имеют такой сложности. Большее число, такое как 100, обеспечивает небольшую безопасность ценой эффективности, когда для сложных задач создаются глубокие узлы, которые затем отсекаются (сокращение поясняется позже).

Деревья решений не дают ошибок в обучающем наборе (если только ограничение глубины не останавливает рост) и, таким образом, переобучаются. Для любой функции существует дерево решений, представляющее ее сколь угодно хорошо. Для  $k = 2$  размерность VC дерева = количеству листьев (см. [26.51]), не является конкретной функцией глубины, но ограничена функцией ограничения глубины. Операция *обрезки* заменяет поддереву мажоритарным листом.

Простым способом решить, следует ли выполнять обрезку поддерева, является знаковый тест (см. главу 21. *Вычислительная статистика*). Правильно/неправильно классифицированный пример как деревом, так и узлом — это ничья, в противном случае это победа того, кто правильно угадал. Поскольку поддерево не хуже узла, количество выигрышей для узла и поддерева составляет соответственно  $nDraws/2$  и  $(nDraws/2 + \text{разница в правильно классифицированных примерах})$ . По умолчанию при сокращении используется  $z$ -балл  $= 1$ , что немного увеличивает сокращение, но дает гораздо более простое дерево. Эксперименты говорят, что значения 0,5 и 0,25 дают наилучшие результаты, а значения  $> 1$  плохо подходят для большинства протестированных наборов данных. Таким образом, обычно статистическое значение  $z = 2$ , соответствующее 95%-ной достоверности, является слишком консервативным. Выбор  $z$ -показателя с помощью перекрестной проверки не улучшается и выполняется медленнее (рис. 26.6).

DT zcv 50	DT-z0 50	DT-z0.25 50	DT-z0.5 50	DT-z1 50	DT-z2 50
0.889	0.892	0.896	0.892	0.871	0.779

**Рис. 26.6.** Характеристики некоторых опций дерева решений. Все метрики производительности используют кривую сбалансированную точность, усредненную по нескольким наборам данных

Обрезка выполняется рекурсивно, на обратном пути после построения дерева и не раньше, чтобы избежать ситуации, подобной проблеме `хог`, когда любая первая функция оказывается неэффективна, а любая вторая — эффективна. Общий алгоритм:

1. Найти наилучшее разделение с помощью пошагового расчета.
2. Разделить данные на левую и правую части.

3. Рекурсивно выполняется на частях, пока не получится чистый узел или не превысит некоторую глубину  $m$ ;
4. Попробовать выполнить обрезку.

Реализация предполагает числовое  $x$ , имеет несколько мер безопасности, чтобы избежать проблем с неверными данными, и поддерживает режим случайного леса (обсуждается далее в этой главе):

```
struct DecisionTree
{
    struct Node
    {
        union
        {
            int feature; // для внутренних узлов
            int label; // для листовых узлов
        };
        double split;
        Node *left, *right;
        bool isLeaf() {return !left;}
        Node(int theFeature, double theSplit): feature(theFeature),
            split(theSplit), left(0), right(0) {}
    } *root;
    Freelist<Node> f;
    double H(double p) {return p > 0 ? p * log(1/p) : 0;}
    template<typename DATA> struct Comparator
    {
        int feature;
        DATA const& data;
        double v(int i) const {return data.data.getX(i, feature);}
        bool operator()(int lhs, int rhs) const {return v(lhs) < v(rhs);}
        bool isEqual(int lhs, int rhs) const {return v(lhs) == v(rhs);}
    };
    void rDelete(Node* node)
    {
        if(node)
        {
            rDelete(node->left);
            f.remove(node->left);
            rDelete(node->right);
            f.remove(node->right);
        }
    }
    typedef pair<Node*, int> RTYPE;
    template<typename DATA> RTYPE rHelper(DATA& data, int left, int right,
        int nClasses, double pruneZ, int depth, bool rfMode)
    {
        int D = data.getX(left).getSize(), bestFeature = -1,
            n = right - left + 1;
        double bestSplit, bestRem, h = 0;
        Comparator<DATA> co = {-1, data};
```



```

Vector<int> counts(nClasses, 0);
for(int j = left; j <= right; ++j) ++counts[data.getY(j)];
for(int j = 0; j < nClasses; ++j) h += H(counts[j] * 1.0/n);
int majority = argMax(counts.getArray(), nClasses),
    nodeAccuracy = counts[majority];
Bitset<> allowedFeatures;
if(rfMode)
{ // примеры признаков в случайном лесе
    allowedFeatures = Bitset<>(D);
    allowedFeatures.setAll(0);
    Vector<int> p = GlobalRNG().sortedSample(sqrt(D), D);
    for(int j = 0; j < p.getSize(); ++j) allowedFeatures.set(p[j], 1);
}
if(h > 0) for(int i = 0; i < D; ++i) // поиск лучшего признака
                                           // и разделение
    if(allowedFeatures.getSize() == 0 || allowedFeatures[i])
    {
        co.feature = i;
        quickSort(data.permutation.getArray(), left, right, co);
        int nRight = n, nLeft = 0;
        Vector<int> countsLeft(nClasses, 0), countsRight = counts;
        for(int j = left; j < right; ++j)
        { // инкрементное вращение счетчиков
            int label = data.getY(j);
            ++nLeft;
            ++countsLeft[label];
            --nRight;
            --countsRight[label];
            double fLeft = data.getX(j, i), hLeft = 0,
                fRight = data.getX(j + 1, i), hRight = 0;
            if(fLeft != fRight)
            { // равные значения не разделяются
                for(int l = 0; l < nClasses; ++l)
                {
                    hLeft += H(countsLeft[l] * 1.0/nLeft);
                    hRight += H(countsRight[l] * 1.0/nRight);
                }
                double rem = hLeft * nLeft + hRight * nRight;
                if(bestFeature == -1 || rem < bestRem)
                {
                    bestRem = rem;
                    bestSplit = (fLeft + fRight)/2;
                    bestFeature = i;
                }
            }
        }
    }
}
if(depth <= 1 || h == 0 || bestFeature == -1)
    return RTYPE(new(f.allocate())Node(majority, 0), nodeAccuracy);
// разделение примеров на левую и правую части
int i = left - 1;

```

```

    for(int j = left; j <= right; ++j)
        if(data.getX(j, bestFeature) < bestSplit)
            swap(data.permutation[j], data.permutation[++i]);
    if(i < left || i > right)
        return RTYPE(new(f.allocate())Node(majority, 0), nodeAccuracy);
    Node* node = new(f.allocate())Node(bestFeature, bestSplit);
    // рекурсивное вычисление потомков
    RTYPE lData = rHelper(data, left, i, nClasses, pruneZ, depth - 1,
        rfMode), rData = rHelper(data, i + 1, right, nClasses, pruneZ,
        depth - 1, rfMode);
    node->left = lData.first;
    node->right = rData.first;
    int treeAccuracy = lData.second + rData.second, nTreeWins =
        treeAccuracy - nodeAccuracy, nDraws = n - nTreeWins;
    // попытка не выполнять обрезку
    if(!rfMode &&
        signTestAreEqual(nDraws/2.0, nDraws/2.0 + nTreeWins, pruneZ))
    {
        rDelete(node);
        node->left = node->right = 0;
        node->label = majority;
        node->split = 0;
        treeAccuracy = nodeAccuracy;
    }
    return RTYPE(node, treeAccuracy);
}
Node* constructFrom(Node* node)
{
    Node* tree = 0;
    if(node)
    {
        tree = new(f.allocate())Node(*node);
        tree->left = constructFrom(node->left);
        tree->right = constructFrom(node->right);
    }
    return tree;
}
public:
    template<typename DATA> DecisionTree(DATA const& data, double pruneZ = 1,
        bool rfMode = false, int maxDepth = 50): root(0)
    {
        assert(data.getSize() > 0);
        int left = 0, right = data.getSize() - 1;
        PermutedData<DATA> pData(data);
        for(int i = 0; i < data.getSize(); ++i) pData.addIndex(i);
        root = rHelper(pData, left, right, findNClasses(
            data), pruneZ, maxDepth, rfMode).first;
    }
    DecisionTree(DecisionTree const& other)
        {root = constructFrom(other.root);}
    DecisionTree& operator=(DecisionTree const& rhs)
        {return genericAssign(*this, rhs);}

```

```

int predict(NUMERIC_X const& x) const
{
    assert(root); // проверка качества данных
    Node* current = root;
    while(!current->isLeaf()) current = x[current->feature] <
        current->split ? current->left : current->right;
    return current->label;
}
};

```

Результирующее дерево для данных о цветках ириса (рис. 26.7) с единственным признаком длины лепестка (признак 2) дает 100%-ную точность для *seposa* (метка 0) и *versicolor* (метка 1), но только 60% для *virginica* (метка 2).

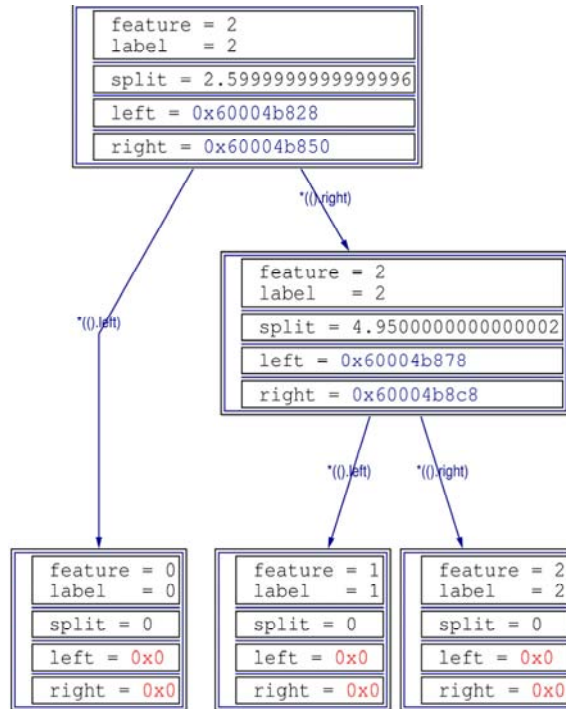
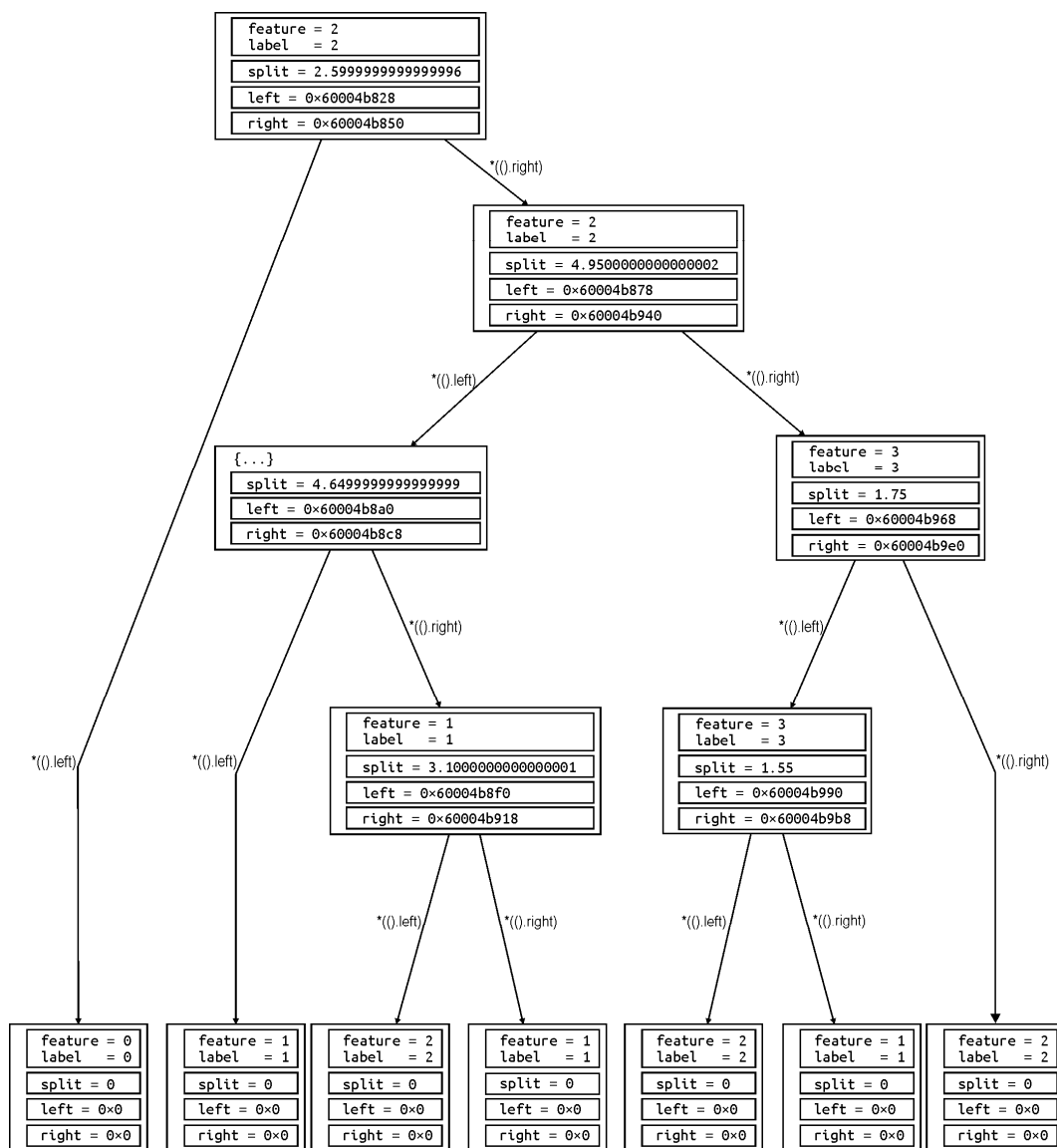


Рис. 26.7. Структура памяти дерева решений на данных о цветках ириса

Необрезанное дерево устроено немного сложнее из-за использования ширины лепестков и чашелистиков, но имеет более высокую точность (рис. 26.8).

Предполагая сбалансированное разделение, потому что для корня (и рекурсивно для любого узла с  $n$  примерами) выбор разделения занимает  $O(\text{nlg}(n)Dk)$  времени, общая работа для корня  $C(n) = 2C(n/2) + O(\text{nlg}(n)Dk)$ , поэтому основная теорема:  $C(n) = O(\text{nlg}(n)^2 Dk)$ . Поскольку максимальная глубина равна  $m$ , наихудший случай —  $O(m \text{nlg}(n) Dk)$ . Обрезка на это не влияет.

Для применения ORM можно закодировать древовидную структуру, задействовав 2 бита на узел, т. е. используя обход в глубину для записи узла, затем 1 бит для левого потомка и 1 бит для правого. Для идентификатора функции требуется  $\lceil \lg(D) \rceil$  битов и



**Рис. 26.8.** Структура памяти необработанного дерева решений на данных о цветках ириса

метка  $\lceil \lg(k) \rceil$ . Для разделения можно масштабировать данные на диапазон  $[0, 1]$  и представлять их с точностью до 0,001, на что достаточно около 10 битов. Масштабирование не влияет на логику, поскольку деревья решений масштабно-инвариантны. Имеется  $m/2$  внутренних узла и  $m/2 + 1$  листов, так что

$$|h| \approx m \left( 10 + \frac{\lg(kD)}{2} \right),$$

а это в предположении  $\lg(kD) < 20$  и  $p = 0,05$  приводит к сложности  $\approx 3,7\sqrt{\frac{m}{n}}$ .

Для  $k = 2$  получение дерева с  $d$  листьями означает (по границе размерности VC), что при  $p = 0,05$  дает коэффициент сложности:

$$\approx \sqrt{\frac{d \ln(2en/d)}{n}} + \sqrt{\frac{1,5}{n}} \approx 1,8 \sqrt{\frac{m}{n}},$$

предполагая, что  $n/d < 100$  в логарифмическом масштабе. Для любой меры сложности поддерево с тремя узлами должно отвечать неравенству  $R_{\text{корень}} < R_{\text{дерево}} + a\sqrt{\frac{1}{n}}$ , которое знаковый тест проверяет с помощью  $z$ -балла в качестве  $a$ .

Число используемых функций  $\leq n$ , и обычно выбираются одни и те же функции, что делает дерево решений хорошим средством выбора встроенных функций.

Некоторые недостатки дерева решений:

- ◆ не является универсально согласованным, потому что существуют распределения, в которых расщепление на основе энтропии не дает результатов на  $S$  размера  $\infty$  (см. [26.19]). Но практические наборы данных не имеют такой структуры;
- ◆ границы решений имеют тенденцию быть очень грубыми и состоят из кусочных функций. Когда данные линейно разделимы, это не имеет значения, если отступ шире, чем погрешность. Случайный лес сглаживает границу, обычно повышая точность;
- ◆ нестабильность (см. [26.11]). Небольшие изменения в данных приводят к большим изменениям в структуре, но малому изменению общей точности. С обрезкой проблем меньше.

## 26.9. Механизм опорных векторов

*Механизм опорных векторов* (Support Vector Machine, SVM) является теоретически хорошим классификатором с весьма приличной практической производительностью для многих типов данных. Базовая версия работает с  $k = 2$ . Если данные линейно разделимы, существует множество гиперплоскостей  $f(x) = wx + b$  таких, что для любого  $x_i \in \text{data}$ ,  $f(x_i) < 0$ , если  $y_i = -1$ , и  $> 0$  если  $y_i = 1$ . SVM вычисляет гиперплоскость, разделяющую два класса, с такими  $w$  и  $b$ , что поле = минимальное расстояние ( $x_i$ , гиперплоскость) является максимальным. То есть  $x_i$ , ближайшие к гиперплоскости, равноудалены от нее. Интуитивно понятно, что при ходьбе по минному полю лучше идти по тропинке между минами (рис. 26.9).

Расстояние ( $x$ , гиперплоскость)  $= f(x) / \|w\|$ . Поскольку можно масштабировать  $f$  для получения произвольных полей, нужно выполнить нормализацию, чтобы получить *каноническую гиперплоскость*, где  $\min|f(x)| = 1$ . Таким образом, для опорных векторов  $|f(x)| = 1$  и отступа  $= 1 / \|w\|$  ее максимизация эквивалентна минимизации  $w^2$ . Общий SVM работает для линейно неразделимых данных следующим образом:

- ◆ отображение  $x$  на расширенное пространство признаков более высокой размерности с использованием функции отображения признаков  $F$ . Гиперплоскость становится равной  $f(x) = wF(x) + b$ ;

- ♦ разрешение некоторых  $x$  находится за пределами отступов, что дает *мягкий SVM* (рис. 26.10). Используйте неактивные переменные  $\varepsilon_i$ , чтобы решить, в какой степени  $x_i$  может быть снаружи.

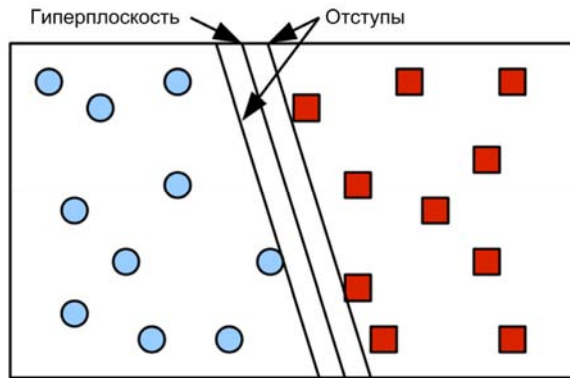


Рис. 26.9. Типичное разделение SVM

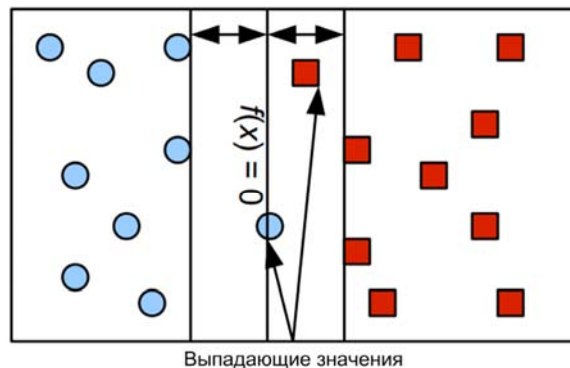


Рис. 26.10. Типичное мягкое разделение SVM

Задача оптимизации с ограничениями формулируется так (см. [26.9]):

- ♦  $\min \frac{1}{2} w^2 + C \sum \varepsilon_i$  при условии:
- ♦  $y_i f(x_i) \geq 1 - \varepsilon_i$ ;
- ♦  $\varepsilon_i \geq 0$ .

Постоянная  $C \geq 0$  определяет компромисс между размером поля и точностью. Значение  $C = 0$  означает, что ни одно  $x_i$  не может быть снаружи. Случаи в зависимости от  $y_i f(x_i)$ :

- ♦  $> 1$  — правильно классифицировано далеко от отступа, но с правой стороны;
- ♦  $= 1$  — правильно классифицировано на границе;
- ♦  $> 0$  и  $< 1$  — правильно классифицировано и внутри отступа;
- ♦  $< 0$  — неправильно классифицировано и не на той стороне отступа.

Можно добавить к VC-размерности  $d$  гиперплоскости, чтобы ограничить риск, но это не лучший метод анализа. Самый простой случай — это когда данные линейно разделимы. Теорема (см. [26.43]): предположим, что координатная плоскость смещена так, что оптимальное  $b = 0$ . Пусть  $r$  — радиус наименьшей гиперсферы, содержащей данные в расширенном пространстве и  $Q$  такие, что  $\min_i |f(x_i)| = 1$  и  $\frac{1}{\|w\|} \leq Q$  — это размер маржи. Затем для любой гиперплоскости с таким  $w$ ,  $d \leq (rQ)^2$ . Таким образом, значения параметров, а не только их количество ограничены  $d$ , а максимизация отступа является вполне хорошей целью.

## 26.10. Линейный SVM

Использование идентичности  $F$  дает линейный SVM, который ограничен линейными границами класса, но гораздо быстрее обучается и работает в сети. Ограничения SVM подразумевают, что  $\varepsilon_i = \max(0, 1 - y_i f(x_i))$ , что приводит к эквивалентной неограниченной задаче  $\frac{1}{2}lw^2 + \sum \max(0, 1 - y_i f(x_i))$  с  $l = 1/C$ . Предпочтительнее использовать штраф  $l|w|$ , потому что с помощью  $L_1$  регуляризация весов (также называемое *лассо*) имеет тенденцию производить *разреженное решение* со многими  $w_i = 0$  (см. [26.29]). Это связано с тем, что в эквивалентной задаче с ограничениями допустимая область содержит области, соответствующие некоторым переменным, равным 0, и одна из них, вероятно, достигнет наилучшего допустимого набора уровней (рис. 26.11).

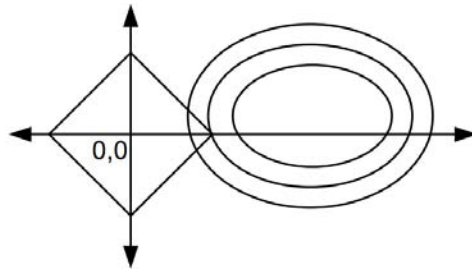


Рис. 26.11. Наилучший допустимый набор уровней

Эта задача выпукла, но недифференцируема. Использование SGD дает хорошее приближенное решение в отношении ошибки оценки (см. [26.44, 26.10]) и позволяет выполнять обучение в реальном времени. Нужно выражение, ожидаемое значение которого является субградиентом функции. Здесь субградиент представляет собой сумму, и надо использовать  $n \times$  значение примера  $i$  для случайного  $i$ . Уравнения обновления для наблюдения примера  $i$ :

- ◆  $\forall 0 \leq j \leq D, w_{ij} = r_i (w_{ij} > 0 ? 1 : -1) l - n(y_i F(x_i) > 1 ? 0 : y_i x_i)$ ;
- ◆  $b = r_i (y_i f(x_i) > 1 ? 0 : y_i)$ .

Таким образом, примеры за пределами поля увеличивают его, а примеры на правильной стороне уменьшают. Представление близких к марже примеров первым или последним дает другой ответ, поэтому порядок имеет значение.

Для параметров:

- ◆ начальная скорость обучения = 1, и такая скорость, видимо, хорошо работает для многих задач без расхождений;
- ◆ количество проходов по данным =  $\lceil 10^5/n \rceil$ , потому что SGD-сходимость асимптотически равна  $O(\sqrt{n})$ . При этом достигается относительная точность  $O(0,003)$ , что кажется разумным, и в моих экспериментах сработало хорошо. Для очень больших  $n$  выполняется только один проход, и просматривать все данные может быть необязательно. Для малых  $n$  одного прохода недостаточно;
- ◆ порядок стратифицируется по классам или случайно. Порядком по умолчанию может быть сортировка примеров по классам, поэтому стратифицированный порядок обычно лучше, но случайный может быть проще и имеет известную теоретическую производительность.

Хотя SGD хорош для обобщения, он не предназначен для определения того, какие коэффициенты равны 0, чтобы получить разреженное решение из-за медленной сходимости. Последнее является активной темой исследований (см. [26.29]), но координатный спуск (см. главу 24. Численная оптимизация) обычно хорошо работает в пакетном режиме. Для эффективной оптимизации одного веса за раз используйте  $f(x_i) = \text{sum}_i + w_{ij}x_{ij}$  или  $\text{sum}_i + b$ , где  $\text{sum}_i$  представляет собой сумму неизменных весов. Таким образом, можно оценить влияние переменной  $j$  за время  $O(n)$  независимо от  $D$ . Одномерные оптимизации решаются с помощью бреккетинга и золотого сечения (см. главу 24. Численная оптимизация).

Координатный спуск не гарантирует, что он сойдется или будет быстрым, но обычно никаких проблем не возникает. Детали реализации:

- ◆ бюджет оценки =  $10^5$ ;
  - ◆ используйте значения по умолчанию для точности завершения и начального размера шага. Они хорошо работают для обращения нерелевантных переменных в ноль;
  - ◆ поскольку SGD хорош с точки зрения начальной сходимости и надежности, когда координатный спуск застревает, его стоит использовать в первую очередь:
1. На любом проходе нужно обрабатывать каждый пример с помощью SGD.
  2. Запустите координатный спуск, чтобы получить лучшую точность.
  3. Для предсказания при заданном  $x$  вычислите запас гиперплоскости и соответствующим образом выполните классификацию.

```
class BinaryLSVM
```

```
{
    Vector<double> w;
    double b, l;
    int learnedCount;
    static int y(bool label){return label * 2 - 1;}
    static double loss(double fxi, double yi){return max(0.0, 1 - fxi * yi);}
    double f(NUMERIC_X const& x) const{return dotProduct(w, x) + b;}
    template<typename DATA> class GSLLFuncion
    {
        DATA const& data;
        mutable Vector<double> sums;
```



```

    Vector<double> &w;
    double &b, l;
    int j, D;
    mutable int evalCount;
    double getSumI(double wj, int i) const
    {
        return sums[i] + (j == D ? wj - b : (wj - w[j]) * data.getX(i, j));
    }
public:
    GSL1Funcor(DATA const& theData, double& theB, Vector<double>& theW,
        double theL): data(theData), sums(theData.getSize(), theB),
        w(theW), b(theB), l(theL), j(0), D(getD(data)), evalCount(0)
    {
        for(int i = 0; i < data.getSize(); ++i)
            sums[i] += dotProduct(w, data.getX(i)) + b;
    }
    void setCurrentDimension(int theJ)
    {
        assert(theJ >= 0 && theJ < D + 1);
        j = theJ;
    }
    int getEvalCount() const {return evalCount;}
    int getSize() const {return D + 1;}
    Vector<double> getX() const
    {
        Vector<double> x = w;
        x.append(b);
        return x;
    }
    double getXi() const {return j == D ? b : w[j];}
    double operator()(double wj) const
    {
        ++evalCount;
        double result = j == D ? 0 : l * abs(wj);
        for(int i = 0; i < data.getSize(); ++i)
            result += loss(getSumI(wj, i), y(data.getY(i)));
        return result/data.getSize();
    }
    void bind(double wjNew)
    { // сперва обновляются суммы
        for(int i = 0; i < data.getSize(); ++i)
            sums[i] = getSumI(wjNew, i);
        (j == D ? b : w[j]) = wjNew;
    }
};

public:
    BinaryLSVM(pair<int, double> const& p): w(p.first), l(p.second), b(0),
        learnedCount(0) {}
    template<typename DATA> BinaryLSVM(DATA const& data, double theL,
        int nGoal = 100000, int nEvals = 100000): l(theL), b(0),
        w(getD(data)), learnedCount(0)

```

```

{ // SGD используется первым
    for(int j = 0; j < ceiling(nGoal, data.getSize()); ++j)
        for(int i = 0; i < data.getSize(); ++i)
            learn(data.getX(i), data.getY(i), data.getSize());
    // затем координатный спуск
    GSL1Functor<DATA> f(data, b, w, l);
    unimodalCoordinateDescent(f, nEvals);
}

int getLearnedCount(){return learnedCount;}
void learn(NUMERIC_X const& x, int label, int n = -1)
{ // в реальном времени используется только SGD
    if(n == -1) n = learnedCount + 1;
    double rate = RMRate(learnedCount++), yl = y(label);
    for(int i = 0; i < w.getSize(); ++i)
        w[i] -= rate * (w[i] > 0 ? 1 : -1) * 1/n;
    if(yl * f(x) < 1)
    {
        w -= x * (-yl * rate);
        b += rate * yl;
    }
}

int predict(NUMERIC_X const& x) const{return f(x) >= 0;}
template<typename MODEL, typename DATA>
static double findL(DATA const& data)
{ // используется также для регрессии
    int lLow = -15, lHigh = 5;
    Vector<double> regs;
    for(double j = lHigh; j > lLow; j -= 2) regs.append(pow(2, j));
    return valMinFunc(regs.getArray(), regs.getSize(),
        SCVRiskFunctor<MODEL, double, DATA>(data));
}
};

```

Никаких упрощений модели не делается, хотя некоторые коэффициенты будут иметь значения на уровне машинной точности. Это и любая проверка модели оставлены на усмотрение пользователя, потому что некоторые допуски здравого смысла, такие как  $10^{-4}$ , не обязательно должны работать для всех предметных областей.

Координатный спуск работает медленно, когда данные хранятся на диске несмежным образом. Здесь данные хранятся в  $D$  смежных частей, каждая из которых содержит все примеры из одной функции. Даже в памяти, если есть лишняя, следует использовать буфер для устранения замедления из-за повторной оценки конвейера данных, особенно масштабирования (рис. 26.12, 26.13).

LSVM не может решить проблему *xor* с  $k = 2$ , потому что не существует линии, которая может выполнить разделение. Несмотря на такую нехватку возможностей моделирования, это полезный метод для многих задач, где  $D$  велико или классы в основном разделимы.

Для  $k > 2$  используется OVO. Для лучшей оценки примените одиночное  $l$  для всех бинарных учащихся. Выберите  $C$ , используя поиск по сетке с перекрестной проверкой

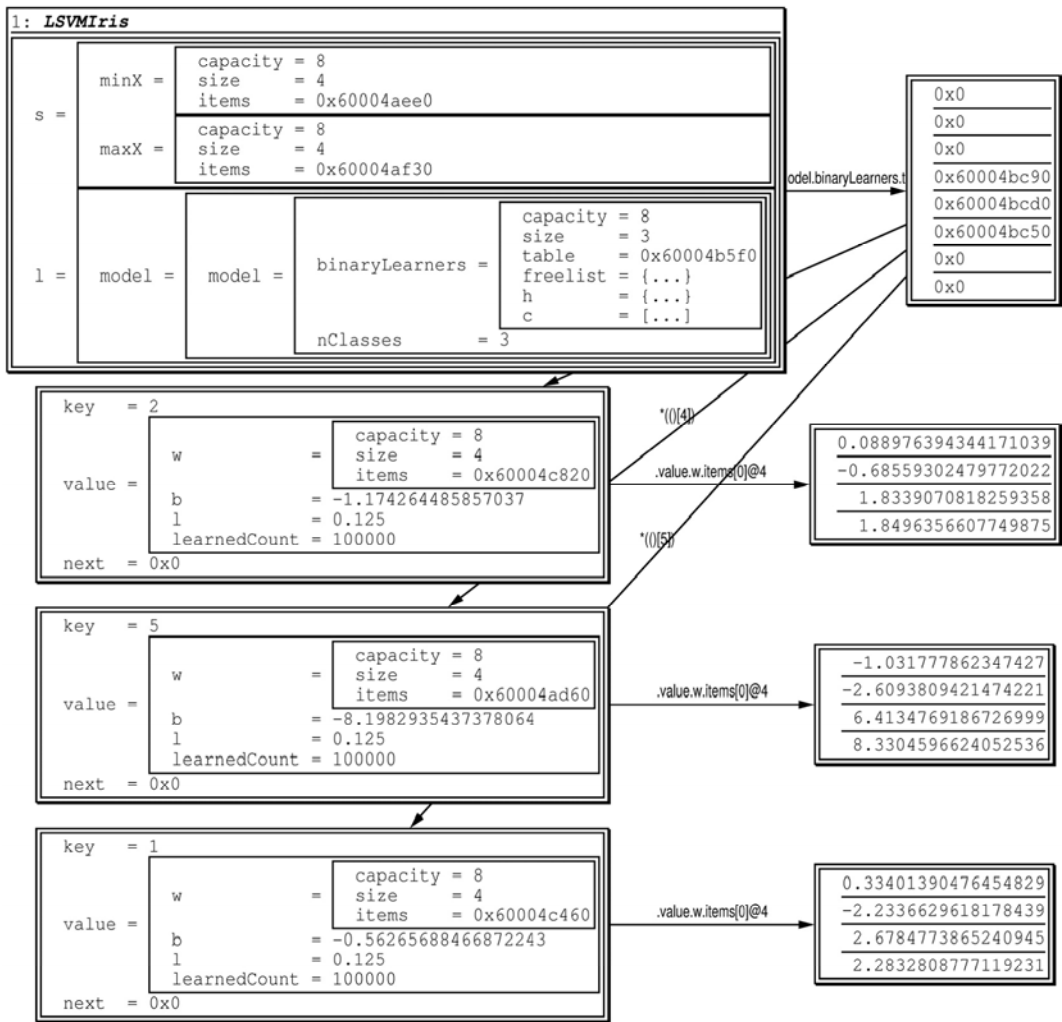


Рис. 26.12. Структура памяти SVM для данных о цветках ириса

SLSVM100k	SLSVMcd100	mqLSVMsgd100kcd100	SLSVMsgd100kcd100
0.874	0.780	0.895	0.917

Рис. 26.13. Сравнение производительности некоторых вариантов (S означает масштабирование диапазона по умолчанию [0, 1])

среди значений  $2^i$ , с нечетным  $i \in [-15, 5]$ . Для лучшего обобщения сначала начните с большого  $l$ :

```
template<typename MODEL, typename DATA>
static double findL(DATA const& data)
{
    int lLow = -15, lHigh = 5;
    Vector<double> regs;
```

```

        for(double j = lHigh; j > lLow; j -= 2) regs.append(pow(2, j));
        return valMinFunc(regs.getArray(), regs.getSize(),
            SCVRiskFunctor<MODEL, double, DATA>(data));
    }
}

struct NoParamsLSVM
{
    typedef MulticlassLearner<BinaryLSVM, double> MODEL;
    MODEL model;
    template<typename DATA> NoParamsLSVM(DATA const& data): model(data,
        BinaryLSVM::findL<MODEL, DATA>(data)) {}
    int predict(NUMERIC_X const& x) const{return model.predict(x);}
};

typedef ScaledLearner<NoParamsLearner<NoParamsLSVM, int>, int> SLSVM;

```

## 26.11. Ядро SVM

Для задач вроде `xor` существуют нелинейные сепараторы. Чтобы ввести нелинейность, включите искусственные признаки, которые являются нелинейными функциями исходных признаков, используя ядра. Регуляризация  $L_1$  удалит объекты в расширенном пространстве, не затрагивая исходные пространства, поэтому вместо этого можно минимизировать отступы.

При использовании ядер задача по-прежнему выпукла, потому что и цель, и ограниченные выпуклы, поэтому имеет место сильная двойственность (см. [26.43]). Лагранжиан

$$L = \frac{1}{2} w^2 + C \sum \varepsilon_i - \sum a_i (y_i f(x_i) - 1 + \varepsilon_i) - \sum c_i \varepsilon_i.$$

Условия ККТ в оптимуме (см. главу 24. Численная оптимизация):

1.  $0 = \nabla_w L = w - \sum a_i y_i (x_i) \rightarrow w = \sum a_i y_i f(x_i).$
2.  $0 = \nabla_b L = -\sum a_i y_i \rightarrow \sum a_i y_i = 0.$
3.  $0 = \nabla_{\varepsilon_i} L = C - a_i - c_i.$
4.  $0 = a_i (y_i f(x_i) - 1 + \varepsilon_i).$
5.  $0 = c_i \varepsilon_i.$
6.  $a_i, c_i \geq 0.$

(3) и (6) подразумевают  $0 \leq a_i \leq C$ . Случаи:

- ◆  $a_i = C \rightarrow 0 < \varepsilon_i = 1 - y_i f(x_i) \rightarrow$  внутри или снаружи отступа;
- ◆  $a_i = 0 \rightarrow \varepsilon_i = 0 \rightarrow$  правильно классифицировано и выходит за пределы поля;
- ◆  $0 < a_i < C \rightarrow 0 = \varepsilon_i = 1 - y_i f(x_i) \rightarrow$  правильно классифицировано на границе, можно вычислить  $b$ .

Подстановка двойственных значений для  $w$  приводит к:

$$L = \frac{1}{2} (\sum a_i y_i f(x_i))^2 - \sum a_j y_j f(x_j) (\sum a_i y_i f(x_i)) + \sum a_i - b \sum a_i y_i + \sum \varepsilon_i (C - a_i - c_i).$$

Первые две части объединяются, а последние две равны 0, поэтому

$$L = \sum a_i - \frac{1}{2} \sum a_i y_i a_j y_j K_{ij}.$$

Для любого  $i$  такого, что  $0 < a_i < C$ ,  $b = y_i - \sum a_j y_j K_{ij}$ . Напрямую используя  $K$  и упрощая, получаем:

♦  $\max V(a) = \sum a_i - \frac{1}{2} \sum a_i y_i a_j y_j K_{ij}$  при условии, что:

♦  $0 \leq a_i \leq C$ ;

♦  $\sum a_i y_i = 0$ .

Имея оптимальную  $a_i^*$  для этой задачи квадратичного программирования,  $f(x) = \sum a_i y_i^* K(x_i, x) + b$ . Опорные векторы равны  $x_i$ , для которых  $a_i^* > 0$ . Численно используется типичная точность  $\sqrt{\epsilon_{machine}}$  (такое маленькое значение может показаться неожиданным, но большинство примеров во время оптимизации не затрагивается). Решение сохраняет их с соответствующими  $x_i$  и  $y_i$ .

Особая структура задачи позволяет получить более простое и эффективное решение, чем общее квадратичное программирование. Пусть  $d_i = y_i a_i$  (также  $a_i = y_i d_i$ ) и  $[L_i, H_i] = [0, C]$ , если  $y_i = 1$ , и  $[-C, 0]$  в противном случае. Тогда:

♦  $\max \sum y_i d_i - \frac{1}{2} \sum d_i d_j K_{ij}$  при условии:

♦  $L_i \leq d_i \leq H_i$ ;

♦  $\sum d_i = 0$ .

По теореме Осумы о разложении (см. [26.9]) итеративная оптимизация по любому подмножеству из  $\geq$  двух переменных сходится к решению. В частности, для любых  $i$  и  $j$   $d_i$  и  $d_j$  находятся в ящике, определяемом ограничениями (рис. 26.14).

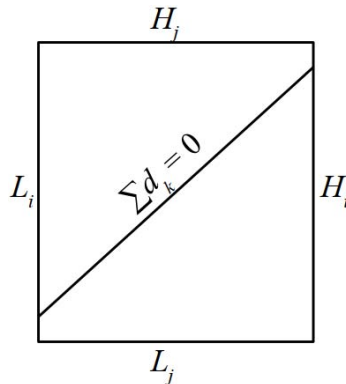


Рис. 26.14. Ограничения для двух переменных

Пусть  $g_k = \nabla V(d_k)[k] = y_k - \sum d_l K_{kl}$  (« $\frac{1}{2}$ » исчезает из-за симметрии). Если решение не является оптимальным, существует  $l$  такое, что  $d_l < H_i$ , и  $j$  такие, что  $d_j > L_j$ , так что можно увеличить  $V$ , увеличив  $d_l$  и уменьшив  $d_j$  на некоторый шаг  $s$ . Без ограничений

ящика  $d_k(s) = d_k + s_k$ , где  $s_k = \begin{cases} s, & \text{если } k = i \\ -s, & \text{если } k = j \\ 0 & \end{cases}$ . Тогда

$$V(s) = C_1 + \sum_{k \in \{i, j\}} d_k(s) \left( y_k - \sum_{l \notin \{i, j\}} d_l(s) K_{kl} \right) - \frac{1}{2} \sum_{k, l \in \{i, j\}} d_k(s) d_l(s) K_{kl} =$$

$$C_2 + \sum_{k \in \{i, j\}} s_k \left( y_k - \sum_{l \notin \{i, j\}} d_l K_{kl} \right) - \frac{1}{2} \sum_{k, l \in \{i, j\}} (s_k d_l + s_l d_k) K_{kl} - \frac{1}{2} \sum_{k, l \in \{i, j\}} s_k s_l K_{kl}.$$

Симметричный случай:

$$\frac{1}{2} \sum_{k, l \in \{i, j\}} (s_k d_l + s_l d_k) K_{kl} = \sum_{k \in \{i, j\}} s_k \left( \sum_{l \in \{i, j\}} d_l K_{kl} \right),$$

поэтому можно скомбинировать  $l$  разделов:

$$V(s) = C_2 + \sum_{k \in \{i, j\}} s_k g_k - \frac{1}{2} \sum_{k, l \in \{i, j\}} s_k s_l K_{kl}.$$

Максимизируйте  $V(s)$ , когда

$$0 = \frac{\partial V}{\partial s} = g_i - g_j - s(K_{ii} - 2K_{ij} + K_{jj}).$$

Тогда

$$s_{\text{opt}} = \frac{g_i - g_j}{K_{ii} - 2K_{ij} + K_{jj}}.$$

Эта формула лежит в основе *алгоритма SMO*:

1. Использовать двойное представление.
2. Инициализировать коэффициенты поддержки равными 0, а их градиенты равными 1.
3. Для некоторого числа итераций:
4.     Выбрать две переменные для оптимизации, используя жадный выбор.
5.     Проверить сходимость.
6.     Аналитически решить полученную задачу оптимизации.
7.     Обновить градиенты.
8. Сохранить ненулевые коэффициенты поддержки и соответствующие векторы.
9. Спрогнозировать  $x$ , вычислить отступ и выбрать класс максимального отступа.

В силу положительной определенности матрицы ядра знаменатель  $> 0$  (на самом деле, это квадрат расстояния  $L_2$  между примерами в расширенном пространстве), если только  $K$  не является недопустимым, т. к. в этом числовая отмена достаточно сильна, и примеры с одним и тем же  $x$  имеют разные  $y$ . Для надежности используйте значение-заполнитель  $s_{\text{opt}} = \infty$ , что сделает один такой пример опорным вектором (альтернатива — удалить оба значения). Принимая во внимание ограничения

ящика,  $s = \min(H_i - d_i, d_j - L_j, s_{\text{opt}})$ . Чтобы обновить  $g$ , используйте формулу  $\frac{\partial g_k}{\partial s} = -(K_{ik} - K_{jk})$ . Самый простой выбор  $i$  и  $j$  таков, что зазор  $= g_i - g_j$  максимален (это называется *эвристикой максимальной нарушающей пары*). Таким образом, решение является оптимальным, если зазор  $= 0$ , но лучше использовать некоторую малую точность, обычно 0,001 (см. [26.16]). Поскольку  $b = g_k$ , для любого  $k$  такого, что  $L_k < d_k < H_k$ ,  $g_j < g_k < g_i$ . Так что можно установить  $b$  равным  $g_i$  или  $g_j$  (для численной стабильности возможен вариант  $\frac{g_i + g_j}{2}$  при завершении. Не удастся найти пра-

вильные  $i$  и  $j$ , только если все  $y$  одинаковы, и в этом случае  $w = 0$  и для любого  $kb = y_k$  является решением, поскольку для любого  $k \varepsilon_k = 0$ .

На практике узким местом во время выполнения является вычисление  $K$ , значения которого кешируются для повышения эффективности. Для этого подходит кеш LRU с ограничением по памяти, но для простоты реализация кеширует все значения. Асимптотически число опорных векторов равно  $O(n)$  при правильно масштабированном  $C$  (см. [26.9]). SMO сходится к любой точности за конечное число итераций (см. [26.9]), но для безопасности реализация устанавливает бюджет  $\max(10n, 10\,000)$ . Идея состоит в том, что каждая итерация обновляет два (обычно) опорных вектора, поэтому небольшие наборы данных получают большую точность, а более крупные имеют  $O(n)$  опорных векторов, хотя обычно намного меньше, чем  $n$ :

```
template<typename KERNEL = GaussianKernel, typename X = NUMERIC_X> struct SVM
{
    Vector<X> supportVectors;
    Vector<double> supportCoefficients;
    double bias;
    KERNEL K;
    template<typename DATA> double evalK(LinearProbingHashTable<long long,
        double>& cache, long long i, long long j, DATA const& data)
    {
        long long key = i * data.getSize() + j;
        double* result = cache.find(key);
        if(result) return *result;
        else
        {
            double value = K(data.getX(i), data.getX(j));
            cache.insert(key, value);
            return value;
        }
    }
    int makeY(bool label){return label * 2 - 1;}
    double lowDiff(bool label, double C, double d){return label ? d : d + C;}
    double highDiff(bool label, double C, double d){return label ? C - d: d;}
public:
    template<typename DATA> SVM(DATA const& data, pair<KERNEL, double> const&
        params, int maxRepeats = 10, int maxConst = 10000): K(params.first)
    {
        double C = params.second;
```

```

assert(data.getSize() > 0 && C > 0);
bias = makeY(data.getY(0)); // случай, если есть всего 1 класс
LinearProbingHashTable<long long, double> cache;
int n = data.getSize(), maxIters = max(maxConst, n * maxRepeats);
Vector<double> d(n, 0), g(n);
for(int k = 0; k < n; ++k) g[k] = makeY(data.getY(k));
while(maxIters--)
{ // выбор направлений с помощью самой нарушающей пары
    int i = -1, j = -1; // i может увеличиваться, а j уменьшаться
    for(int k = 0; k < n; ++k)
    { // усреднение для устойчивости
        if(highDiff(data.getY(k), C, d[k]) > 0 && (i == -1 ||
            g[k] > g[i])) i = k;
        if(lowDiff(data.getY(k), C, d[k]) > 0 && (j == -1 ||
            g[k] < g[j])) j = k;
    }
    if(i == -1 || j == -1) break;
    bias = (g[i] + g[j])/2; // стремление к стабильности
    // проверка оптимального условия
    double optGap = g[i] - g[j];
    if(optGap < 0.001) break;
    // вычисление минимума по направлению и ограничений ящика
    double denom = evalK(cache, i, i, data) -
        2 * evalK(cache, i, j, data) + evalK(cache, j, j, data),
        step = min(highDiff(data.getY(i), C, d[i]),
            lowDiff(data.getY(j), C, d[j]));
    // сокращение шага к ограничениям ящика, если нужно, проверка
    // ошибок вычислений в вычислении ядра или дублирования данных,
    // если ошибочные точки выпадают из ящика
    if(denom > 0) step = min(step, optGap/denom);
    // обновление коэффициентов и градиента опорного вектора
    d[i] += step;
    d[j] -= step;
    for(int k = 0; k < n; ++k) g[k] += step *
        (evalK(cache, j, k, data) - evalK(cache, i, k, data));
} // определение опорных векторов
for(int k = 0; k < n; ++k) if(abs(d[k]) > defaultPrecEps)
{
    supportCoefficients.append(d[k]);
    supportVectors.append(data.getX(k));
}
}

int predict(X const& x) const
{
    double sum = bias;
    for(int i = 0; i < supportVectors.getSize(); ++i)
        sum += supportCoefficients[i] * K(supportVectors[i], x);
    return sum >= 0;
}
};

```



Время выполнения равно  $O(\text{количество итераций} \times n)$ . Радиальное базисное ядро усредняет только близлежащие примеры полей из-за экспоненциального уменьшения веса на расстоянии. Небольшое значение  $C$  приводит к меньшему количеству опорных векторов, потому что отступ менее приспособлен к изолированным примерам.

В нелинейно разделимых данных невозможно ограничить размерность VC в зависимости от маржи, но обычно точный анализ дает аналогичные границы. Пусть потери  $p$ -отступа равны:

$$L_p(x) = \begin{cases} 1, & \text{если } x < 0 \\ 0, & \text{если } x \geq 0 \\ 1 - x/p, & \text{если } 0 \leq x < p \end{cases}.$$

Тогда на обучающих данных проводится эмпирическая граница риска  $R_p$ . Теорема (см. [26.43]): пусть  $b = 0$ , а  $\text{Tr}(G)$  = след матрицы ядра. Тогда для любых  $p, q > 0$  с вероятностью  $\geq 1 - q$ :

$$R(h) \leq R_p + 2\sqrt{\frac{\text{Tr}(G)}{m}} \frac{Q}{p} + 3\sqrt{\frac{\ln(2/q)}{2m}}.$$

В частности, значение  $p = 1$  полезно, потому что *hinge loss* ограничивает  $p$ -потерю, и  $R_1 \leq \sum \epsilon_i$ . Это оправдывает дизайн SVM как попытку минимизировать сумму нарушений отступа и максимизировать отступ. Условие  $b \neq 0$  является лишь незначительным нарушением предположения на практике (см. [26. 56]) и в теории, поскольку другие параметры должны компенсировать это.

В некотором смысле использование ядер оптимально, потому что оно позволяет выразить любую граничную форму с достаточной поддержкой примеров, основываясь на предположении о многообразии.

## 26.12. Мультиклассовый нелинейный SVM

Используйте метод OVO из-за его общих преимуществ, а также потому, что ему нужно меньше памяти для хранения кеша ядра:

```
template<typename KERNEL = GaussianKernel, typename X = NUMERIC_X>
class MulticlassSVM
{
    // для ускорения нужен буфер
    typedef pair<KERNEL, double> P;
    MulticlassLearner<BufferLearner<SVM<KERNEL, X>,
        InMemoryData<X, int>, P>, P> mcl;
public:
    template<typename DATA> MulticlassSVM(DATA const& data,
        pair<KERNEL, double> const& params): mcl(data, params) {}
    int predict(X const& x) const { return mcl.predict(x); }
};
```

Хорошим, но медленным подходом к выбору  $u$  и  $C$  является экспоненциальный поиск по сетке (см. главу 24. Численная оптимизация) с перекрестной проверкой. При прочих равных хочется получить малое значение  $C$  и большой  $u$ . В LIB-SVM используется сет-

ка с  $2^i$  точками с нечетным  $i \in [-15, 3]$  для  $u$  и  $[-5, 15]$  для  $C$ , при этом необходимо перебрать 110 значений (см. [26.9]). Метод хорошо работает на практике, но основан на предположении о том, что данные смасштабированы так, что этот диапазон является разумным. Однако мои эксперименты показывают, что простой дискретный поиск по компасу с 10 оценками (см. главу 24. Численная оптимизация) дает аналогичную точность и работает в несколько раз быстрее (рис. 26.15).

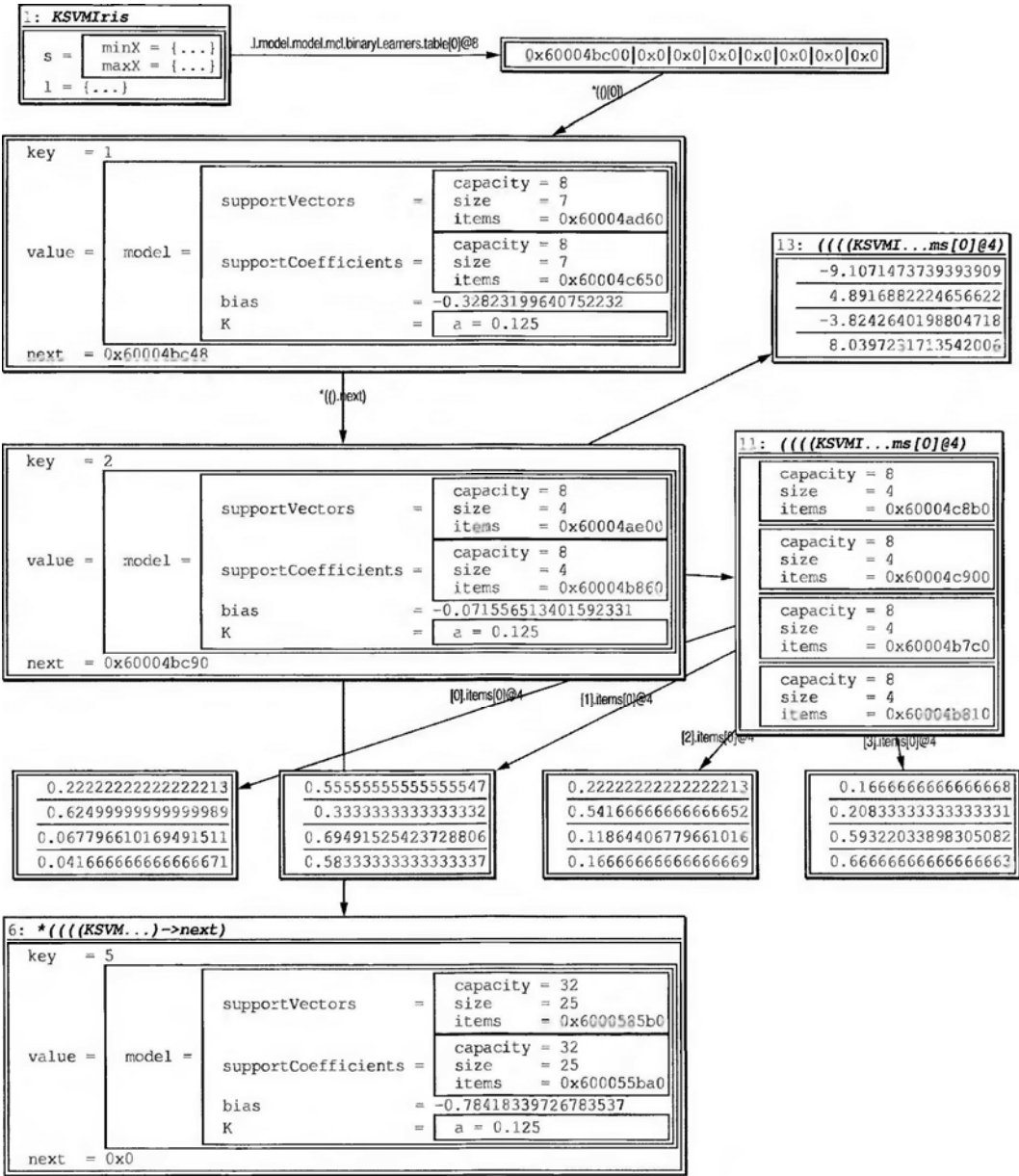


Рис. 26.15. Структура памяти K SVM на данных о цветках ириса

```
struct NoParamsSVM
{
    MulticlassSVM<> model;
    struct CVSVMFuncutor
    {
        typedef Vector<double> PARAMS;
        MulticlassSVM<> model;
        template<typename DATA> CVSVMFuncutor(DATA const& data,
            PARAMS const& p):
            model(data, make_pair(GaussianKernel(p[0]), p[1])) {}
        int predict(NUMERIC_X const& x) const {return model.predict(x);}
    };
    template<typename DATA> static pair<GaussianKernel, double>
        gaussianMultiClassSVM(DATA const& data, int CLow = -5,
            int CHigh = 15, int yLow = -15, int yHi = 3)
    {
        Vector<Vector<double> > sets(2);
        for(int i = yLow; i <= yHi; i += 2) sets[0].append(pow(2, i));
        for(int i = CLow; i <= CHigh; i += 2) sets[1].append(pow(2, i));
        Vector<double> best = compassDiscreteMinimize(sets,
            SCVRiskFuncutor<CVSVMFuncutor, Vector<double>, DATA>(data),
            10);
        return make_pair(GaussianKernel(best[0]), best[1]);
    }
    template<typename DATA> NoParamsSVM(DATA const& data): model(data,
        gaussianMultiClassSVM(data)) {}
    int predict(NUMERIC_X const& x) const {return model.predict(x);}
};
typedef ScaledLearner<NoParamsLearner<NoParamsSVM, int>, int> SSVM;
```

Авторы LIBSVM рекомендуют масштабировать признаки на диапазон [0, 1] (см. [26.31]). Ядерный SVM с выбором параметров работает медленно и плохо масштабируется при больших  $n$  (рис. 26.16). Таким образом, его использование ограничено малым или средним  $n$ .

SSVM_10n10k_001_seps_c10	SSVM_10n10k_001_seps_r20	SSVM_10n10k_001_seps_g	MQSVM_10n10k_001_seps_c10
0.940	0.911	0.944	0.920

Рис. 26.16. Сравнение производительности вариантов настроек параметров и масштабирования

## 26.13. Нейронная сеть

Нейронная сеть предназначена для выполнения регрессии. В задаче классификации предсказывается значение  $\Pr(f(x) = y)$ . У нейрона есть функция активации  $g(x)$ , например  $\frac{1}{1 + e^{-x}}$ , которая нормализует  $x$  в простой диапазон вроде такого, как (0, 1), список весов  $w$  для любого признака  $f$  и функцию смещения с постоянным значением 1. Веса  $w$  образуют гиперплоскость в  $X$ . Нейрон вычисляет расстояние от заданной точки до этой

гиперплоскости как  $g(\sum w_i f_i)$ , что является расстоянием, потому что  $\sum w_i x_i = a$  для любого  $a$  есть набор гиперплоскостей. Итак, нейрон разбивает  $X$  на  $a \geq 0$  и  $a < 0$ .

Сеть представляет собой DAG нейронов, расположенных *слоями* так, что входными данными для первого слоя являются признаки, а для любых других — выходные данные предыдущего слоя. Последний слой является *выходным слоем*. Сеть обычно полностью подключена, но не обязательно.

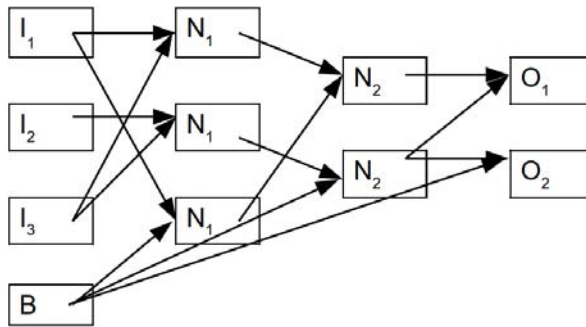


Рис. 26.17. Сеть с двумя скрытыми слоями ( $B$  — единица смещения)

Сеть с прямой связью с  $L$ -слоями — это функция, определенная рекурсивно как  $f(x) = f_{k,L-1}, f_{k,j+1} = \sum g(w_{kj} f_{kj}), f_{k,0} = x[k]$ . Обычно  $g$  бывают разными, в зависимости от поставленной задачи:

- ♦ выходной слой, регрессия — идентичность;
- ♦ выходной слой, классификация — логистическая функция. Он симметричен относительно  $(0, 0,5)$ ;
- ♦ скрытый слой — тангенс.

По сути, каждый последующий слой представляет собой преобразованное пространство признаков, которое ближе к  $Y$ . Невыходные слои находят расположение  $X$  (в двумерном случае это набор возможных пересекающихся линий, которые разбивают плоскость на области), конкретное значение которого определяет конкретную область для  $x$ , который затем используется выходным слоем для принятия решения. Как правило, но не всегда, каждый нейрон соединяется с каждым нейроном предыдущего слоя и смещения.

Теорема (см. [26.30, 26.38]): сеть с одной функцией идентичности  $g$  и одним скрытым слоем с ограниченной и непостоянной функцией активации может сколь угодно хорошо аппроксимировать с достаточным количеством скрытых нейронов в:

- ♦ норме  $L_\infty$  — любая непрерывная функция в компактной области, если  $g$  также непрерывна;
- ♦ норме  $L_p$  — для любого  $p$  любая функция, расстояние  $L_p$  которой до 0 функции  $< \infty$ .

Чтобы быть полезной для обучения, функция  $g$  также должна быть дифференцируема. Общие сигмоиды, такие как логистическая функция или тангенс, удовлетворяют этому условию. Но технически теорема неприменима к классификации, потому что даже при

$k = 2$  искомая функция не является непрерывной и обычно не является конечно близкой к 0. На практике это не вызывает проблем. Количество скрытых нейронов, необходимых для получения конкретной ошибки аппроксимации, неизвестно и зависит от задачи:

```
class NeuralNetwork
{
    struct Neuron
    {
        Vector<int> sources;
        Vector<double> weights;
        double output, error;
    };
    mutable NUMERIC_X inputs;
    bool isContinuousOutput;
    mutable Vector<Vector<Neuron> > layers;
    double actOutput(double x) const
    {return isContinuousOutput ? x : 1/(1 + exp(-x));}
    double actInner(double x) const{return tanh(x);}

public:
    NeuralNetwork(int D, bool theIsContinuousOutput = false,
        double theInitialLearningRate = 1): inputs(D), learnedCount(1),
        initialLearningRate(theInitialLearningRate),
        isContinuousOutput(theIsContinuousOutput) {}
    void addLayer(int nNeurons) {layers.append(Vector<Neuron>(nNeurons));}
    void addConnection(int layer, int from, int to, double weight)
    {
        Vector<Neuron>& last = layers[layer];
        last[from].sources.append(to);
        last[from].weights.append(weight);
    }
};
```

Входные данные распространяются на следующие слои, а выходные данные последнего слоя имеют результат:

```
void propagateInputs(NUMERIC_X const& x) const
{
    inputs = x;
    for(int i = 0; i < layers.getSize(); ++i)
        for(int j = 0; j < layers[i].getSize(); ++j)
        {
            Neuron& n = layers[i][j];
            double sum = n.error = 0;
            for(int k = 0; k < n.sources.getSize(); ++k)
                sum += n.weights[k] * getInput(i, n.sources[k]);
            n.output = i == layers.getSize() - 1 ?
                actOutput(sum) : actInner(sum);
        }
}
```

```
double getInput(int layer, int source) const
{
    return source == -1 ? 1 : layer == 0 ?
        inputs[source] : layers[layer - 1][source].output;
}
Vector<double> evaluate(NUMERIC_X const& x) const
{
    propagateInputs(x);
    Vector<double> result;
    for(int i = 0; i < layers.lastItem().getSize(); ++i)
        result.append(layers.lastItem()[i].output);
    return result;
}
```

В процессе обучения веса корректируются, чтобы минимизировать ошибку  $L_2$  выходных нейронов от оценки каждого обучающего примера. Процесс не выпуклый, поэтому используется локальная оптимизация. Самый простой, масштабируемый и эффективный метод обучения — это SGD, а производные рассчитываются с использованием *обратного распространения*. Для любого нейрона на ошибку влияют только ее веса, поэтому можно использовать градиентный спуск. Для любого выходного нейрона ошибка:

$$\varepsilon = \frac{1}{2} (g(w_f) - y)^2.$$

Относительно  $w$ ,  $\varepsilon' = tf$  для  $t = og'(w, f)$ , и ошибка вывода  $o = g(wf) - y$ . Для нейрона  $j$  в слое перед выходным слоем,  $\varepsilon_j$  = сумме ошибок нейронов, которые используют его выходные данные в качестве признака  $= \frac{1}{2} \sum (g(w_k f_k) - y_k)^2$ . Что касается  $w_j$ ,

$\varepsilon_j' = \sum (g(w_k f_k) - y_k) g'(w_k f_k) w_k[j] g'(w_j f_j) f_j = t_j f_j$  для  $o_j = \sum t_k w_k[j]$ . Согласно математической индукции формула работает для любого невыходного нейрона, что приводит к обновлению градиентного спуска  $w_{i+1} = w_i - s_i t f$ , где  $s_i$  — некоторый размер шага, удовлетворяющий условиям Роббинса — Манро (см. главу 24. Численная оптимизация). SGD сходится медленно, поэтому явной регуляризации обычно не требует. Используйте  $[10^5/n]$  или  $[10^6/n]$  проходов по данным, чтобы получить разумную точность (примерно от  $10^{-3}$  до  $10^{-2}$  асимптотически).

Для  $g(x) = \frac{1}{1 + e^{-x}}$ ,  $g'(x) = g(x)(1 - g(x))$ , для тангенса  $g'(x) = 1 - g(x)^2$ . Эти соотношения позволяют вычислить производные непосредственно из выходов нейронов, что дает экономию пространства:

```
long long learnedCount;
double initialLearningRate;
double learningRate()
{
    return initialLearningRate/pow(learnedCount++, 0.501);
}
double actOutputDeriv(double fx) const
{
    return isContinuousOutput ? 1 : fx * (1 - fx);
}
double actInnerDeriv(double fx) const {return 1 - fx * fx;}
void learn(NUMERIC_X const& x, Vector<double> const& results)
{
    assert(results.getSize() == layers.lastItem().getSize());
```

```

propagateInputs(x);
Vector<Neuron>& last = layers.lastItem();
for(int j = 0; j < last.getSize(); ++j) last[j].error =
    last[j].output - results[j];
double r = learningRate();
for(int i = layers.getSize() - 1; i >= 0; --i)
    for(int j = 0; j < layers[i].getSize(); ++j)
    {
        Neuron& n = layers[i][j];
        double temp = n.error * (i == layers.getSize() - 1 ?
            actOutputDeriv(n.output) : actInnerDeriv(n.output));
        for(int k = 0; k < n.sources.getSize(); ++k)
        { // обновление весов и ошибок предыдущего слоя
            int source = n.sources[k];
            if(i > 0 && source != -1)
                layers[i - 1][source].error += n.weights[k] * temp;
            n.weights[k] -= r * temp * getInput(i, source);
        }
    }
}

```

Для  $k = 2$  известны только границы VC-размерности  $d$  для сетей с конкретным  $g$ . Пусть  $W$  = количество весов в сети с сигмовидной функцией активации скрытого слоя и пороговой функцией активации на выходе. Тогда (см. [26.51, 26.2])  $O(W^2) < d < O(W^4)$ . Но здесь не учитываются вес и размер данных и не делается предположений об активации. Предполагая, что для представления каждого веса требуется  $O(1)$  битов, сети требуется  $O(W)$  битов, что приводит к лучшей верхней границе ORM. Вероятно, существуют лучшие, в настоящее время неизвестные, границы, которые учитывают размеры данных и весов. Тем не менее мы видим, что каждый вес имеет значение. Обучение на одном примере занимает время  $O(W)$ .

Применение SGD требует осторожности (см. [26.44]). Может возникнуть ситуация *насыщения*, т. е. когда для любого  $x$  суммы на одном или нескольких нейронах слишком малы/велики, а  $g'(x) \approx 0$ . Тем не менее, хотя для некоторых примеров суммы могут быть огромными и приводить к малым обновлениям, у других примеров может получаться достаточно большое  $g'(x)$ . Типичный рецепт решения задачи классификации для классической неглубокой архитектуры и параметров:

- ◆ для  $k > 2$  использовать метод OVO. В бинарных задачах выходной нейрон имеет значения 0/1. Нейронная сеть лучше различает два класса, чем множество. Подсчитайте выходные данные выходного слоя как доли для голосования OVO, не округляя до 0 или 1 (в качестве альтернативы в методе OVA для любого класса используйте выходной нейрон 0/1 с логистической активацией и выберите класс, соответствующий нейрону с наивысшим значением. В отличие от SVM, OVO менее эффективен в  $O(k)$  раз, но границы решений оказываются проще, и достаточно нескольких скрытых нейронов);
- ◆ используйте один скрытый слой с 5 нейронами. Эксперименты показывают, что это правило работает хорошо (см. [26.23]) и позволяет работать в режиме реального времени. Для 5 нейронов нелинейность больше, чем при логистической регрессии, и не слишком велика для возникновения переобучения. Другие числа вида  $2^i + 1$ , та-

кие как 3 или 9, или использование перекрестной проверки, дают аналогичные результаты;

- ◆ масштабируйте входные данные, чтобы получить среднее значение, равное 0, и стандартное отклонение, равное 1. Это помогает сохранять небольшие суммы;
- ◆ инициализируйте выходные нейроны и веса смещения равными 0, а остальные — равномерно распределенными значениями в диапазоне  $(-a, a)$ , где  $a = \sqrt{3/D}$ . Тогда последние веса будут иметь среднее значение 0 и стандартное отклонение  $1/\sqrt{D}$ , среднее значение суммы 0 и стандартное отклонение 1. Это делает насыщение маловероятным и дает разным скрытым нейронам разные веса, так что они не будут учиться одному и тому же (а если начать с 0, то это возможно). Каждая скрытая единица по существу делает случайную проекцию при инициализации;
- ◆ для скрытых слоев используйте симметричную около 0 функцию  $g$ . Большинство источников рекомендуют  $g = \tanh$ . Здесь используется та же логика, что и для масштабирования входных данных до 0, — например, с логистическим  $g(0) = 0,5$ , поэтому суммы на нейронах следующего слоя будут равны  $0,5 \times$  количество входов, которые могут быть немедленно насыщены;
- ◆ хорошо работает начальная скорость обучения  $= 1$ ;
- ◆ используйте 5 сетей, различающихся только начальными весами, и усредняйте их выходы. Это делается для устранения дисперсии и, как правило, хорошо работает экспериментально (см. [26.23]).

```
class BinaryNN
```

```
{
    Vector<NeuralNetwork> nns;
    void setupStructure(int D, int nHidden)
    {
        double a = sqrt(3.0/D);
        for(int l = 0; l < nns.getSize(); ++l)
        {
            NeuralNetwork& nn = nns[l];
            nn.addLayer(nHidden);
            for(int j = 0; j < nHidden; ++j)
                for(int k = -1; k < D; ++k) nn.addConnection(0, j, k,
                    k == -1 ? 0 : GlobalRNG().uniform(-a, a));
            nn.addLayer(1);
            for(int k = -1; k < nHidden; ++k) nn.addConnection(1, 0, k, 0);
        }
    }

public:
    BinaryNN(int D, int nHidden = 5, int nNns = 5):
        nns(nNns, NeuralNetwork(D)){setupStructure(D, nHidden);}
    template<typename DATA> BinaryNN(DATA const& data, int nHidden = 5, int
        nGoal = 100000, int nNns = 5): nns(nNns, NeuralNetwork(getD(data)))
    {
        int D = getD(data), nRepeats = ceiling(nGoal, data.getSize());
        setupStructure(D, nHidden);
        for(int j = 0; j < nRepeats; ++j)
            for(int i = 0; i < data.getSize(); ++i)
                learn(data.getX(i), data.getY(i));
    }
}
```



```

void learn(NUMERIC_X const& x, int label)
{
    for(int l = 0; l < nns.getSize(); ++l)
        nns[l].learn(x, Vector<double>(1, label));
}
double evaluate(NUMERIC_X const& x) const
{
    double result = 0;
    for(int l = 0; l < nns.getSize(); ++l)
        result += nns[l].evaluate(x)[0];
    return result/nns.getSize();
}
int predict(NUMERIC_X const& x) const {return evaluate(x) > 0.5;}
};
class MulticlassNN
{
    MulticlassLearner<NoParamsLearner<BinaryNN, int>, EMPTY> model;
public:
    template<typename DATA> MulticlassNN(DATA const& data): model(data) {}
    int predict(NUMERIC_X const& x) const {return model.classifyByProbs(x);}
};
typedef ScaledLearner<NoParamsLearner<MulticlassNN, int>, int, EMPTY, ScalerMQ> SNN;

```

У нейронных сетей (рис. 26.18) есть много проблем, которые не решаются годами исследований:

- ◆ поиск (SGD или другие методы) легко попадает в ловушку в локальных минимумах произвольного качества. Усреднение нескольких случайно обученных сетей ведет к проблеме катастрофы центрального предела (см. главу 24. Численная оптимизация), но лучших альтернатив все равно нет. Вы можете попробовать методы глобальной оптимизации, но нужна явная регуляризация, чтобы они не переобучались, — можно использовать штраф  $L_2$ , который для нейронных сетей называется *затуханием веса*. Тем не менее это занимает слишком много времени, а результаты вряд ли будут воспроизводимыми. Таким образом, ядро SVM используется вместо нейронных сетей в качестве метода «черного ящика», когда нужно реализовать нелинейность. Но в некоторых приложениях, таких как игры, глобально обученная нейронная сеть может дать хорошие результаты (см. [26.41]). Но существуют и другие методы, которые могли бы дать лучшие результаты;
- ◆ обучение работает медленно даже при использовании SGD и занимает время  $O(k^2 D \times \text{количество примеров} \times \text{количество скрытых нейронов})$ . По сути, для выбора простора нет. В некоторых реализациях затухание подвергается перекрестным проверкам (см. [26.23, 26.14, 26.13]), но более быстрый SGD низкой точности обеспечивает автоматическую регуляризацию.

Сети с одним скрытым слоем типа «черный ящик» обычно работают хуже, чем самые эффективные алгоритмы  $A$  вроде SVM или случайного леса (обсуждается позже), но они полезны в некоторых особых случаях, таких как онлайн-обучение. *Самые эффективные сети используют знания предметной области*. Например, знаменитая сверточная сеть для распознавания цифр (см. [26.28]) группирует соседние пиксели (в более общем смысле — сильно коррелированные признаки), и на ее разработку ушли годы.

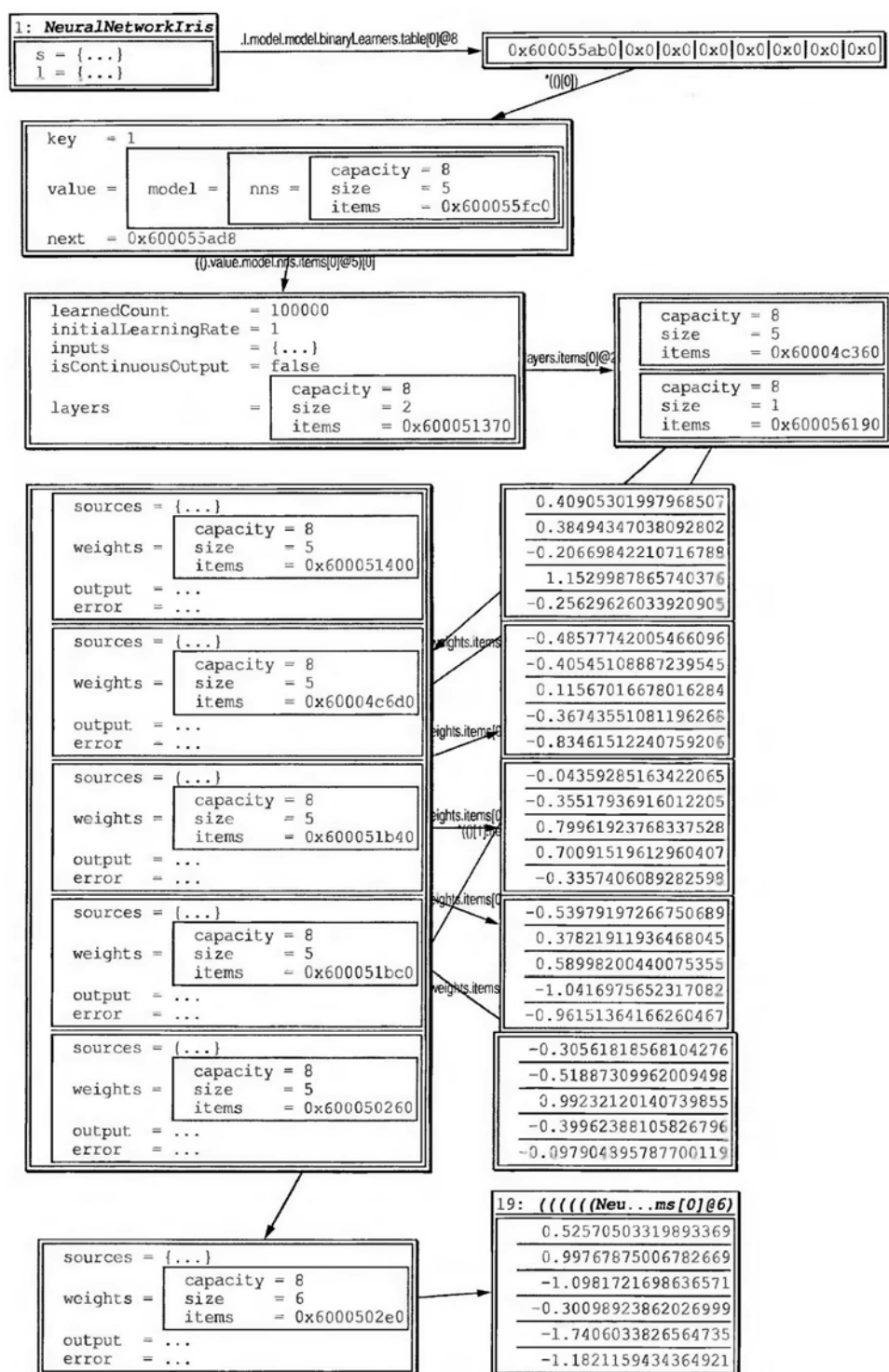


Рис. 26.18. Структура памяти нейронной сети на данных о цветках ириса

Также в некоторых предметных областях можно проводить обучение на слегка повернутых изображениях, что существенно повышает производительность (см. [26.44]).

*Глубокое обучение* — это новое веяние в области нейронных сетей, для которого характерны следующие особенности (см. [26.44]):

- ◆ более одного скрытого слоя (глубокая архитектура). Одному скрытому слою для изучения некоторых признаков может потребоваться экспоненциально много нейронов. Наличие большего количества слоев позволяет избежать этого;
- ◆ неконтролируемая предварительная подготовка, которая делает возможной обучение глубокой архитектуры. У стандартной сети с более чем одним скрытым слоем существует *затухающий градиент* (ошибки, отправленные на нижние слои,  $\approx 0$ ) и слишком много проблем с локальными минимумами;
- ◆ использование *функции активации ReLU*, которая представляет собой обрезанную линию (технически неограниченную, но со сходимостью — см. [26.5]). В настоящее время считается, что она лучше классической тангенциальной функции.

При работе с данными без учителя поможет *автоматическое кодирование*:

- ◆ к любому скрытому слою прикрепляется дополнительный слой декодера, который пытается реконструировать вход. Можно использовать стандартный SGD с обратным распространением (рис. 26.19);

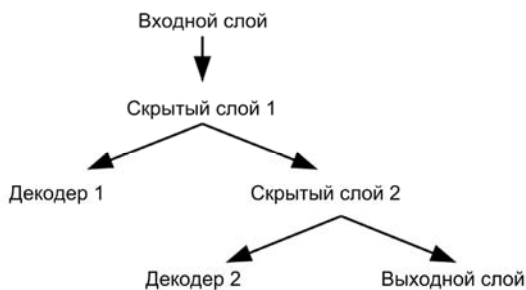


Рис. 26.19. Скрытые слои и декодеры

- ◆ начиная с первого слоя, обучайте каждый слой, используя декодер как последний слой, но без изменения уже обученных слоев. Необходимо перекрестно проверять начальную скорость обучения блоков декодера, потому что производные функций активации оказываются не ограничены;
- ◆ используйте для обратного распространения обычный порядок;
- ◆ в простой архитектуре используется равное количество скрытых нейронов (с перекрестной проверкой) для любого слоя  $i$ , возможно, 5 слоев. Это делается для того, чтобы избежать переобучения, но есть много, вероятно, лучших вариантов (см. [26.5]);
- ◆ из-за использования неконтролируемых данных невозможно разложить задачу на несколько бинарных, поэтому используйте  $k$  выходных блоков с OVA. Этот метод требует изучения более сложных границ, но работает быстрее, а глубокая архитектура предположительно компенсирует проблемы, т. к. находит лучшие признаки.

Глубокое обучение в настоящий момент активно исследуется и еще не произвело конкурентоспособного алгоритма «черного ящика» (это спорный вопрос, который еще предстоит экспериментально подтвердить). Но многие результаты, относящиеся к конкретным приложениям, весьма актуальны, поскольку основные изобретатели получили награду ACM Turing. Пока открыты вопросы о том, как настроить глубокую нейронную сеть так, чтобы она была столь же эффективна, как, например, случайный лес, и почему у нее нет проблем с переобучением.

## 26.14. Ансамбли рандомизации

Объединение нескольких функций  $f$  может сработать лучше, чем одна функция. Самый простой способ сделать это — использовать большинство голосов, т. е. брать за ответ наиболее популярный класс. *Теорема присяжных*: если  $T$  функций  $f_i$  независимы и для каждого  $\Pr(\text{выбранный класс правильный} > 0,5)$  при  $T \rightarrow \infty$   $\Pr(\text{большинство ошибочно}) \rightarrow 0$ . При  $k > 2$  достаточно правильности  $> 1/k$ , предполагая, что неправильный ответ более вероятен, чем правильный. Доказательство: вероятность правильного ответа будет увеличиваться намного быстрее, чем вероятность другого ответа, которая может увеличиться только случайно.

Но множество  $f$  не может быть полностью независимым, потому что все они обучаются на одних и тех же данных. Кроме того, дисперсия неинформативных  $f_i$  будет вносить существенный шум в близкие решения информативных  $f$ , поэтому следует убедиться, что  $f$  является информативной.

*Дерево решений с мешками* — это применение механизма бэггинга (см. главу 25. *Основы машинного обучения*) для получения преимуществ рандомизации. Деревья не обрезаются, чтобы уменьшить смещение, но ограничение по глубине по-прежнему действует. *Случайный лес* позволяет улучшить этот метод, уменьшая предвзятость за счет более высокой дисперсии из-за уменьшения корреляции между деревьями. Каждое дерево становится более случайным вследствие запрета использования случайного подмножества функций для разделения, т. е. для каждого узла выбирается свое подмножество. Обычно  $f_i$  с малой корреляцией друг с другом имеют мало общего.

Количество разрешенных признаков  $a$  должен быть достаточно маленьким, чтобы декоррелировать деревья, но достаточно большим, чтобы не ослаблять их. В исходной статье выбирается значение  $a = \sqrt{D}$ , которое кажется надежным. Альтернативный вариант  $a = \lfloor \lg(D) \rfloor$  дает меньшую корреляцию, а логическое  $a = D^{3/4}$  (8 из 16, 27 из 81 и т. д.) — большую силу. По моим тестам первый вариант хуже, а второй примерно такой же, так что нет причин отдавать предпочтение ни тому, ни другому. Как улучшить определенные параметры:

- ◆ разнообразие деревьев — меньше  $a$ ;
- ◆ сила каждого дерева — больше  $a$ ;
- ◆ производительность — меньше  $a$  и  $T$ .

Выбор  $T$  зависит от проблемы и сам по себе довольно труден:

- ◆ слишком большие значения неэффективны с точки зрения затрат времени и памяти;
- ◆ слишком маленьких недостаточно для стабилизации прогнозов и результатов в рандомизированном алгоритме с высокой дисперсией.

В источниках предлагается использовать  $T = 1000$ , и это значение довольно распространено. Разумно выполнять поиск по сетке от этого числа, используя приближенный коэффициент  $\sqrt{10}$  шагов. Судя по результатам, значение 300 кажется лучше (рис. 26.20):

- ◆ значение 1000 связано с вариантом использования. Согласно экспериментам использование большего количества чисел перестает иметь значение (см. [26.12]);
- ◆ 100 — наименьшее значение с достаточно хорошей производительностью;
- ◆ 100 — близко к плохой производительности, а 1000 — к ненужной неэффективности;
- ◆ относительные стандартные отклонения всех трех значений расположены близко, но способствуют выбору чуть большего варианта;
- ◆ для прогнозирования требуется  $O(T)$  базовых прогнозов учащегося;
- ◆ случайный лес полезен как «черный ящик», иногда с дополнениями, поэтому ни эффективность, ни дисперсия не должны являться проблемой при выборе по умолчанию;
- ◆ использование перекрестной проверки для удвоения чисел происходит медленно, дисперсия в производительности может привести к плохому выбору, и  $T$ , пока достаточно велико, не имеет большого значения.

Bag 1k	RF lg 1000	RF 075 1k	RF 05 30	RF 05 100	RF 05 300	RF 05 1k	RF 05 3k
0.941	0.957	0.971	0.953	0.962	0.964	0.967	0.968

Рис. 26.20. Некоторые результаты сравнения производительности значений  $a$  и  $T$

Случайный лес достаточно хорошо оценивает вероятности, используя пропорции счета. Бэггинг тоже подходит, но только с неустойчивым основанием  $A$ :

1. Выбрать  $T(a$  жестко задано).
2.  $T$  раз:
3. Создать повторную выборку размером  $n$  из  $S$ .
4. Получить дерево решений из повторной выборки, случайным образом выбирая признаки для любого узла, не обрезая дерево;
5. Чтобы предсказать  $x$ , классифицировать его, используя каждое дерево, и найти класс большинства или использовать пропорции счета, чтобы получить вероятности.

```
class RandomForest
{
    Vector<DecisionTree> forest;
    int nClasses;
public:
    template<typename DATA> RandomForest(DATA const& data, int nTrees = 300):
        nClasses(findNClasses(data))
    {
        assert(data.getSize() > 1);
        for(int i = 0; i < nTrees; ++i)
        {
            PermutedData<DATA> resample(data);
```

```

    for(int j = 0; j < data.getSize(); ++j)
        resample.addIndex(GlobalRNG().mod(data.getSize()));
    forest.append(DecisionTree(resample, 0, true));
}

template <typename ENSEMBLE> static int classifyWork(NUMERIC_X const& x,
    ENSEMBLE const& e, int nClasses)
{
    Vector<int> counts(nClasses, 0);
    for(int i = 0; i < e.getSize(); ++i) ++counts[e[i].predict(x)];
    return argMax(counts.getArray(), counts.getSize());
}

int predict(NUMERIC_X const& x) const
{
    return classifyWork(x, forest, nClasses);
}

Vector<double> classifyProbs(NUMERIC_X const& x) const
{
    Vector<double> counts(nClasses, 0);
    for(int i = 0; i < forest.getSize(); ++i)
        ++counts[forest[i].predict(x)];
    normalizeProbs(counts);
    return counts;
}
};

```

Дополнительным признаком выступает оценка риска *вне мешка*, т. е. для любого  $z_i \in S$ , вычисление  $y$  на основе подлеса, состоящего из деревьев, не обученных на  $z_i$ . Оценка работает так же точно, как оценка независимого набора тестов (см. [26.12]). Поскольку примерно  $1/e$  деревьев не включает конкретный пример,  $a \approx 200$  деревьев дает хорошую производительность, и метод лучше применять, имея  $> 600$  деревьев.

Теоретическая проблема заключается в том, что случайный лес, основанный на жадно обученном дереве решений, не является согласованным из-за маловероятных особых случаев (см. [26.6]). Но на практике это, кажется, не вызывает проблем.

## 26.15. Обучение в реальном времени

Ближайший сосед, линейный SVM и нейронная сеть обучаются в реальном времени. Но ближайший сосед занимает слишком много памяти, а линейный SVM и нейронная сеть могут не иметь максимальной производительности. Кроме того, с точки зрения согласованности все, кроме ближайшего соседа, сохраняют модель постоянной сложности в отношении количества изученных примеров, тогда как лучшие автономные алгоритмы адаптируют свои модели.

Алгоритм реального времени  $A$ , которому требуется  $k = 2$ , также может работать с  $k > 2$ . Увидев новый пример, при необходимости отрегулируйте  $k$ , и будут созданы новые бинарные учащиеся. Если некоторые классы неизвестны в течение длительного времени, у новых учащихся будет меньше данных для обучения. Например, если бы 0 и 1 появились первыми, а 2 позже, сравнение 0 против 2 и 1 против 2 пропустило бы примеры с  $y = 0$  и 1. Эта проблема не возникает, если данные не отсортированы по классам.

```

template<typename LEARNER, typename PARAMS = EMPTY, typename X = NUMERIC_X>
class OnlineMulticlassLearner
{
    mutable Treap<int, LEARNER> binaryLearners;
    int nClasses;
    PARAMS p;
    int makeKey(short label1, short label2) const
    {return label1 * numeric_limits<short>::max() + label2;}

public:
    OnlineMulticlassLearner(PARAMS const& theP = PARAMS(),
    int initialNClasses = 0): nClasses(initialNClasses), p(theP) {}
    void learn(X const& x, int label)
    {
        nClasses = max(nClasses, label + 1);
        for(int j = 0; j < nClasses; ++j) if(j != label)
        {
            int key = j < label ? makeKey(j, label) : makeKey(label, j);
            LEARNER* s = binaryLearners.find(key);
            if(!s)
            {
                binaryLearners.insert(key, LEARNER(p));
                s = binaryLearners.find(key);
            }
            s->learn(x, j < label);
        }
    }
    int predict(X const& x) const
    {
        assert(nClasses > 0);
        Vector<int> votes(nClasses, 0);
        for(int j = 0; j < nClasses; ++j)
            for(int k = j + 1; k < nClasses; ++k)
            {
                LEARNER* s = binaryLearners.find(makeKey(j, k));
                if(s) ++votes[s->classify(x) ? k : j];
            }
        return argMax(votes.getArray(), votes.getSize());
    }
};

```

Это позволяет реализовать линейный SVM в реальном времени. Не зная  $n$ , можно использовать известное количество обработанных примеров. В случае масштабирования студентизация, как правило, работает лучше, чем диапазон, скорее всего, из-за лучшей оценки дисперсии. Для тестирования на данных в автономном режиме выберите примеров  $10^6$  из  $S$  и изучите их в реальном времени. Случайная выборка для большинства наборов данных дает гораздо лучшую производительность, чем использование исходного порядка, поскольку она улучшает баланс. Обеспечение баланса является критической проблемой в обучении в реальном времени, поскольку  $A$  оценивает границы разделов, для которых обучаемая единица представляет собой слои примеров с разными метками:

```

class SRaceLSVM
{
    ScalerMQ s;
    typedef pair<int, double> P;
    RaceLearner<OnlineMulticlassLearner<BinaryLSVM, P>, P> model;
    static Vector<P> makeParams(int D)
    {
        Vector<P> result;
        int lLow = -15, lHigh = 5;
        for(int j = lHigh; j > lLow; j -= 2)
        {
            double l = pow(2, j);
            result.append(P(D, l));
        }
        return result;
    }
public:
    template<typename DATA> SRaceLSVM(DATA const& data):
        model(makeParams(getD(data))), s(getD(data))
    {
        for(int j = 0; j < 1000000; ++j)
        {
            int i = GlobalRNG().mod(data.getSize());
            learn(data.getX(i), data.getY(i));
        }
    }
    SRaceLSVM(int D): model(makeParams(D)), s(D){}
    void learn(NUMERIC_X const& x, int label)
    {
        s.addSample(x);
        model.learn(s.scale(x), label);
    }
    int predict(NUMERIC_X const& x) const{return model.predict(s.scale(x));}
};

```

Производительность сильно зависит от наличия достаточного количества примеров ( $\geq 10\,000$ ) в сбалансированном порядке. Это позволяет SGD быстро находить отступы, не беря в расчет отдаленные иллюзорные отступы. Аналогично масштабируется и нейронная сеть (рис. 26.21):

```

class SOnlineNN
{
    ScalerMQ s;
    OnlineMulticlassLearner<BinaryNN, int> model;
public:
    template<typename DATA> SOnlineNN(DATA const& data): model(getD(data)),
        s(getD(data))
    {
        for(int j = 0; j < 1000000; ++j)
        {
            int i = GlobalRNG().mod(data.getSize());

```



```

        learn(data.getX(i), data.getY(i));
    }
}
SOnlineNN(int D): model(D), s(D) {}
void learn(NUMERIC_X const& x, int label)
{
    s.addSample(x);
    model.learn(s.scale(x), label);
}
int predict(NUMERIC_X const& x) const {return model.predict(s.scale(x));}
};

```

SracedLSVM 10r6	MqracedLSVM 10r6	SonlineNNr 10r6
0.908	0.923	0.936

Рис. 26.21. Сравнение производительности

## 26.16. Бустинг

*Бустинг* — это попытка создать набор  $f_i$  такой, что каждая следующая функция пытается улучшить производительность на примерах, неправильно классифицированных ансамблем до нее, придавая им больший вес. Например, в задаче регрессии метод бустинга, по сути, похож на *удвоение по Тьюки*, т. е. сначала выполняется регрессия для  $y$ , а затем снова для ошибок. Предполагая, что  $k = 2$  с  $y \in \{-1, 1\}$ , нужны базовые классификаторы  $h_i$  и веса  $a_i$  такие, что комбинированный классификатор  $\text{sign}(F = \sum a_i h_i)$  имеет минимальный риск  $= \sum L(F_j, y_j)$ , где  $L$  — некоторая функция потерь, а  $F_j$  — сумма примера  $j$ .

Оптимизация является NP-сложной, когда  $L$  представляет собой двоичную потерю, поэтому вместо этого необходимо использовать *суррогатную функцию потерь*  $L$ , которая:

- ♦  $\geq$  бинарных потерь — бустинг уменьшает последнюю с использованием верхней границы;
- ♦ выпукла — тогда  $\sum L_j$  выпукла и легко оптимизируется;
- ♦ монотонно убывает —  $F$ , по сути, определяет достоверность правильного решения, поэтому примеры с большим  $F$  безопаснее классифицировать.

Нахождение  $F$ , которое сводит к минимуму  $L$ , есть числовая минимизация в пространстве функций, которая может быть выполнена с помощью градиентного спуска. Начиная с отсутствия информации  $F = 0$ , каждый следующий  $h$  является шагом к минимуму:

1.  $F_j = 0$  для любого примера  $j$ .
2.  $T$  раз:
  3. Найти  $h$  ближайшее к  $d = -dL/dF$ .
  4. Выбрать  $a_j$  аналитически или численно.
  5.  $F_j += a_j h_j$  для любого примера  $j$ .

Чтобы найти  $h$ , нужно минимизировать одно из:

- ♦ *AnyBoost* — скалярное произведение  $h \times d$  удобно для классификации и требует взвешивания примеров. Для  $L$  такого, что  $d \geq 0$ , что верно для всех популярных  $L$ , это эквивалентно взвешиванию каждого примера  $j$  на  $d_j$ . Чтобы избежать обучения базы  $A$  на взвешенных примерах, выполните передискретизацию с использованием распределения, заданного весами. Обычно размер повторной выборки =  $n$ ;
- ♦ *повышение градиента* — норма  $L_2 \|h - d\|$  удобна для регрессии с потерями  $L_2$  и не требует взвешивания. Необходимо обучить алгоритм регрессии, чтобы предсказывать действительное  $y$  для минимизирования  $L$ . Примерно это же делает удвоение Тьюки.

*AdaBoost* получается с использованием экспоненциальной потери  $L(F, y) = e^{-yF}$ . Если  $r = \sum (h \neq y)d$  — это взвешенная ошибка  $h$ , оптимальное  $a = \frac{1}{2} \ln\left(\frac{1-r}{r}\right)$  (см. [26.59]).

Теоретически нужна слабая обучающая база  $A$  (см. [26.59]), которая имеет ошибку  $\varepsilon < 0,5$  с вероятностью  $\geq 1 - p$  для любого  $p > 0$  и достаточно большого (в зависимости от  $p$ )  $n$ . Чтобы реализация работала, нужно всего лишь  $\varepsilon < 0,5$  для любой повторной выборки. Теорема (см. [26.43]): предположим, что:

- ♦ база  $A$  слабая, и ее  $G$  имеет размерность VC, равную  $d$ ;
- ♦  $a_i$  нормализуется так, что  $\sum a_j = 1$ ;
- ♦  $L_1$  отступ  $m = \min_i y_i (ah(x_i)) > 0$ ;
- ♦ Каждое  $h_j$  достигает  $\varepsilon_j < 1/2$ .

Затем после  $T$  раундов и любого фиксированного достигнутого  $m$  с вероятностью  $\geq 1 - p$ :

$$R_f \leq 2^T \prod \sqrt{\frac{1-m}{j} (1-\varepsilon_j)^{1+m}} + \sqrt{\frac{8d \ln(en/d)}{nm}} + \sqrt{\frac{-\ln(p)}{2n}}.$$

В частности, для фиксированного  $p$  и достаточного количества раундов ошибка обобщения составляет  $O(1/\sqrt{n})$  и  $\rightarrow 0$ . Это невозможно для  $R_{об} > 0$  — слабое допущение  $A$  слишком сильное, поэтому более интуитивно понятно предположить, что распределение по  $Z$  урезано так, чтобы любые неопределенные примеры меток имели нулевую поддержку. Затем, наблюдая шумный пример из реального распределения:

- ♦ в тестовых данных, получаем единственную ошибку;
- ♦ в обучающих данных, получаем «дыру для ошибок» на основе полей вокруг примера.

Вторая проблема намного хуже — эксперименты с AdaBoost показывают, что ошибка обобщения  $\approx 0,5$  для обучения на очень зашумленных данных (см. [26.59]). В более общем смысле (см. [26.39]) любой алгоритм бустинга с любой выпуклой потерей при условии постоянной доли зашумленных примеров в обучающих данных можно заставить работать почти случайным образом. Но большинство практических наборов данных не являются таким худшим случаем, и AdaBoost работает хорошо (см. [26.59]). Тем не менее теоретически не существует золотой середины между отличной производительностью при  $R_{об} = 0$  и плохой при  $R_{об} > 0$ .

Важным выводом является то, что база  $A$  должна балансировать между низкой сложностью и силой, чтобы получить хорошие отступы. Сильно усеченное дерево решений является естественной базой  $A$ . Для базы  $A$  ошибки обобщения и обучения должны быть близки, а производительность — максимально лучше случайной. Первое более важно — например, неотсеченное дерево решений, скорее всего, будет иметь ошибку обучения, равную 0, даже при повторной выборке. Может показаться, что ветви решений, используемые в распознавании лиц, являются лучшими  $A$ . Но экспериментально (см. [26.14]) и теоретически (см. [26.59]) они работают плохо. Кроме того, для чего-то вроде задачи `xor` ветви не являются слабыми учениками. Наконец, ветви дают больше поля вокруг примеров с зашумленными метками, что приводит к большим дырам. В идеале база  $A$  должна быть достаточно прочной, чтобы максимально изолировать отверстия. Распознавание лиц использует очень специально спроектированные ветви, с которыми бустинг работал хорошо из-за его свойства уменьшать смещение. Для вектора  $x$  не существует причин не использовать деревья решений в качестве базы  $A$ .

*SAMME* расширяет AdaBoost на случай  $k > 2$ , требуется базовая точность обучаемого  $> 1/k$  (см. [26.58]). Что касается случайного леса, используйте  $T = 300$ . Чем меньше  $T$ , тем меньше вероятность переобучения, но это не является проблемой для случайного леса. Реализация следует градиентному спуску с экспоненциальными потерями, но делает упрощения:

- ◆ отрицательные градиенты  $d$  обрабатываются напрямую, а веса и суммы данных  $F$  — нет;
- ◆ веса равны  $\log\left(\frac{1-r}{r}\right) + \log(k-1)$ , как и для AdaBoost при  $k = 2$  (постоянные множители не имеют значения), и формула обновления веса использует  $y \in \{0, 1\}$  для упрощения алгебры.

Если какая-либо точность  $< 1/k$ , для надежности неэффективные  $h$  удаляются. Если  $error = 0$ ,  $\log(1)/0 = \text{NaN}$ , повторная выборка завершится ошибкой для любого последующего раунда. Надежное решение состоит в том, чтобы заменить ансамбль на  $h$  (удаление его может удалить все эффективные  $h$  для очень простых задач).

Некоторые модификации имеют смысл и улучшают расширяемость, но не производительность в целом:

- ◆ используйте  $L$ , которые придают меньшее значение неправильно классифицированным примерам — в частности, из-за вероятностной интерпретации многие варианты используют *биномиальные отклонения* (также называемые *логистическими*) в качестве потери, равные  $\lg(1 + e^{-\text{margin}})$ , которые масштабируются до предельных двоичных потерь (хотя постоянные коэффициенты не имеют значения). Поскольку формула почти линейна слева, вес наихудшего случая  $\approx 1$  для любого ошибочно классифицированного примера. Но экспериментально это не помогает бороться ни с шумом, ни с переобучением (см. [26.59]). Получение аналитических весов работает только с экспоненциальными и логистическими потерями (см. [26.59]) — для многих других единственным выбором является численная оптимизация, которую трудно анализировать и которая требует надежной реализации. Hinge-потери не работают, потому что правильно классифицированные примеры за пределами отступа = 1 имеют вес 0, и слишком много примеров игнорируется в последующих раундах (рис. 26.22);

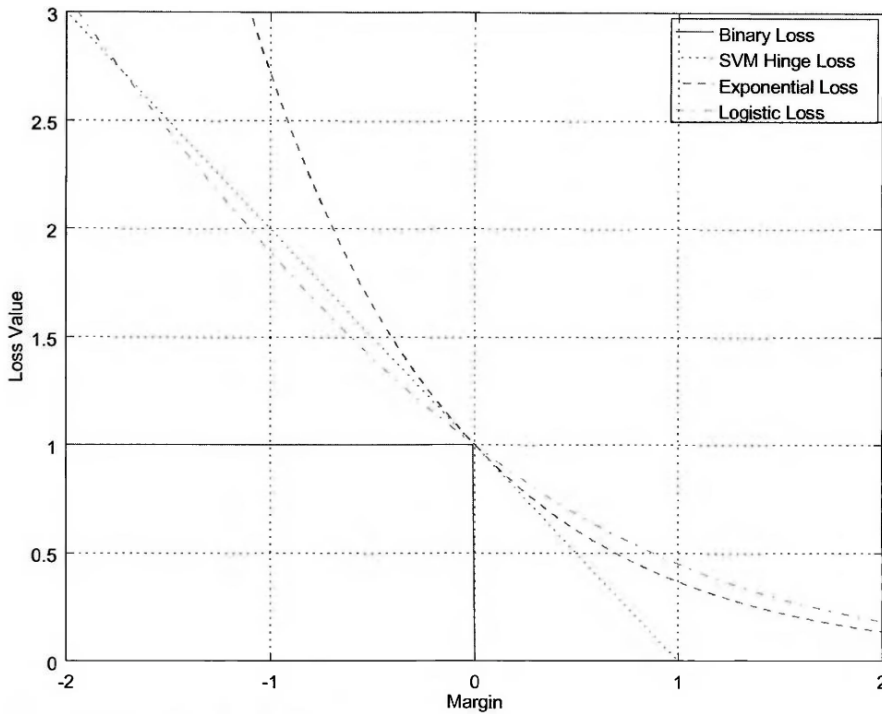


Рис. 26.22. Общие функции потерь

- ◆ для  $k > 2$  подходит отступ  $F(\text{метка}) - F(\text{наилучшая ложная метка})$ , потому что именно такие границы использует классификация. Но во многих вариантах используется  $F(\text{метка}) - \sum_{i \neq \text{метка}} F(i)$ , что слишком пессимистично (большинство вместо множества), но ограничивает отступ, который становится также легче анализировать;
- ◆ аналитические веса могут переобучаться. Вместо этого предлагается использовать необязательные постоянные веса (см. [26.28]), а веса Роббинса — Манро кажутся еще лучше, гарантируя сходимость градиентного спуска. Но такой неосведомленный выбор может привести к недооценке учащихся с плохой успеваемостью.

Получаем алгоритм *RMBoost* (обсуждается позже) с  $T = 1000$  по умолчанию, на основе экспериментальных сравнений. Значение 100 работает немного хуже. В целом для бустинга следует использовать SAMME, но *RMBoost* легко расширяется до классификации с учетом затрат. Это полезно, потому что в моих экспериментах пропорции счета не дают точных вероятностей, вероятно, из-за более сильной зависимости между основанием  $f$ . Код представлен в разделе изучения стоимости в этой главе.

Интуитивно понятно, что в то время, как случайный лес уменьшает дисперсию, бустинг уменьшает и то и другое — в основном смещение в первых нескольких итерациях и дисперсию в последних (см. [26.59]).

## 26.17. Масштабное обучение

Многие алгоритмы от природы распараллеливаемы, например:

- ◆ перекрестная проверка — этапы и тестируемые параметры независимы;
- ◆ ближайший сосед — сохраняется несколько индексов и объединяются ответы;
- ◆ случайный лес — деревья независимы;
- ◆ мультиклассовое преобразование — бинарные обучающиеся независимы,

а многие нет:

- ◆ учащиеся после бустинга зависят друг от друга;
- ◆ невозможно обучить бинарный SVM параллельно.

## 26.18. Экономичное обучение

Распространенной проблемой является учет стоимости ошибок. Например, для предсказания: потеря от ношения зонтика < потери отсутствия его во время шторма, но  $f$  может сказать, что *зонтик не нужен*, если его  $\Pr(\text{дождь}) = 49\%$ . Кроме того, для безопасного объекта ошибочный отказ во входе намного дешевле, чем доступ для кого-то, кого пускать нельзя.

Что касается затрат, правильные решения имеют стоимость 0 или какое-то отрицательное значение затрат/прибыли, а неправильные — стоимость  $C$  (предсказываемый класс, фактический класс) для некоторой матрицы затрат  $C$ , которая назначает ячейку матрицы затрат для любой ошибки. Если каждая ошибка одинаково затратна, все затраты = 1. Обычно  $C$  нормируется:

- ◆ диагональ = 0, т. е. правильный ответ ничего не стоит;
- ◆ все остальные записи  $\in (0, 1]$ . Это гарантирует отсутствие эффекта диапазона для алгоритмов, на которые он влияет. Умножение  $C$  на скаляр приводит к эквивалентной задаче.

```
void scaleCostMatrix(Matrix<double>& cost)
{
    double maxCost = 0;
    for(int r = 0; r < cost.getRows(); ++r)
        for(int c = 0; c < cost.getRows(); ++c)
            maxCost = max(maxCost, cost(r, c));
    cost *= 1/maxCost;
}
```

Теорема (см. [26.42]): прибавление константы для любого столбца  $C$  дает эквивалентную задачу. Это упрощает нормализацию для диагоналей с отрицательной стоимостью — прибавьте значение диагонали к каждому соответствующему столбцу.

Вы можете рассчитать стоимостной риск для модели, действующей матрицы затрат и ошибок. Обычно используют *стоимостной риск* с соответствующими доверительными границами:

```
double evalConfusionCost(Matrix<int> const& confusion,
    Matrix<double> const& cost)
```

```

{
    int k = confusion.rows, total = 0;
    assert(k == confusion.columns && k == cost.rows && k == cost.columns);
    double sum = 0;
    for(int r = 0; r < k; ++r)
        for(int c = 0; c < k; ++c)
        {
            total += confusion(r, c);
            sum += confusion(r, c) * cost(r, c);
        }
    return sum/total;
}

```

Способы минимизации риска затрат:

- ◆ игнорировать затраты — приемлемо для высокоточных  $A$  и не дает большой разницы в стоимости, потому что ошибки случаются редко и благодаря нормализации стоимостной риск  $\leq$  бинарный риск  $\times$  максимальная стоимость любой ошибки;
- ◆ использовать  $A$ , которые выводят вероятности. Это особенно удобно в случае со случайным лесом, где оценки вероятности достаточно точны. Идея других методов состоит в том, чтобы уменьшить ошибку оценки путем корректировки маржи с учетом затрат:

```

template<typename LEARNER = RandomForest> class CostLearner
{
    Matrix<double> cost;
    LEARNER model;
public:
    template<typename DATA> CostLearner(DATA const& data,
        Matrix<double>const& costMatrix): model(data), cost(costMatrix) {}
    int predict(NUMERIC_X const& x)const
        {return costClassify(model.classifyProbs(x), cost);}
};

```

- ◆ изменять  $A$  индивидуально для учета затрат. Нужно заменить бинарный риск ценовым риском во всех решениях, включая выбор параметров перекрестной проверкой и оптимизацию внутренних целей или жадное принятие решений. Иногда это выглядит неуклюже — например, для дерева решений не существует энтропии, взвешенной по стоимости;
- ◆ использовать универсальные методы, чтобы любой  $A$  мог учитывать расходы. Простым решением является повторная выборка — например выбор примера с весом, пропорциональным его ожидаемой стоимости. Это хорошо работает для  $k = 2$ . *Теорема народной стоимости* (см. [26.57]): для  $k = 2$  для любого  $f$  ожидаемый риск от классификации с затратами эквивалентен ожидаемому риску от классификации без затрат на выборках из распределения, пропорционального затратам. Так можно запускать  $A$  на выборках из последнего, полученного ресемплингом. Для  $k > 2$  это не работает, потому что затраты зависят от неизвестных предсказанных меток. Эвристический прием заключается в использовании средней стоимости (см. [26.42]):

```

template<typename LEARNER, typename PARAMS = EMPTY,
    typename X = NUMERIC_X> class AveCostLearner

```

```

{
    LEARNER model;
    template<typename DATA> static Vector<double> findWeights(
        DATA const& data, Matrix<double> const& costMatrix)
    { // инициализация средними значениями весов
        int k = costMatrix.getRows(), n = data.getSize();
        assert(k > 1 && k == findNClasses(data));
        Vector<double> classWeights(k), result(n);
        for(int i = 0; i < k; ++i)
            for(int j = 0; j < n; ++j)
                classWeights[i] += costMatrix(i, j);
        for(int i = 0; i < n; ++i) result[i] = classWeights[data.getY(i)];
        normalizeProbs(result);
        return result;
    }
public:
    template<typename DATA> AveCostLearner(DATA const& data,
        Matrix<double>const& costMatrix, PARAMS const& p = PARAMS()):
        model(data, findWeights(data, costMatrix), p) {}
    int predict(X const& x) const {return model.predict(x);}
};

```

Простой способ создать взвешенный  $A$  — использовать небольшой ансамбль мешков размера 5–15, основанный на весе. База  $A$  не нуждается в изменениях, потому что ансамбли легко настраиваются для прямой поддержки веса. Передискретизация на основе мешков лучше, чем классическая передискретизация, которая генерирует слишком много данных:

```

template<typename LEARNER, typename PARAMS = EMPTY>
class WeightedBaggedLearner
{
    Vector<LEARNER> models;
    int nClasses;
public:
    template<typename DATA> WeightedBaggedLearner(DATA const& data,
        Vector<double> weights, PARAMS const& p = PARAMS(), int nBags = 15):
        nClasses(findNClasses(data))
    {
        assert(data.getSize() > 1);
        AliasMethod sampler(weights);
        for(int i = 0; i < nBags; ++i)
        {
            PermutedData<DATA> resample(data);
            for(int j = 0; j < data.getSize(); ++j)
                resample.addIndex(sampler.next());
            models.append(LEARNER(resample, p));
        }
    }
    int predict(NUMERIC_X const& x) const
    {return RandomForest::classifyWork(x, models, nClasses);}
};

```

Бустинг по природе своей адаптирован к управлению затратами, потому что он использует обратную связь в последующих раундах. SAMME и другие аналитические решатели не применяются, поскольку формулы не распространяются на затраты, но вы можете модифицировать числовые решатели, такие как RMBoost. Вместо того, чтобы ограничивать бинарные потери логистикой, свяжите потери с поправкой на стоимость с логистикой с поправкой на стоимость. Например, для RMBoost скорректированный градиент = логистический градиент  $\times C$  (фактический класс, лучший неправильный класс):

```
Matrix<double> getEqualCostMatrix(int nClasses)
{
    Matrix<double> result(nClasses, nClasses);
    for(int i = 0; i < nClasses; ++i)
        for(int j = 0; j < nClasses; ++j) if(i != j) result(i, j) = 1;
    return result;
}

template<typename LEARNER = NoParamsLearner<DecisionTree, int>,
        typename PARAMS = EMPTY, typename X = NUMERIC_X> class RMBoost
{
    Vector<LEARNER> classifiers;
    int nClasses;
    struct BinomialLoss
    {
        Vector<Vector<double> > F;
        BinomialLoss(int n, int nClasses): F(n, Vector<double>(nClasses, 0))
        {}
        int findBestFalse(int i, int label)
        {
            double temp = F[i][label];
            F[i][label] = -numeric_limits<double>::infinity();
            double result = argMax(F[i].getArray(), F[i].getSize());
            F[i][label] = temp;
            return result;
        }
        double getNegGrad(int i, int label, Matrix<double>const& costMatrix)
        {
            int bestFalseLabel = findBestFalse(i, label);
            double margin = F[i][label] - F[i][bestFalseLabel];
            return costMatrix(label, bestFalseLabel)/(exp(margin) + 1);
        }
    };
};

public:
    template<typename DATA> RMBoost(DATA const& data, Matrix<double>
        costMatrix = Matrix<double>(1, 1), PARAMS const& p = PARAMS(),
        int nClassifiers = 100): nClasses(findNClasses(data))
    { // начальные веса основаны на средней стоимости
        if(costMatrix.getRows() != nClasses)
            costMatrix = getEqualCostMatrix(nClasses);
        int n = data.getSize();
        assert(n > 0 && nClassifiers > 0);
        BinomialLoss l(n, nClasses);
        Vector<double> dataWeights(n), classWeights(nClasses);
```



```

for(int i = 0; i < nClasses; ++i)
    for(int j = 0; j < nClasses; ++j)
        classWeights[i] += costMatrix(i, j);
for(int i = 0; i < n; ++i)
    dataWeights[i] = classWeights[data.getY(i)];
for(int i = 0; i < nClassifiers; ++i)
{
    normalizeProbs(dataWeights);
    AliasMethod sampler(dataWeights);
    PermutedData<DATA> resample(data);
    for(int j = 0; j < n; ++j) resample.addIndex(sampler.next());
    classifiers.append(LEARNER(resample, p));
    for(int j = 0; j < n; ++j)
    {
        l.F[j][classifiers.lastItem().predict(data.getX(j))] +=
            RMRate(i);
        dataWeights[j] = l.getNegGrad(j, data.getY(j), costMatrix);
    }
}
}
int predict(X const& x) const
{
    Vector<double> counts(nClasses, 0);
    for(int i = 0; i < classifiers.getSize(); ++i)
        counts[classifiers[i].predict(x)] += RMRate(i);
    return argMax(counts.getArray(), counts.getSize());
}
};

```

Поскольку затраты  $\in (0, 1]$ , можно определить стоимостную точность как 1-стоимостной риск. Они кривые и усредняются по нескольким наборам данных как обычные точности (рис. 26.23).

RF	RFProbC	RMBDT	SVM	SVMave15	SVM RMB15
0.963	0.999	0.969	0.964	0.981	0.990

Рис. 26.23. Сравнение производительности на  $S$  генерируется детерминистически, делая каждую недиагональную запись равной либо 0,01, либо 1

Таким образом, для вектора  $X$  случайный лес с вероятностным выходом кажется предпочтительным методом. В противном случае усиленный ансамбль SVM небольшого размера 15 работает хорошо. Реализация предназначена для вектора  $X$  для простоты:

```

class BoostedCostSVM
{
    RMBBoost<MulticlassSVM<>, pair<GaussianKernel, double> > model;
public:
    template<typename DATA> BoostedCostSVM(DATA const& data,
        Matrix<double> const& cost = Matrix<double>(1, 1)):
        model(data, cost, NoParamsSVM::gaussianMultiClassSVM(data), 15) {}
    int predict(NUMERIC_X const& x) const{return model.predict(x);}
};
typedef ScaledLearner<BoostedCostSVM, int, Matrix<double> > SBoostedCostSVM;

```

Однако нельзя ожидать, что форсированный SVM будет работать лучше, чем SVM, для бесплатного обучения. Для  $k = 2$  альтернативой повышению стоимости является метод средней стоимости, который хорошо работает с передискретизацией:

```
class AveCostSVM
{
    typedef pair<GaussianKernel, double> P;
    AveCostLearner<WeightedBaggedLearner<MulticlassSVM<>, P>, P> model;
public:
    template<typename DATA> AveCostSVM(DATA const& data,
        Matrix<double> const & cost = Matrix<double>(1, 1)):
        model(data, cost, NoParamsSVM::gaussianMultiClassSVM(data)) {}
    int predict(NUMERIC_X const& x) const {return model.predict(x); }
};

typedef ScaledLearner<AveCostSVM, int, Matrix<double> > SAveCostSVM;
```

Несмотря на привлекательность, общие методы так или иначе изменяют данные, что может создать проблемы. Например, повторная выборка дублирует некоторые примеры, потенциально заставляя  $A$  уделять слишком много внимания нерепрезентативным примерам. Использование вероятностей, как в случайном лесу, может быть проблематичным, если они неточны. Таким образом, модификации, специфичные для  $A$ , будут иметь наименьшую ошибку оценки, но изменение каждого  $A$  для обработки затрат требует много времени.

## 26.19. Несбалансированное обучение

В заданном наборе данных некоторые классы могут встречаться чаще, чем другие. Это может быть связано с естественным распределением данных или смещениями при отборе, поскольку экземпляры редких классов анализируются дороже. Например, при предсказании погоды в большую часть дней дождь не идет, поэтому отсутствие дождя всегда предсказывается достаточно точно.

Существуют два основных типа дисбаланса:

- ◆ глобальный — предположение о независимости и случайности нарушается, вызывая неестественный дисбаланс;
- ◆ местный —  $X$  имеет много пятен из нескольких примеров, которые окружены более многочисленными примерами разных классов. Это может случиться, даже если примеры являются независимыми и случайными, и для распространенного класса нужно достаточно сложное распределение.

В любом случае классификация будет иметь ошибку оценки из-за смещения полей в сторону меньшей группы, увеличивая ошибку обобщения. Лучшее решение обеих проблем — получить больше независимых случайных данных, если это возможно.

В некоторой степени можно исправить глобальный дисбаланс. Нам нужно максимально возместить риск, связанный с дисбалансом (см. [26.46]). Игнорировать дисбаланс можно для умеренно несбалансированных наборов данных, т. е. когда отношение примеров составляет порядка 2 к 1. Обычно трудно точно сказать, когда дисбаланс умеренный, а когда нет. Кроме того, важна разница между  $S$  и фактическим распределением данных, а не между  $S$  и всеми остальными равноклассными распределениями. Например, цифры

для почтового конверта берутся из почтовых индексов, где некоторые цифры менее вероятны, чем другие, из-за распределения населения и почтовых индексов, даже если наборы данных разумно уравновешены. Глобальный дисбаланс обычно является проблемой только в том случае, если он вызывает локальный дисбаланс. Например, если в классе-меньшинстве имеется 1000 или около того примеров, даже дисбаланс 1000 к 1 может не сказаться на хорошем  $A$ , потому что для достаточно простого распределения все исправления будут иметь достаточную поддержку. Чтобы понять это, рассмотрите требования согласованности  $k$ -NN. Также может быть полезно сравнить количество классов перед началом обучения. Некоторые  $A$ , такие как бустинг и SVM, кажутся от природы устойчивыми к локальному дисбалансу, т. к. строят отступы вокруг всех групп малых примеров.

Простое решение заключается в снижении затрат на обучение. Согласно народной теореме риск для выборок из сбалансированного распределения = обучению на основе исходного значения со стоимостью (прогнозируемое, фактическое) = (количеству экземпляров прогнозируемого класса / количество экземпляров фактического класса). Но, хотя затраты известны без ошибок оценки, затраты из пропорций классов, вероятно, не будут отражать распределение, из которого они были отобраны, если  $n$  не велико. Сокрытие, как правило, работает плохо (см. [26.46]). По той же причине индивидуальное изменение  $A$  с учетом распределения классов может придать ему слишком большой акцент, хотя это должно повысить производительность при сбалансированной точности, если тестовые данные имеют аналогичные пропорции дисбаланса.

Прямая передискретизация является более надежной в том смысле, что различные  $A$  не доверяют слепо пропорциям повторной выборки, а учатся на них настолько, насколько необходимо. Нужно, чтобы каждый класс был представлен в равной степени, т. е. с равным общим весом для любого класса:

```
template<typename DATA>
Vector<double> findImbalanceWeights(DATA const& data)
{
    int n = data.getSize(), properK = 0, nClasses = findNClasses(data);
    Vector<double> counts(nClasses);
    for(int i = 0; i < n; ++i) ++counts[data.getY(i)];
    for(int i = 0; i < nClasses; ++i) if(counts[i] > 0) ++properK;
    Vector<double> dataWeights(n, 0);
    for(int i = 0; i < data.getSize(); ++i)
        dataWeights[i] = 1.0/properK/counts[data.getY(i)];
    return dataWeights;
}
```

Взвешенный бэггинг позволяет делать то же самое с другими базовыми учащимися, такими как SVM:

```
class ImbalanceSVM
{
    WeightedBaggedLearner<MulticlassSVM<>,
        pair<GaussianKernel, double> > model;
public:
    template<typename DATA> ImbalanceSVM(DATA const& data): model(data,
        findImbalanceWeights(data), NoParamsSVM::gaussianMultiClassSVM(data)) {}
}
```

```
int predict(NUMERIC_X const& x) const{return model.predict(x);}
};
typedef ScaledLearner<NoParamsLearner<ImbalanceSVM, int>, int> SImbSVM;
```

Случайный лес расширяется напрямую — измените его начальную загрузку, чтобы использовать метод псевдонима, и пример как в бустинге:

```
class WeightedRF
{
    Vector<DecisionTree> forest;
    int nClasses;
public:
    template<typename DATA> WeightedRF(DATA const& data, Vector<double> const
        & weights, int nTrees = 300): nClasses(findNClasses(data))
    {
        assert(data.getSize() > 1);
        AliasMethod sampler(weights);
        for(int i = 0; i < nTrees; ++i)
        {
            PermutedData<DATA> resample(data);
            for(int j = 0; j < data.getSize(); ++j)
                resample.addIndex(sampler.next());
            forest.append(DecisionTree(resample, 0, true));
        }
    }
    int predict(NUMERIC_X const& x) const
    {
        return RandomForest::classifyWork(x, forest, nClasses);
    }
};
class ImbalanceRF
{
    WeightedRF model;
public:
    template<typename DATA> ImbalanceRF(DATA const& data, int nTrees = 300):
        model(data, findImbalanceWeights(data), nTrees) {}
    int predict(NUMERIC_X const& x) const{return model.predict(x);}
};
```

Повторная выборка может привести к ошибке оценки из-за дублирования примеров, поэтому это несовершенное, но, возможно, лучшее доступное решение для сильного дисбаланса (рис. 26.24).

RF 05 300	SSVM 10n10k_001_seps c10	ImbRF 300	ImbSVM
0.964	0.940	0.971	0.946

Рис. 26.24. Некоторые результаты сравнения производительности

Однако повторная выборка не заменяет хорошие независимые случайные данные, т. к. качество обучения зависит от качества данных, а мусор на входе дает мусор на выходе.

Для алгоритмов реального времени повторная выборка в случае дисбаланса проблематична, потому что нельзя разумно оценить пропорции классов.

Решение локального дисбаланса по существу безнадежно — в лучшем случае умный  $A$  может выделить разумные пределы вокруг небольших групп, но для этого нужно достаточно примеров, иначе лучше рассматривать небольшие группы как шум.

## 26.20. Выбор признаков

Для первой попытки удобной стратегией является использование поиска по оболочке. Случайный лес, вероятно, является лучшей базой  $A$  из-за его высокой точности и скорости. У него есть некоторые различия в производительности, но это не должно быть проблемой:

```
template<typename SUBSET_LEARNER = RandomForest> struct SmartFSLearner
{
    typedef FeatureSubsetLearner<SUBSET_LEARNER> MODEL;
    MODEL model;
public:
    template<typename DATA> SmartFSLearner(DATA const& data, int limit = 20):
        model(data, selectFeaturesSmart(SCVRiskFunctor<MODEL, Bitset<>, DATA>(
            data), getD(data), limit)) {}
    int predict(NUMERIC_X const& x) const {return model.predict(x);}
};
```

С учетом моих экспериментов с несколькими наборами данных в среднем выбираются 44% признаков с разумной сбалансированной точностью 0,938. Для данных о цветах выбирается только признак 3.

Для маленьких  $D$  и  $n$  может быть полезен алгоритм SVM, потому что он имеет слишком высокую точность и не имеет дисперсии, хотя и медленнее. Можно попытаться ускорить его, выбрав параметры после любого запуска, но это может испортить поиск, если для разных подмножеств нужны разные параметры.

Среди встроенных методов наиболее полезными являются дерево решений и линейный SVM. Например, линейные веса SVM, возможно, умноженные на среднее значение признака, определяют относительное влияние признаков на результат линейной комбинации, поэтому функции с большим влиянием должны быть более важными. Кроме того, поскольку добавление признаков к проблеме с линейным разделением не повышает точность, для данных, идеально классифицированных с помощью линейного SVM, можно удалить признаки до тех пор, пока они не перестанут быть линейно разделимыми.

## 26.21. Сравнение классификаторов

Основные экспериментальные исследования приведены в работах [26.14, 26.13 и 26.23 (особенно полная работа)]. В последних двух работах лучшим считается случайный лес, а в первой он на втором месте. SVM и нейронная сеть также преуспевают во всех трех, но не так сильно. Дерево решений, наивный байесовский алгоритм и  $k$ -NN неконкурентоспособны. Другие алгоритмы, которые также не обладают желаемыми свойствами, не имеют вариантов использования.

Далее приведено более простое сравнение нескольких наборов данных UCI (рис. 26.25). Некоторые из них уже разделены на обучение и тестирование, — те, которые не были разделены с использованием стратифицированного 20%-ного удержания.

sMeanNN	B RF SVM	SKNN oddlg	SSVM 10n10k 001 seps c10
0.842	0.973	0.910	0.940
SLSVMsgd100kcd100	mqk2nn	DT-z1 50	RF 05 300
0.917	0.927	0.871	0.964
NumericalBayesE			
0.820			

Рис. 26.25. Сравнение нескольких наборов данных UCI

Показанные результаты говорят, что вы можете ожидать наилучшей производительности от случайного леса и SVM, а нейронная сеть немного отстает. Выбор лучшего из них дает еще лучшую производительность:

```
template<typename X, typename Y> struct LearnerInterface
{
    virtual Y predict(X const& x) const = 0;
    virtual LearnerInterface* clone() const = 0;
};

template<typename LEARNER, typename X, typename Y>
struct TypeFreeLearner: public LearnerInterface<X, Y>
{
    LEARNER model;
    template<typename DATA> TypeFreeLearner(DATA const& data): model(data) {}
    Y predict(X const& x) const{return model.predict(x);}
    LearnerInterface<X, Y>* clone() const{return new TypeFreeLearner(*this);}
};

template<typename Y, typename X = NUMERIC_X>
class BestCombiner
{
    LearnerInterface<X, Y>* model;
    double risk;
public:
    BestCombiner(): model(0) {}
    BestCombiner(BestCombiner const& rhs): model(rhs.model->clone()) {}
    BestCombiner& operator=(BestCombiner const& rhs)
    {return genericAssign(*this, rhs);}
    template<typename LEARNER, typename DATA, typename RISK_FUNCUTOR>
    void addNoParamsClassifier(DATA const& data, RISK_FUNCUTOR const& r)
    {
        double riskNew = r(EMPTY());
        if(!model || riskNew < risk)
        {
            delete model;
            model = new TypeFreeLearner<LEARNER, X, Y>(data);
            risk = riskNew;
        }
    }
}
```

```

Y predict(X const& x) const { assert(model); return model->predict(x); }
~BestCombiner() { delete model; }
};

class SimpleBestCombiner
{
    BestCombiner<int> c;
public:
    template<typename DATA> SimpleBestCombiner(DATA const& data)
    {
        c.addNoParamsClassifier<RandomForest>(data, SCVRiskFunctor<
            NoParamsLearner<RandomForest, int>, EMPTY, DATA>(data));
        c.addNoParamsClassifier<SVM>(data, SCVRiskFunctor<
            NoParamsLearner<SVM, int>, EMPTY, DATA>(data));
    }
    int predict(NUMERIC_X const& x) const { return c.predict(x); }
};

```

Интересным результатом является то, что сочетание только случайного леса и SVM дает лучшую среднюю криволинейную сбалансированную точность 0,971, но кажется более разумным сравнить больше моделей для новой задачи. Например, вы можете использовать LSVM, но для общего использования наилучшая политика выбора требует обширного исследования со многими наборами данных.

Приведенные здесь сравнения не самые лучшие из возможных — обратите внимание, что мы сравниваем алгоритмы, поэтому использование перекрестной проверки вместо удержания уменьшает дисперсию. Но это заняло бы в несколько раз больше времени, поэтому такое сравнение не было приведено.

Другой  $A$  обычно полезен только для особых случаев, хотя существуют наборы данных, для которых они имеют максимальную производительность:

- ◆ ближайший сосед — для необычных функций расстояния, когда SVM не масштабируется или требует быстрого исследования;
- ◆ наивный байесовский метод — для крупномасштабных категориальных данных, где признаки достаточно независимы;
- ◆ линейный SVM — для крупномасштабных данных.

Существует дилемма, которая заключается в том, пытаться ли понять данные, изучая понятные шаблоны, или использовать классификаторы, которые «каким-то чудом» работают хорошо. Производительность может быть приемлемой/неприемлемой и хорошей/плохой. Проблемным результатом является плохое понимание неприемлемой производительности. Обычно понятные модели, такие как небольшие деревья решений или линейный метод опорных векторов, подходят только для очень малых  $D$ , что характерно для многих приложений, требующих понимания для человека, — например, решение о выдаче кредита. Разумная стратегия для нового набора данных:

1. Если  $X$  не является вектором, используйте SVM со специализированным ядром или  $k$ -NN со специальной функцией расстояния. Лучше использовать SVM из-за его более разреженной модели и лучшей производительности, но  $k$ -NN часто является первым используемым алгоритмом.
2. Если  $n$  слишком велико, используйте LSVM для числовых данных и наивный байесовский метод для категориальных данных.

3. Попробуйте использовать сильно обрезанное дерево решений, затем LSVM. Другие варианты не нужны, если производительность одного из рассмотренных достаточно высока, потому что результаты дают представление о данных. Дерево решений, по сути, является единственным быстрым  $A$  с использованием ресурсов  $o(n^2)$ , поэтому для исследовательского анализа это лучший метод.
4. Попробуйте эффективных учащихся «черного ящика»: выберите случайный лес, SVM или нейронную сеть, а можно просто выбрать случайный лес из-за его скорости. Последний вариант предпочтительнее из-за эффективности и может работать лучше, когда  $n$  слишком мало (возможно,  $< 100$ ). Чтобы выбрать лучший из нескольких  $A$ , сначала можно опробовать несколько простых вещей на одном и том же наборе данных. Это множественное тестирование, но оно не столь эффективно, потому что ясность предиктора является основным критерием отбраковки.
5. Если производительность недостаточно высока, посмотрите на данные и попытайтесь выяснить, почему обучение не удастся. Вероятно, нужно помочь  $A$ , выполнив проектирование функций или дополнив его логикой, специфичной для предметной области, такой как заведомо хорошая структура нейронной сети.
6. Если приведенные рекомендации недостаточно хороши, изучите методы, специфичные для рассматриваемой области и связанные с ней. Существуют многие другие специализированные  $A$ , полезные для конкретных областей, но не в целом, и они в этой главе не обсуждаются. Но вполне вероятно, что данные не содержат никакой полезной информации о метках.

## 26.22. Примечания по реализации

Все реализации требовали серьезного исследования и настройки параметров, несмотря на то, что они основывались на простых алгоритмах, которые обсуждаются во многих учебниках. В каждой есть что-то новое:

- ◆ матрица ошибок — я предпочитаю не рассчитывать доверительные интервалы по специфической для класса метрике;
- ◆ дерево решений — критерии отсека и ограничение глубины являются оригинальными, и я использую бинарные узлы;
- ◆  $k$ -NN — расчет  $k$  по умолчанию является оригинальным;
- ◆ наивный байесовский подход: позволять всем алгоритмам выполняться в реальном времени — это оригинальная идея;
- ◆ SVM — алгоритм оптимизации, основанный на поиске по дискретному компасу, является оригинальным и, кажется, хорошо работает за счет выбора хорошей отправной точки;
- ◆ случайный лес — выбор параметров оригинальный;
- ◆ бустинг — использование реализации численного повышения для экономичного обучения является оригинальным.

Комбинация случайного леса и SVM, пожалуй, наиболее сомнительна. Я бы, вероятно, использовал только случайный лес и один алгоритм для любых векторных данных.



## 26.23. Комментарии

Для оценки производительности в информационном поиске полнота и точность использовались для  $k = 2$  и вычислялись только для примеров с  $y = 1$ , который, предположительно, является интересующим классом. Таким образом, эти показатели можно рассматривать как потерю слишком большого количества информации, если важны оба класса.

Существуют и другие довольно популярные метрики производительности, зависящие от приложения (см. [26.55]):

- ♦ *F-мера* — определена для класса как гармоническое среднее значение точности и полноты, из которого он пытается создать единую комбинированную метрику. Но она имеет смысл только для задач с  $k = 2$ , где класс важен, а другое нет. Например, для онлайн-поиска важны только релевантные результаты, и вы не хотите ни пропустить хорошие результаты, ни получить нерелевантные;
- ♦ *площадь под ROC-кривой (AUC)* — применяется к бинарным классификаторам. Эта метрика дает возможность зафиксировать производительность по всем возможным матрицам затрат (см. [26.55]) и иногда пользуется популярностью у исследователей. Но для единой матрицы затрат метод сводится к чему-то вроде *F-меры*, только менее интуитивно понятной. Вычисление ее напрямую имеет смысл только для вероятностных бинарных классификаторов. Другие могут использовать моделирование для обучения затратам со случайно выбранными матрицами затрат, что хорошо работает при  $k = 2$ . Наборы инструментов, такие как Weka, вычисляют AUC автоматически, но их преимущество, если таковое имеется, по сравнению со сбалансированной точностью неясно, а сложность вычислений весьма велика и существенна. Также неясно, как распространить это на  $k > 2$ ;
- ♦ *каппа-метрика* технически является мерой согласия, дающей значения  $\in [-1, 1]$ , но в этом контексте измеряет скорректированную точность, не считая точности на примерах, которые, как ожидается, будут случайно классифицированы правильно. Например, любая  $f$ , которая предпочитает более представленные классы, будет случайно иметь искусственно высокую точность, и каппа-метрика может это обнаружить. Сравнение по каппа аналогично сравнению по точности, за исключением того, что каппа снижает приоритет более представленных классов. Как правило, производительность по сравнению с базовым методом полезна, поскольку она определяет, полезна ли модель. Например, любая модель прогнозирования погоды должна превосходить среднее историческое значение и вчерашнюю погоду. Каппа-метрика — это производительность не по неосведомленному  $A$ , который всегда возвращает класс большинства, а по неосведомленному конкретному  $A$ , что делает ее похожей на сбалансированную точность, но гораздо менее интуитивно понятной. Также эта метрика плохо распространяется на обучение с затратами, но сбалансированная точность вычисляется легко.

Для каппа и других метрик, которые нельзя назвать достоверными, единственный способ получить ее — за счет получения повторной выборки результатов набора тестов, создания соответствующих матриц достоверности и использования начальной загрузки для полученных метрик.

Многие методы могут давать эвристические вероятности, но не без проблем. Например, наивный Байес нуждается в нормализации, но сама модель обычно упрощенная. Логич-

стическая регрессия (обсуждается позже) также естественным образом выводит вероятности, но она точна только в том случае, если ее модель верна. Для дерева решений обычно определяется количество листьев, но результирующие вероятности неточны, если листья маленькие. Похоже, что вероятностный вывод случайного леса является единственным достаточно точным вариантом.

Аналогом размерности VC для  $k > 2$  является *размерность Натараджана* (см. [26.51]). Но границы концентрации, основанные на сложности Радемахера, для этого алгоритма еще не разработаны, и это мало как повлияет на концептуальное понимание.

Для наивного Байеса при моделировании числовых данных вместо биннинга можно использовать априорно-апостериорные методы байесовской статистики. Но простые распределения, например нормальное, задаются несколькими параметрами, и вы не сможете моделировать многопиковые распределения. Подсчет категориальных признаков с достаточным количеством категорий позволяет делать это, поэтому группирование, как правило, работает лучше (см. [26.55]).

Для  $k$ -NN другие естественные расширения бесполезны:

- ◆ взвешивание найденных соседей по функции расстояния — интуитивно следует взвешивать более близких соседей больше, возможно, используя  $w_i = \frac{1}{1 + d(x, \text{coseed}_i)}$ , но на практике это частично отменяет сглаживание  $k > 1$ .

Асимптотически оптимальным является отказ от взвешивания (см. [26.19]). *Критерий Кригинга* для регрессии берет среднее значение всех соседей, взвешенных по расстоянию, и работает хорошо, но, по-видимому, только для регрессии в малом  $D$ , например для пространственной статистики;

- ◆ удаление некоторых экземпляров — похоже, этот метод потенциально решает проблему с памятью. Но не существует хорошего способа выполнить это удаление, потому что бесполезными являются только экземпляры внутри полей и шумные неправильно помеченные экземпляры. Асимптотически выбор экземпляра не может снизить риск (см. [26.19]). Один из разумных методов под названием *IB3* плохо проходит тесты (см. [26.23]). Другие эвристики лучше с точки зрения точности, экономии памяти или производительности (см. [26.25]), но недостаточны, чтобы оправдать их использование, в основном из-за неэффективности, — обычно время выполнения равно  $O(n^2)$ . Теоретически полезные результаты (см. [26.27]) неприменимы на практике. Похоже, что только SVM правильно выбирает экземпляры при выборе опорных векторов.

*Хеширование с учетом местоположения* (Locality-Sensitive Hashing, LSH) (подробнее см. [26.1]) позволяет находить ближайших соседей в пределах фиксированного расстояния  $R$  с высокой вероятностью и более эффективно, чем при использовании структур данных точного поиска путем вычисления хешей, которые, вероятно, отображают близкие  $x$  в одну и ту же корзину. Хотя в некоторых приложениях (в основном не связанных с классификацией и использующих специализированные хеш-функции) они очень успешны, в целом же полезная реализация с евклидовым расстоянием проблематична. Основной попыткой является экономия памяти за счет сохранения только  $y$  и хешей и возврата большинства найденных меток или возврата к ближайшему среднему значению, если ничего не найдено.  $R$  и параметр « $k$ » находятся путем перекрестной проверки, а « $l$ » выставляется равным 10. В моих тестах метод оказался лишь немного

точнее ближайшего среднего. Любое улучшение потребует гораздо больше памяти и работает не намного лучше, чем обычный запрос  $k$ -NN, для которого VP- или  $k$ -d-дерево работают быстро, несмотря на плохой худший случай.

Для деревьев решений *индекс Джини* (см. [26.3]) является альтернативой энтропии. По сути, он дает вероятность того, что пример будет неправильно классифицирован. Производительность этот индекс имеет аналогичную.

Можно уменьшить коэффициент  $\lg(n)$  во время выполнения построения дерева решений путем предварительной сортировки примеров по каждому атрибуту (см. [26.55]), но дополнительное использование памяти и сложность кода того не стоят. Некоторые варианты деревьев решений допускают многовариантные деревья, в которых функция разделяется только один раз и больше не рассматривается. Но точки деления может быть трудно определить, потому что выбор, основанный на получении информации, становится предвзятым в пользу признаков с большим количеством разделений. Поправки, такие как *коэффициент усиления* (см. [26.3]), решают эту проблему лишь частично. Кроме того, реализация более сложная.

Было предложено множество методов обрезки дерева решений. По результатам экспериментов ни один из них не оказался лучше во всех случаях (см. [26.3]). Большинству требуется отдельный проверочный набор, который нельзя повторно использовать при создании окончательного дерева, что приводит к более слабой общей модели. Наиболее известным методом является *отсечение сложности по стоимости*: сгенерируйте несколько все более простых деревьев, отбрасывая «наименее полезный» узел за раз, и выберите самое простое дерево, производительность которого находится как минимум в пределах одного стандартного отклонения от производительности наиболее точного дерева. Поскольку размерность VC-дерева — это количество листьев, метод имеет смысл. Основной метод, использующий тот же набор данных, — *пессимистическое удаление ошибок*. Реализованный в C4.5, он использует биномиальные пределы достоверности из табличной интерполяции при  $\alpha = 0,25$ , отклоняя дерево, если производительность дерева на этом уровне не улучшает производительность узла. В более новой реализации Века используется интервал оценки Уилсона (см. [26.55]).

Мои тесты показывают, что возможно применить расширение, называемое *модельным деревом*, которое заменяет большинство листьев более умными моделями, такими как логистическая регрессия (см. [26.49]). Несмотря на интуитивную привлекательность, модельные деревья, по-видимому, экспериментально не исследовались.

Некоторые очевидные недостатки метода:

- ◆ потери интерпретируемости;
- ◆ листья должны содержать много данных, чтобы можно было избежать ошибок оценки, которые для большей части данных проблемой не являются;
- ◆ разделение гораздо менее эффективно, потому что нужно строить конечные модели для любого разделения вместо постепенного обновления энтропии.

Еще одна возможность (называемая *наклонным деревом*) заключается в расщеплении линейной комбинации нескольких переменных. Как правило, это выполняется сложно, и проверенные подходы не превосходят обычное дерево решений (см. [26.23]). Основная цель — создание деревьев с меньшим количеством узлов, что позволяет сделать дерево более интерпретируемым, но это субъективно, поскольку требуется больше

оценок для сложных разбиений. Полезные алгоритмы для многомерного разбиения основаны на эвристике и могут быть медленными при больших  $k$  (см. [26.54]).

Можно попробовать разделить пространство узлов наклонного дерева с помощью LSVM. Цель этого — имитировать что-то вроде BSP-дерева (погуглите), чтобы в случаях, когда выполнить линейное разделение проблематично, дальнейшее разделение позволило улучшить производительность. Но в моих экспериментах сделать этого не удалось, потому что из-за собственной и обычно высокой точности линейного SVM разделенные данные будут иметь сильный дисбаланс классов. Поэтому LSVM 2-го и более высокого уровня, вероятно, научатся выбирать класс большинства в основном потому, что SGD очень чувствителен к дисбалансу классов, и передискретизация исправить проблему не помогает.

Несколько глупое расширение деревьев — это *правила* (см. [26.55]). Идея состоит в том, что правила легче понять, поэтому можно попробовать свести дерево к списку правил. Но компактное дерево может раздуться на множество правил. Деревья легко понять, когда они маленькие, а правила никогда не бывают меньше деревьев.

Из-за простоты линейных разделителей, таких как линейный SVM, существует много разновидностей:

- ◆ *логистическая регрессия* — эквивалентна нейронной сети без скрытых слоев и линейной регрессии, которая пытается предсказать  $\Pr(y_i = 1|x_i)$  на основе логит-функции. После нескольких десятилетий использования этот метод стал популярен в статистике и очень похож на линейный SVM. Задача оптимизации в обоих случаях сводится к минимизации штрафа( $w$ ) +  $\sum L(y_i, f(x_i))$ . С помощью  $L = \ln(1 + e^{y_i f(x_i)})$  приходим к логистической регрессии. Оба метода с точки зрения ошибки обобщения обычно работают схожим образом. Теоретически SVM является предпочтительным, потому что он не оценивает вероятности, игнорируя примеры за пределами отступа и, таким образом, работает надежнее. Но решить логистическую регрессию с высокой точностью проще, потому что ее цель = выпуклая функция + дифференцируемая функция, для которых в настоящее время существуют более эффективные решатели (см. [26.37]). Кроме того, координатный спуск в логистической регрессии гарантированно сходится, в отличие от SVM (см. [26.29]);
- ◆ *линейный дискриминантный анализ* (Linear Discriminant Analysis, LDA) — предполагает, что примеры с разными метками берутся из многомерных нормальных распределений с одинаковой ковариацией. Метод хорошо работает для многих задач, очень эффективен, работает в реальном времени и не зависит от порядка примеров, в отличие от SGD. Но предположения о нормальности и одинаковой ковариации на практике выполняются редко, и логистическая регрессия более надежна, поскольку делает строго меньше предположений, а линейность логарифмических шансов подразумевается нормальностью (см. [26.28]). Кроме того, необходимо выполнение условия  $n \geq D$ , чтобы LDA работал как есть. Для больших  $D$ , даже с модификациями алгоритма, оценка ковариационной матрицы затруднительна. Тем не менее LDA может иметь преимущество для онлайн-обучения;
- ◆ линейный SVM и линейный SVM с квадратичными hinge-потерями (логистическая регрессия) основаны на регуляризации  $L_2$ . Это приводит к более простым задачам оптимизации из-за дополнительной дифференцируемости, но в остальном преимуществ не дает. Несмотря на гораздо более эффективные алгоритмы решения с высо-

кой точностью в некоторых случаях (см. [26. 56]), с точки зрения ошибки обобщения SGD трудно добиться такого улучшения, чтобы можно было предпочесть более сложный подход.

Остерегайтесь того, что в некоторых источниках используется формулировка «штраф + сумма/ $n$ ». Этот и несколько других вариантов составляют другую задачу с другими свойствами, поэтому не следует выбирать их по сравнению со стандартным SVM.

В некоторых источниках упоминается устаревший *алгоритм персептрона*. Он предназначен для обучения нейронных сетей без скрытых слоев, работает на заданной пользователем небольшой постоянной скорости обучения и, в отличие от SGD, не сходится, если примеры не являются линейно разделимыми.

Для алгоритма SMO значения точности завершения, максимальное количество итераций и точность выбора опорных векторов являются эвристическими, но в то же время в настоящий момент они наиболее известны и надежно работают для правильно масштабированных данных и ядер. Точность завершения, равная 0,001, вызывает подозрения, потому что она абсолютна и не учитывает относительные  $C$  и  $b$ . Тем не менее для эффективности значение должно быть самым большим, т. к. его уменьшение не приведет к заметному улучшению обобщения для подавляющего большинства задач, и трудно улучшить испытанную стратегию LIBSVM. Сообщается, что использование  $b = 0$  приводит к меньшему времени выполнения и лучшим критериям завершения (см. [26.53]), но требует дополнительных исследований.

В другом методе оптимизации, используемом LIBSVM,  $i$  и  $j$  выбираются на основе информации второго порядка (см. [26.9]). Этот метод быстрее, чем максимальная нарушающая пара, но не существенно. Полезны и другие оптимизации (см. [26.16]).

Ядерный SVM не может правильно работать в реальном времени, потому что использует  $O(n)$  память для больших  $n$ . Необходимо поддерживать набор примеров и в конечном итоге отбрасывать те, которые якобы не могут быть опорными векторами. Если большие затраты памяти не являются проблемой, среди многих алгоритмов (см. [26.52, 26.21]) метод *LASVM* (см. [26.8]) в настоящее время является наиболее эффективным подходом. Кроме того, можно решить SVM в первичном случае. В частности, теорема о представителе напрямую применима к SVM при  $b = 0$  (также косвенно к  $b \neq 0$ ), потому что минимизация обратной маржи + hinge-риск имеют именно эту форму. Затем можно задействовать SGD или *ядерный PEGASOS*, но, в отличие от LASVM и SMO, поскольку первичное значение недифференцируемо, они не могут использовать информацию о градиенте для выбора переменных для оптимизации. Оставляя множество опорных векторов равными 0, LASVM находит разреженное решение, в отличие от других.

Поскольку большие значения  $C$  соответствуют большим отступам, было бы интересно изменить цель выбора параметра, чтобы отказаться от некоторой точности в пользу большего  $C$  или, по крайней мере, разрешить связи таким образом. Этот вопрос, по всей видимости, является неизученным.

И для ядерного, и для линейного SVM и их вариантов при выборе значения  $C$  или  $l$ , кажется, лучше начать с решения ранее решенной задачи на эффективность. Но для регуляризации многих решателей удобнее делать все стартовые значения равными 0, потому что сделать начальное значение 0, как правило, проще, чем установить его обратно на 0.

Поиск по сетке и моя эвристическая настройка — это лишь некоторые из методов оптимизации параметров. Многие другие рассмотрены в работе [26.33]. Автоматизированный выбор параметров, для которого SVM — важный вариант использования, в настоящее время является очень активной областью исследований.

Поскольку использование хорошего ядра вроде гауссиана всегда работает хорошо, можно попытаться автоматически выбрать ядро из данных (см. [26.26]). Но преимущества этого подхода на практике неясны в основном из-за неэффективности наиболее известных в настоящее время подходов.

Случайный лес похож на бэггинг, объединенный с *методом случайных подпространств*, который обучает ансамбли классификаторов на случайных подмножествах признаков. Последнее применимо к любому основанию  $A$ , а первое — только к деревьям, но корреляция между ними гораздо меньше, поскольку низкое смещение и высокая дисперсия деревьев делают их идеальными для случайных лесов, а на другие  $A$  этот метод не распространяется.

Было предпринято несколько попыток улучшения случайного леса, но все они имеют свои недостатки. В *лесе вращения* (см. [26.36]) используется разреженная матрица для поворота  $x$  и генерируется дерево решений результата, из которых создается лес. Идея довольно интересна, но вращение происходит медленно, требует масштабирования данных и, как правило, не улучшает случайные леса (см. [26.23]). Еще одна возможность состоит в том, чтобы при построении дерева выбрать следующую функцию на основе некоторой функции того, насколько полезной является эта функция, но результат получается аналогичен бэггингу.

О мощности аппроксимации нейронных сетей известно больше (см. [26.50]), но мало что из этой информации полезно. Имейте в виду, что некоторые такие результаты, по-видимому, не зависят от  $D$ , но константы в различных границах зависят, и нейронные сети не преодолевают проклятие размерности. Существуют асимптотические нижние границы приближения функций, зависящие от  $n$  и  $D$  (см. [26.47]).

SGD — не единственный способ обучения нейронных сетей. Пакетные методы вычисляют градиенты на основе всех примеров и используют общие методы оптимизации, такие как L-BFGS, который сходится сверхлинейно (см. главу 24. *Численная оптимизация*), что позволяет получать решения с высокой точностью. Но для больших  $n$  вычисление градиентов обходится дорого, а решения как высокой, так и низкой точности имеют одинаковое обобщение, поэтому предпочтение следует отдавать SGD. Вопреки интуиции большое количество скрытых нейронов работают лучше, чем малое количество, возможно потому, что из-за случайной инициализации мы получаем более высокую вероятность обнаружить хороший паттерн в некоторых нейронах, в то время как остальные просто остаются неэффективными, не внося дополнительной ошибки оценки.

Другой подход к упорядочению сети заключается в ранней остановке. Но риск может уменьшиться еще больше после увеличения (см. [26.44]), поэтому конкретного способа сделать такую остановку нет. Потеря веса более безопасна.

Сети прямой связи используются чаще других, но это не единственный вариант. *Рекуррентная сеть* позволяет нейронам обучаться на собственных выводах (т. е. заниматься самокритикой), но эта сеть сильнее страдает от затухающих градиентов. Тем не менее они недавно начали использоваться как часть глубокого обучения, особенно для распознавания речи. *Сеть RBF* использует ядра на входном слое, но работает не очень хоро-

шо (см. [26.23]), а ядерный SVM лучше справляется с нелинейностью. *Байесовская нейронная сеть* продемонстрировала максимальную производительность на нескольких многомерных наборах данных (см. [26.28]). Идея состоит в том, чтобы иметь нормальные  $(0, \sigma)$  априорные веса для некоторого  $\sigma$ , например 1. Предполагая, что выходные данные независимы, вычисляем  $\text{Pr}(y_i|x_i, \text{веса})$ . Это позволяет сделать вывод с помощью выборки MCMC из апостериорного значения. Указанный метод очень медленный, но кажется подходящим для решаемых задач (см. [26.28]). Сложность и медлительность этой сети препятствуют ее повсеместному использованию. Можно было бы получить те же результаты от усреднения нескольких сетей, оптимизированных с использованием глобального поиска с уменьшением веса, но этот метод тоже медленный. Поскольку MCMC выполняет оптимизацию и усреднение одновременно, возможно, повторный локальный поиск может делать то же самое, т. е. усреднять все свои локально оптимизированные результаты по переходам, но неясно, как найти хороший шаг перехода.

Другой, возможно, более популярный способ глубокого обучения — *ограниченная машина Больцмана* (Restricted Boltzmann Machine, RBM). Но автокодировщик проще, и в настоящее время не существует экспериментальных сравнений, в которых один из них показал себя лучше другого, — см. [26.44, 26.21 и 26.5], где приведено дальнейшее обсуждение тонкостей нейронных сетей. Также интересно использовать неконтролируемый вывод из глубокой сети в качестве признаков для другого  $A$ .

Когда что-то касается популярных новых алгоритмов, таких как глубокое обучение, важно уметь фильтровать хайп, который поднимают в популярных СМИ. В новостях часто пишут, что какая-то компания инвестировала несколько миллиардов долларов в применение глубокого обучения — например, в прототип беспилотного автомобиля. Скорее всего, это не более чем реклама того, какие они «крутые». Я был бы удивлен, если бы через несколько десятилетий глубокое обучение стало более важным, чем, например, оптимизация с помощью линейного программирования, которая сэкономила миллионы компаниям-первопроходцам, а сегодня уже не является горячей темой и требует большого количества инженерных разработок для эффективной работы. Один из советов о глубоком обучении, который я прочитал, звучит так: «Не пытайтесь повторить это дома». Тем не менее, кажется, что классические «поверхностные» методы, такие как SVM, были вдоль и поперек изучены, а «глубокие» модели имеют потенциал для улучшения.

Было предложено много вариантов бустинга, но никакие обширные исследования еще не подтвердили их достоинства. Алгоритм *AdaBoost.M1* использует AdaBoost для  $k > 2$  как есть, за исключением запрета переворачивания ответов, но требует базовой точности  $> 0,5$ . *AdaBoost.MH* и другие алгоритмы, основанные на выходных кодах (см. [26.59]), допускают более слабых учеников, но являются более сложными, чем SAMME, и не было доказано, что они работают лучше. *AdaBoost.L* (см. [26.59]) использует биномиальную потерю с аналитически полученными оптимальными весами, улучшая более ранний *LogitBoost*, но, согласно опубликованным (см. [26.59]) и моим тестам, он работает аналогично AdaBoost.M1, и нет причин использовать его вместо SAMME. Логистические потери не сильно отличаются от экспоненциальных, потому что в обоих примерах веса падают экспоненциально быстро.

Несмотря на то что метод бустинга градиента не подходит для задачи классификации, *повышение градиентного дерева* работает хорошо (см. [26.28]). Идея состоит в том, чтобы использовать метод OVA и обучить  $k$  деревьев регрессии малой глубины. Затем

для любого примера объединить их результаты с использованием полиномиальной потери в один балл  $= 1$  — предполагаемая вероятность правильного класса. Полиномиальная потеря также напрямую дает вероятности, но у них есть числовые проблемы, если не используется надлежащая нормализация. Однако:

- ◆ метод неэффективен — нужно  $k$  деревьев, и обучение каждого минимизации полиномиального градиента потерь не позволяет использовать вычисление приращения, а поиск полиномиальных минимизаторов потерь требует численной минимизации. Использование малой глубины не помогает;
- ◆ как утверждается в работе [26.23], случайный лес по-прежнему побеждает;
- ◆ рекомендуется использовать небольшой постоянный вес дерева, поскольку оптимальные веса могут привести к существенному переобучению, что очень подозрительно;
- ◆ не проводилось исследований, в которых этот метод сравнивался бы с другими вариантами бустинга на многих наборах данных.

На практике нет причин, по которым обычно предпочтение отдается бустингу, а не случайному лесу, который устойчив к шуму. Более сложные алгоритмы, такие как *BrownBoost* (см. [26.59, 26.17]), тоже сопротивляются шуму, в конце концов отказываясь от некоторых примеров (теряя выпуклость), но экспериментальных доказательств в их пользу не существует.

Вы можете сочетать нерелевантные  $A$  с использованием *голосования взвешенным большинством*, где база  $A$  с точностью  $a$  имеет относительный вес  $\ln\left(\frac{a}{1-a}\right)$

(см. [26.36]). Этот метод менее эффективен, чем выбор лучшего, из-за необходимости обучать всех и использовать результаты всех. К тому же он не лучше. Причина отсутствия повышения производительности, вероятно, заключается в том, что, в отличие от однородных комбинаций, таких как случайный лес, базовые учащиеся не должны иметь низкую корреляцию. На самом деле не существует оптимального комбинированного метода из-за эффектов NFL (см. [26.32]).

Вы можете адаптировать деревья решений для задачи онлайн-обучения. Типичным методом является *VLDT* (см. [26.24], а также [26.7], где описаны недавние работы). Его упрощенная версия, которая также применяется к непрерывным меткам, рекурсивно буферизует примеры в каждом листе, пока не будет достигнуто  $m$  (возможно, 100), разбивает лист и отбрасывает его примеры. Отбрасывание делает обрезку ненужной, потому что ошибка оценки не возникает из-за повторного использования примеров для дальнейшего разбиения. Одна из проблем заключается в том, что у вас может получиться отсортированный случайный поток с  $m$  примерами одного и того же класса, и в этом случае необходимо либо увеличить  $m$ , либо отбросить большинство, пока не будут выполнены некоторые критерии баланса. Затраты памяти могут стать проблемой, и хранение листовых буферов на диске может привести к 1 вводу/выводу на обновление в худшем случае, хотя использование большого кеша LRU поможет с этим справиться. Это также работает для деревьев регрессии (см. главу 27. *Машинное обучение: регрессия*).

Также можно выполнять бэггинг (и случайный лес) онлайн, рисуя кратность каждого примера из соответствующего распределения Пуассона (см. [26.24]). Но поскольку буферы дерева решений не любят множественности, это решение сомнительно.



Обучение с учетом затрат требует дополнительных экспериментов и исследований методов, подходящих для различных случаев, а также теоретической базы, которой в настоящее время не хватает. Интересный вариант состоит в том, чтобы ввести класс «не знаю». Дело в том, что незнание обычно обходится намного дешевле, чем принятие неправильного решения. Это может помочь снизить  $E[\text{риск затрат}]$ , но для решения проблемы «не знаю» потребуются модификации, специфичные для  $A$ . Общий подход состоит в том, чтобы изменить  $RMBoost$  так, чтобы он возвращал информацию о том, достаточно ли мал общий отступ, используя, возможно, некоторую константу с перекрестной проверкой.

Интересная идея — перенумеровать примеры, на которых ошибки обходятся дорого. Это реализуется в методе *Metacost* (см. [26.55]). Перемаркировка данных может показаться проблематичной, поскольку она создает дополнительный шум в метках, но в этом нет ничего плохого. Более фундаментальная проблема заключается в использовании мешков для оценки вероятностей. Оценки точны только для неустойчивых  $A$ , которых немного, а для деревьев решений случайный лес работает намного лучше. Если цель состоит в том, чтобы создать понятное дерево решений, использование *Metacost* со случайным лесом вместо бэггинга является хорошим решением.

Для исправления дисбаланса классическим решением является *избыточная выборка*, т. е. случайное увеличение числа примеров миноритарных классов путем повторной выборки. Такой подход может вызвать слишком большое дублирование и переобучение и, в отличие от повторной выборки ансамбля, приводит к единственной повторной выборке. Этот метод использовать не рекомендуется, несмотря на то, что он быстрее. *SMOTE* (см. [26.40]) пытается решить проблему переобучения путем экстраполяции известных примеров, что кажется глупым, несмотря на хорошую производительность, и работает только для вектора  $x$ . Экспериментально изучить дополнительный риск из-за дисбаланса сложно, потому что нужно учитывать множество возможностей, таких как  $n$ , коэффициент дисбаланса, конкретные локальные кластеры и т. д.

*Графические модели* — это класс алгоритмов, обобщающих наивный байесовский подход. Пример такой модели — *байесовская сеть*. Она пытается более точно оценить вероятности, предполагая зависимости между группами признаков (см. [26.48]). Например, в изображении соседние пиксели являются зависимыми, а удаленные — нет. Конечно, учитывая то, что представляет собой изображение, даже удаленные пиксели связаны между собой, поэтому такой метод не кажется подходящим для машинного обучения. Также трудно выяснить, какие функции зависят от других. Хотя некоторые графические модели успешны в определенных областях, таких как обработка естественного языка, где существенные предварительные знания предполагают структуру, они не являются «черным ящиком», поскольку автоматически и эффективно найти хорошую структуру из данных проблематично. Типичным подходом является итеративное группирование коррелированных переменных с использованием некоторой меры корреляции, такой как взаимная информация, или выполнение глобальной оптимизации с использованием  $MDL$  в качестве целевой функции. У этих и других есть многочисленные недостатки, связанные как с ошибками оценки, так и с ошибками аппроксимации, необходимость дискретизации непрерывных переменных способствует обоим. Кроме того, точная оценка вероятностей решает более общую проблему, увеличивая ошибку оценки. Байесовские сети кажутся лучше приспособленными к вероятностному моделированию, управляемому экспертами в предметной области, где есть четкие последовательности событий в логическом порядке.

## 26.24. Советы по дополнительной подготовке

- ◆ Улучшить расчет сбалансированной точности для проверки отсутствия меток.
- ◆ Модифицируйте  $k$ -NN, чтобы использовать хеш-таблицу для поиска класса большинства.
- ◆ Проведите больше экспериментов с методами обрезки дерева решений. Особенно интересной была бы логика типа MDL. Любое решение должно распространяться на дерево регрессии.
- ◆ Переключите SVM на кеш LRU заданного пользователем размера.
- ◆ Создайте общую нейронную сеть, используя ReLU для скрытых нейронов и 3–5 слоев. Улучшает ли это производительность?
- ◆ Случайный лес имеет естественный механизм ранжирования признаков — признаки, выбранные в верхних узлах деревьев, являются наиболее полезными. Исследования детализируют конкретный алгоритм и сравнивают результат с выбором функций на основе методов-оболочек.

## 26.25. Список рекомендуемой литературы

- 26.1. Andoni A. & Indyk P. (2008). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1), 117–122.
- 26.2. Anthony M., & Bartlett P. L. (2009). *Neural Network Learning: Theoretical Foundations*. Cambridge University Press.
- 26.3. Aggarwal C. C. (2014). *Data Classification: Algorithms and Applications*. CRC Press.
- 26.4. Bache K. & Lichman M. (2013). *UCI Machine Learning Repository* [<http://archive.ics.uci.edu/ml>]. University of California. Accessed 10/19/2014.
- 26.5. Bengio Y., Goodfellow I. J., & Courville A. (2016). *Deep Learning*. MIT Press.
- 26.6. Biau G., Devroye L., & Lugosi G. (2008). Consistency of random forests and other averaging classifiers. *The Journal of Machine Learning Research*, 9, 2015–2033.
- 26.7. Bifet A., Gavalda R., Holmes G., & Pfahringer B. (2018). *Machine Learning for Data Streams: With Practical Examples in MOA*. MIT Press.
- 26.8. Bordes A., Ertekin S., Weston J., & Bottou L. (2005). Fast kernel classifiers with online and active learning. *The Journal of Machine Learning Research*, 6, 1579–1619.
- 26.9. Bottou L. (Ed.). (2007). *Large-scale Kernel Machines*. MIT Press.
- 26.10. Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT2010* (pp. 177–186). Physica-Verlag HD.
- 26.11. Breiman L. (1996). Heuristics of instability and stabilization in model selection. *The annals of statistics*, 24(6), 2350–2383.
- 26.12. Breiman L. (2001). Random forests. *Machine learning*, 45(1), 5–32.
- 26.13. Caruana R., Karampatziakis N., & Yessensalina A. (2008). An empirical evaluation of supervised learning in high dimensions. In *Proceedings of the 25th international conference on Machine learning* (pp. 96–103). ACM.
- 26.14. Caruana R., & Niculescu-Mizil A. (2006). An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning* (pp. 161–168). ACM.

- 26.15. Cérou F., & Guyader A. (2006). Nearest neighbor classification in infinite dimension. *ESAIM: Probability and Statistics*, 10, 340–355.
- 26.16. Chang C. C., & Lin C. J. (2011). LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3), 27.
- 26.17. Cheamanunkul S., Ettinger E., & Freund Y. (2014). Non-convex boosting overcomes random label noise. *arXiv preprint arXiv:1409.2905*.
- 26.18. Daniely A., Sabato S., & Shwartz S. S. (2012). Multiclass learning approaches: a theoretical comparison with implications. In *Advances in Neural Information Processing Systems* (pp. 485–493).
- 26.19. Devroye L., Györfi L., & Lugosi G. (1996). *A Probabilistic Theory of Pattern Recognition*. Springer.
- 26.20. Domingos P. (2000). A unified bias-variance decomposition. In *Proceedings of 17th International Conference on Machine Learning*. Stanford CA Morgan Kaufmann (pp. 231–238).
- 26.21. Du K. L., & Swamy M. N. S. (2019). *Neural Networks and Statistical Learning*. Springer.
- 26.22. Elliott D. L. (1993). A better activation function for artificial neural networks.
- 26.23. Fernández-Delgado M., Cernadas E., Barro S., & Amorim D. (2014). Do we need hundreds of classifiers to solve real world classification problems?. *The Journal of Machine Learning Research*, 15(1), 3133–3181.
- 26.24. Gama J. (2010). *Knowledge Discovery from Data Streams*. CRC Press.
- 26.25. García S., Luengo J., & Herrera F. (2014). *Data Preprocessing in Data Mining*. Springer.
- 26.26. Gönen M., & Alpaydm E. (2011). Multiple kernel learning algorithms. *The Journal of Machine Learning Research*, 12, 2211–2268.
- 26.27. Gottlieb L. A., Kontorovitch A., & Nisnevitch P. (2014). Near-optimal sample compression for nearest neighbors. In *Advances in Neural Information Processing Systems* (pp. 370–378).
- 26.28. Hastie T., Tibshirani R., & Friedman J. (2009). *The Elements of Statistical Learning*. Springer.
- 26.29. Hastie T., Tibshirani R., & Wainwright M. (2015). *Statistical Learning with Sparsity: The Lasso and Generalizations*. CRC Press.
- 26.30. Hornik K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2), 251–257.
- 26.31. Hsu C. W., Chang C. C., & Lin C. J. (2010). *A practical guide to support vector classification*.
- 26.32. Hu R., & Damper R. I. (2008). A ‘no panacea theorem’ for classifier combination. *Pattern Recognition*, 41(8), 2665–2673.
- 26.33. Hutter F., Kotthoff L., & Vanschoren J. (2019). *Automated Machine Learning-Methods, Systems, Challenges*. Automated Machine Learning.
- 26.34. Hyafil L., & Rivest R. L. (1976). Constructing optimal binary decision trees is NP-complete. *InformationProcessing Letters*, 5(1), 15–17.
- 26.35. Kohavi R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI (Vol. 14, №. 2, pp. 1137–1145)*.
- 26.36. Kuncheva L. I. (2014). *Combining Pattern Classifiers: Methods and Algorithms*. Wiley.
- 26.37. Lee J. D., Sun Y., & Saunders M. A. (2014). Proximal Newton-type methods for minimizing composite functions. *SIAM Journal on Optimization*, 24(3), 1420–1443.
- 26.38. Leshno M., Lin V. Y., Pinkus A., & Schocken S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6), 861–867.
- 26.39. Long P. M., & Servedio R. A. (2010). Random classification noise defeats all convex potential boosters. *Machine Learning*, 78(3), 287–304.
- 26.40. López V., Fernández A., García S., Palade V., & Herrera F. (2013). An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics. *Information Sciences*, 250, 113–141.

- 26.41. Mandziuk J. (2010). Knowledge-Free and Learning-Based Methods in Intelligent Game Playing. Springer.
- 26.42. Margineantu D. D. (2001). Methods for cost-sensitive learning. (Doctoral dissertation, Oregon State University).
- 26.43. Mohri M., Rostamizadeh A., & Talwalkar A. (2018). Foundations of Machine Learning. MIT Press.
- 26.44. Montavon G., Orr G. B., & Müller K. R. (2012). Neural Networks: Tricks of the Trade. Springer.
- 26.45. Orriols-Puig A., Macia N., & Ho T. K. (2010). Documentation for the data complexity library in C++. Universitat Ramon Llull, La Salle, 196.
- 26.46. Prati R. C., Batista G. E., & Silva D. F. (2014). Class imbalance revisited: a new experimental setup to assess the performance of treatment methods. Knowledge and Information Systems, 1–24.
- 26.47. Ripley, Brian D. (1996). Pattern Recognition and Neural Networks. Cambridge University Press.
- 26.48. Russell S. J., Norvig P. (2020). Artificial Intelligence: a Modern Approach. Prentice Hall.
- 26.49. Rusch T. (2012). Recursive Partitioning of Models of a Generalized Linear Model Type. WU Vienna University of Economics and Business.
- 26.50. Scarselli F., & Tsoi A. C. (1998). Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results. Neural networks, 11(1), 15–37.
- 26.51. Shalev-Shwartz S., & Ben-David S. (2014). Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press.
- 26.52. Shawe-Taylor J., & Sun S. (2011). A review of optimization methodologies in support vector machines. Neurocomputing, 74(17), 3609–3618.
- 26.53. Steinwart I., Hush D., & Scovel C. (2011). Training SVMs without offset. The Journal of Machine Learning Research, 12, 141–202.
- 26.54. Truong A. K. Y. (2009). Fast Growing and Interpretable Oblique Trees via Logistic Regression Models. University of Oxford.
- 26.55. Witten I. H., Frank E., & Hall M. A. (2016). Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann.
26. 56. Yuan G. X., Ho C. H., & Lin C. J. (2012). Recent advances of large-scale linear classification. Proceedings of the IEEE, 100(9), 2584–2603.
- 26.57. Zadrozny B., Langford J., & Abe N. (2003). Cost-sensitive learning by cost-proportionate example weighting. In Data Mining, 2003. ICDM 2003. Third IEEE International Conference on (pp. 435–442). IEEE.
- 26.58. Zhu J., Zou H., Rosset S. & Hastie T. (2009) Multi-class AdaBoost. Statistics and Its Interface. (Vol. 2, pp. 349–360).
- 26.59. Schapire, Robert E. Boosting: foundations and algorithms / Robert E. Schapire and Yoav Freund. The MIT Press, Cambridge, Massachusetts, London, England, 1012.

## 27. Машинное обучение: регрессия

### 27.1. Введение

Регрессия строго сложнее, чем классификация, но при этом полезнее. Читатель должен быть знаком с понятием линейной регрессии из курса статистики.

Теоретически *регрессия* — это аппроксимация стохастической многомерной функции (стохастической из-за возможного шума  $y$ ), за исключением того, что последняя проще, потому что может запрашивать функцию в любой точке. В задаче регрессии особенно сложно выполнить экстраполяцию, потому что для регионов, достаточно далеких от тех, по которым имеются данные, в лучшем случае можно предположить, что неизвестная функция имеет значение, равное средневзвешенному значению известных точек. В этой главе рассматривается представление машинного обучения, но также можно использовать представление статистики. В работе [27.2] приведено хорошее суждение этой идеи.

Из-за проклятия размерности локальный базисный набор чего-то вроде линейных патчей не масштабируется в  $D$ . Таким образом, все методы являются «полулокальными» в том смысле, что они делают глобальные предположения и пытаются моделировать локальное поведение в субрегионах с достаточной поддержкой данных, т. е. имеет место глобальная основа с локальной поддержкой.

### 27.2. Оценка риска

Самая естественная метрика ошибок зависит от того, насколько ожидается, что прогноз будет ошибочным. Это называется *потерей*  $L_1$ . Но многие алгоритмы испытывают трудности с минимизацией этой метрики и хотят вводить больший штраф за большие ошибки. Так что вместо нее в качестве основного критерия оптимизации используются *дифференцируемые потери*  $L_2$ , называемые *среднеквадратичной ошибкой*, СКО (Root-Mean-Squared Error, RMSE). Для сравнения, потеря  $L_\infty$  является ошибкой наихудшего случая, по крайней мере, на тестовом наборе, но ее истинное значение можно оценить только при использовании фиксированного квантиля.

Метрику СКО труднее интерпретировать, чем две другие. Ее можно представить как стандартное отклонение от  $y$  для существующей  $f$ . Получаем  $\text{expStd}$  — % объясненного стандартного отклонения  $= 1 - \frac{\text{СКО}}{\text{stdev}(y_{\text{тестовые данные}})}$ . Представьте, насколько больше  $f$

дает информации по сравнению с простым средним значением меток тестовых данных. По всей видимости, это лучший критерий для сравнения различных  $A$ . Поскольку  $-\text{expStd}$  является монотонной функцией СКО, использование любой из них для сравне-

ния дает один и тот же результат, но первую легко превратить в искривленную оценку для сравнения разных  $A$  на разных наборах данных.

В перекрестной проверке СКО используется напрямую. Хотя у нее нет проблем с минимизацией более интуитивно понятных и прямых потерь  $L_1$ , СКО подходит для использования лучше, т. к. она совместима с различными  $A$  и налагает большие штрафы за большие ошибки. Не во всех приложениях существуют такие предпочтения, но определить идеальную метрику ошибок довольно трудно. В регрессии то, что вам нужно, чего вы хотите и что вы получаете, может различаться, а метрика потерь  $L_2$  лучше умеет давать то, что вам нужно, чем другие. В особых задачах могут использоваться настраиваемые метрики ошибок, учитывающие стоимость ошибок:

```
struct RegressionStats(double expStd, rmse, l1Err, l1InfErr);
RegressionStats evaluateRegressor(
    Vector<pair<double, double> > const& testResult)
{
    IncrementalStatistics yStats, l2Stats, l1Stats;
    for(int i = 0; i < testResult.getSize(); ++i)
    {
        yStats.addValue(testResult[i].first);
        double diff = testResult[i].second - testResult[i].first;
        l1Stats.addValue(abs(diff));
        l2Stats.addValue(diff * diff);
    }
    RegressionStats result;
    result.l1InfErr = l1Stats.maximum;
    result.l1Err = l1Stats.getMean();
    result.rmse = sqrt(l2Stats.getMean());
    result.expStd = 1 - result.rmse/yStats.stdev();
    return result;
}

template<typename LEARNER, typename DATA, typename PARAMS> double
crossValidateReg(PARAMS const& p, DATA const& data, int nFolds = 5)
{
    return evaluateRegressor(crossValidateGeneral<LEARNER,
        typename DATA::Y_TYPE>(p, data, nFolds)).rmse;
}

template<typename LEARNER, typename PARAM, typename DATA>
struct RRiskFuntor
{
    DATA const& data;
    RRiskFuntor(DATA const& theData): data(theData) {}
    double operator() (PARAM const& p) const
    {return crossValidateReg<LEARNER>(p, data);}
};
```

Стратификация в этом случае недоступна, в отличие от классификации, но повторная перекрестная проверка выглядит следующим образом:

```
template<typename LEARNER, typename DATA, typename PARAMS> double
repeatedCVReg(PARAMS const& p, DATA const& data, int nFolds = 5,
    int nRepeats = 5)
```

```

{
    return evaluateRegressor(repeatedCVGeneral<double>(
        LEARNER(data, p), data, nFolds, nRepeats)).rmse;
}
template<typename LEARNER, typename PARAM, typename DATA>
struct RRCVRiskFuncutor
{
    DATA const& data;
    RRCVRiskFuncutor(DATA const& theData): data(theData) {}
    double operator() (PARAM const& p) const
    {return repeatedCVReg<LEARNER>(p, data);}
};

```

Иногда полезно задавать разные доверительные интервалы для отдельных параметров. Методы начальной загрузки позволяют сделать это, хотя некоторые модели, такие как линейная регрессия, имеют на этот случай свою логику и тесты. Но параметры обычно зависят друг от друга — так, например, если один увеличивается, другие должны уменьшаться, чтобы компенсировать это, и т. д. Обычно требуется проверять, не равны ли параметры нулю, а не их фактические значения. И интервалы хороши настолько, насколько хороша модель, потому что, например, если вычислять интервалы смещенной штрафной модели, такой как лассо, на интервалы будет влиять смещение. Доверительный интервал хорош только для того, чтобы сказать, насколько хороша оценка параметра, но он не говорит, насколько хорош сам параметр.

Еще один вопрос заключается в том, чтобы решить, правильно ли моделируются данные с использованием конкретной, обычно параметрической модели, такой как линейная регрессия. Несмотря на некоторые специализированные тесты для конкретных моделей, проверка точности с помощью перекрестной проверки и вариантов является единственным общим методом. Также специализированные методы нельзя применять при любых незначительных изменениях модели. Любая модель будет иметь некоторые ошибки оценки и аппроксимации.

Наконец, учитывая конкретный прогноз для того или иного  $x$ , часто требуется получить для него некоторый доверительный интервал. Бэггинг позволяет проверить прогнозы моделей, обученных на данных с повторной выборкой. Вопрос в том, насколько можно доверять таким интервалам — в лучшем случае они фиксируют дисперсию в модели, но не систематическую ошибку. Так что большой интервал — это тревожный знак, а маленький — не гарантия того, что все хорошо. Это скорее погрешность, чем доверительные интервалы (см. главу 21. *Вычислительная статистика*).

## 27.3. Управление сложностью

Чтобы выполнять полезный анализ регрессии, нужно предположить, что потери  $\leq$  некоторой большой постоянной  $M$ . Это не относится к потерям  $L_2$ , но можно рассматривать потери  $L_2$ , уменьшенные на  $M$ . Если ограничения на потери нет, одно большое значение потери может изменить риск на произвольную величину, поэтому вероятностных гарантий дать нельзя.

Аналогично размерности VC в классификации мерой сложности  $G$  является *псевдоразмерность* (см. [27.10]):

- ♦ для  $n$  точек создать набор классификаторов  $IG$ , заданный выбором констант  $t_i$ . Для любого примера  $ig(x_i) = \text{sign}(g(x_i) - t_i)$ . То есть функции  $g$  дискретизированы относительно того, насколько сильно они колеблются вокруг определенного набора контрольных точек;
- ♦ при наихудшем выборе  $t_i$ :  $\text{PDim}(G) = \text{VC-dim}(IG)$ .

Теорема (см. [27.10]): для  $G$ , состоящего из линейных предикторов,  $\text{PDim}(G) = D + 1$ .

Что касается классификации, ошибка обобщения может быть ограничена функцией ошибки обучения,  $n$ , и сложности  $G$ . Теорема (см. [27.10]): пусть  $G$  имеет псевдоразмерность  $d$ . Тогда с вероятностью  $\geq 1 - p$ :

$$R_f \leq R_{f,n} + M \left( \sqrt{\frac{2d \ln(en/d)}{n}} + \sqrt{\frac{-\ln(p)}{2n}} \right).$$

## 27.4. Линейная регрессия

Нужно получить наиболее подходящую линейную комбинацию функций  $y = wx + b$ , где весовой вектор  $w$  и член смещения  $b$  минимизируют  $\sum (y_i - f(x_i))^2$  для обучающих данных. Для простоты можно сделать  $b$  частью  $w$ , добавив к  $x$  постоянный признак со значением 1. Тогда для  $X = n \times (D + 1)$  матрица значений признаков для всех данных и  $Y$  вектора выходов:  $w = (X^T X)^{-1} X^T Y$ . Предполагая, что  $y_i - f(x_i) \sim$  распределено нормально, существуют доверительные интервалы для  $w$  и  $b$ , но они полезны только в случае, если модель является очень хорошим приближением. Этот случай далее не рассматривается.

Вычисление не выполняется, когда  $D > n$  или  $X$  сингулярно. Нужна дополнительная логика, такая как вычисление псевдоинверсии или в последнем случае удаление избыточных признаков или примеров.

Для обратного вычисления требуется время  $O(n^3)$ , время и пространство  $O(n^2)$ , которое не превышает среднего  $n$ . Более эффективным способом вычисления решения является использование итеративного процесса, такого как регрессия лассо.

Делаются два основных предположения:

- ♦ модель линейная — обычно это приводит к большой ошибке аппроксимации. Но когда у вас мало данных или слишком много переменных, предположение о линейной модели может быть единственным способом контролировать ошибку оценки;
- ♦ ошибки являются независимыми и случайными и распределены нормально. Это разумно для ошибок измерения, т. е. предполагаемая модель верна, но данные подвержены некоторой случайности из-за неправильного измерения. Однако систематическая разница в ошибках встречается часто, потому что, например, их дисперсия может быть больше в больших значениях, чем в малых.

Линейная модель позволяет проводить более сложный анализ — например, оценивать влияние отдельных переменных в решении. Они имеют значение только в том случае, если линейная модель имеет низкую ошибку аппроксимации. Даже добавление дополнительных переменных может привести к получению другой модели, в которой дове-



рительные интервалы для переменных в меньшей модели будут уже другими. Это *парадокс Симпсона* (см. [27.14]). Например, в исследовании предвзятости при поступлении в аспирантуру пол был определен как значимый предиктор в пользу юношей, но оказалось, что девушки на разных факультетах на самом деле лучше проходили отбор, и эффект смещения был связан с тем фактом, что они обращались на более приемлемые для себя факультаты (здесь предиктор пола даже изменил знак). Таким образом, структура модели кажется ограниченной этой конкретной моделью, а не абсолютной истиной.

Но исследование было наблюдательным. Для независимых случайных данных парадокс Симпсона менее вероятен из-за отсутствия глобального смещения, но возможен, если нет аддитивности факторов, как предполагается линейной моделью.

## 27.5. Регрессия лассо

Хорошее решение проблем классической регрессии — применять штраф  $L_1$  к весам, получая то, что дает формула в качестве цели минимизации для малых постоянных  $l$ . Это называется *лассо*. Что касается LSVM, штраф  $L_1$  ведет к получению разреженных решений.

Задача выпуклая и имеет вид выпуклой функции + дифференцируемой функции. Таким образом, спуск по координатам (см. главу 24. Численная оптимизация) является глобально сходящимся (см. [27.6]). Кроме того, одномерные задачи имеют аналитические решения, а штрафы для всех остальных компонентов остаются постоянными, что позволяет проводить пошаговую оценку.

Для простоты добавьте коэффициент  $\frac{1}{2}$  и минимизируйте  $l|w| + \frac{1}{2} \sum (y_i - f(x_i))^2$ . При работе с весом  $j$  агрегируйте другие компоненты  $f(x_i)$  как  $s_i = y_i - (b + \sum_{k \neq j} w_k x_{ik})$ . Этот метод оптимален, когда  $0 \in$  множеству субградиента  $0 = lS(w) + \sum (s_i - w_j x_{ij})(-x_{ij})$ , где

$S$  — знаковая функция. Выполняя решение, получаем  $w_j = \frac{a + lS(w_j)}{c}$  для  $a = \sum s_i x_{ij}$

и  $c = \sum x_{ij}^2$ . Поскольку  $c \geq 0$ ,  $a \geq -l$ , если  $w_j \geq 0$ , и  $\leq l$ , если  $w_j \leq 0$ . Итак, учитывая интервалы  $a$ ,  $w_j < 0$ , если  $a < -l$ ,  $> 0$ , если  $a > l$ , и 0 в противном случае. Таким образом:

$$w_j = \begin{cases} \frac{a-l}{c}, & \text{если } a < -l \\ \frac{a+l}{c}, & \text{если } a > l \\ 0 & \end{cases}$$

Формула является численно устойчивой, потому что малые  $c$  ведут к  $w_j = 0$ .

$b = \frac{1}{n} \sum s_i$ , поскольку  $\frac{1}{2} \sum (s_i - b)^2$  является наименьшим. Значение  $l$  выбирается с помощью перекрестной проверки из того же диапазона, что и у линейного SVM (рис. 27.1).

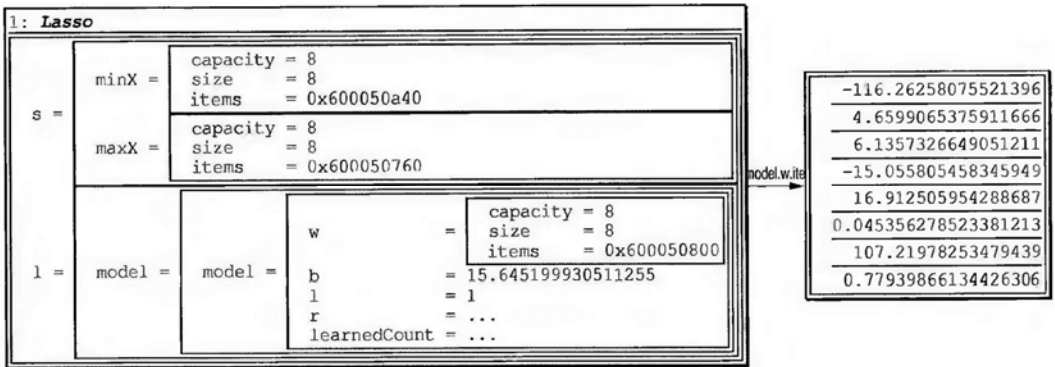


Рис. 27.1. Структура памяти результатов регрессии лассо для данных об энергии

```
class L1LinearReg
{
    Vector<double> w;
    double b, l, r;
    int learnedCount; // это значение и r используются только для обучения
                        // в реальном времени с помощью SGD
    double f(NUMERIC_X const& x) const { return dotProduct(w, x) + b; }
    template<typename DATA> void coordinateDescent(DATA const& data,
        int maxIterations, double eps)
    {
        assert(data.getSize() > 0);
        int D = getD(data);
        Vector<double> sums(data.getSize());
        for(int i = 0; i < data.getSize(); ++i) sums[i] = data.getY(i);
        bool done = false;
        while(!done && maxIterations--)
        {
            done = true;
            for(int j = -1; j < D; ++j)
            {
                double oldVar = j == -1 ? b : w[j];
                // удаление текущей переменной из суммы
                for(int i = 0; i < data.getSize(); ++i)
                    sums[i] += j == -1 ? b : w[j] * data.getX(i, j);
                // решение для оптимальной текущей переменной
                if(j == -1)
                {
                    // обновление смещений
                    IncrementalStatistics s;
                    for(int i = 0; i < data.getSize(); ++i)
                        s.addValue(sums[i]);
                    b = s.getMean();
                }
                else
                {
                    // обновление весов
                    double a = 0, c = 0;
```

```

        for(int i = 0; i < data.getSize(); ++i)
        {
            double xij = data.getX(i, j);
            a += sums[i] * xij;
            c += 1 * xij * xij;
        }
        if(a < -1) w[j] = (a - 1)/c;
        else if(a > 1) w[j] = (a + 1)/c;
        else w[j] = 0;
    }
    // возврат текущей переменной
    for(int i = 0; i < data.getSize(); ++i)
        sums[i] -= j == -1 ? b : w[j] * data.getX(i, j);
    if(abs((j == -1 ? b : w[j]) - oldVar) > eps) done = false;
}

}

public:
    template<typename DATA> L1LinearReg(DATA const& data, double theL,
        int nCoord = 1000): l(theL/2), b(0), w(getD(data)), learnedCount(-1)
    {coordinateDescent(data, nCoord, pow(10, -6));}
    typedef pair<int, pair<double, double> > PARAM; // D/l/r
    L1LinearReg(PARAM const& p): l(p.second.first/2), r(p.second.second),
        b(0), w(p.first), learnedCount(0) {}
    void learn(NUMERIC_X const& x, double y)
    {
        assert(learnedCount != -1); // нельзя смешивать пакетный и офлайн-методы
        double rate = r * RMRate(learnedCount++), err = y - f(x);
        // 1/n*|w| + (y - wx + b)2
        // dw = 1/n*sign(w) - x(y - (wx + b));
        // db = - (y - (wx + b))
        for(int i = 0; i < w.getSize(); ++i) w[i] +=
            rate * (x[i] * err - (w[i] > 0 ? 1 : -1) * 1/learnedCount);
        b += rate * err;
    }
    double predict(NUMERIC_X const& x) const{return f(x);}
    template<typename MODEL, typename DATA>
    static double findL(DATA const& data)
    {
        int lLow = -15, lHigh = 5;
        Vector<double> regs;
        for(double j = lHigh; j > lLow; j -= 2) regs.append(pow(2, j));
        return valMinFunc(regs.getArray(), regs.getSize(),
            RRiskFunctor<MODEL, double, DATA>(data));
    }
};

struct NoParamsL1LinearReg
{
    L1LinearReg model;
    template<typename DATA> NoParamsL1LinearReg(DATA const& data):
        model(data, L1LinearReg::findL<L1LinearReg>(data)) {}
    double predict(NUMERIC_X const& x) const{return model.predict(x);}
};

typedef ScaledLearner<NoParamsLearner<NoParamsL1LinearReg, double>, double> SLasso;

```

Каждая итерация занимает время  $O(n)$ . См. главу 26. *Машинное обучение: классификация*, чтобы узнать, что происходит, когда данные хранятся на диске.

Чтобы выполнить обучение в реальном времени, задействуйте SGD. При заданной скорости  $r$  дифференцирование цели дает уравнения обновления  $w_i += r \left( x\varepsilon - S(w_i) \frac{1}{n} \right)$  и  $b += r\varepsilon$  для  $\varepsilon = y - f(x)$ . Что касается онлайн-LSVM, используйте количество рассмотренных примеров, равное  $n$ . В отличие от LSVM, необходимо настроить начальное  $r$  для гонки, потому что шаги SGD могут стать слишком большими. Кроме того, SGD сходится очень медленно, поэтому вы не сможете получить разреженность, поскольку многие возможные 0-компоненты  $w$  могут быть недостаточно близки к 0. Для гонок начальное  $r$  использует тот же диапазон, что и для  $l$ , который такой же, как и для классификации:

```
typedef pair<int, pair<double, double> > PARAM; // D/1/r
LlLinearReg(PARAM const& p): l(p.second.first/2), r(p.second.second),
    b(0), w(p.first), learnedCount(0) {}
void learn(NUMERIC_X const& x, double y)
{
    assert(learnedCount != -1); // нельзя смешивать пакетный и офлайн-методы
    double rate = r * RMRate(learnedCount++), err = y - f(x);
    for(int i = 0; i < w.getSize(); ++i) w[i] +=
        rate * (x[i] * err - (w[i] > 0 ? 1 : -1) * 1/learnedCount);
    b += rate * err;
}
class SRaceLasso
{
    ScalerMQ s;
    RaceLearner<LlLinearReg, LlLinearReg::PARAM> model;
    static Vector<LlLinearReg::PARAM> makeParams(int D)
    {
        Vector<LlLinearReg::PARAM> result;
        int lLow = -15, lHigh = 5, rLow = -15, rHigh = 5;
        for(int j = lHigh; j > lLow; j -= 2)
        {
            double l = pow(2, j);
            for(int i = rHigh; i > rLow; i -= 2) result.append(
                LlLinearReg::PARAM(D, pair<double, double>(l, pow(2, i))));
        }
        return result;
    }
public:
    template<typename DATA> SRaceLasso(DATA const& data):
        model(makeParams(getD(data))), s(getD(data))
    {
        for(int j = 0; j < 1000000; ++j)
        {
            int i = GlobalRNG().mod(data.getSize());
            learn(data.getX(i), data.getY(i));
        }
    }
}
```

```

SRaceLasso(int D): model(makeParams(D)), s(D) {}
void learn(NUMERIC_X const& x, double y)
{
    s.addSample(x);
    model.learn(s.scale(x), y);
}
double predict(NUMERIC_X const& x) const
{ return model.predict(s.scale(x)); }
};

```

Лассо — это применяемый по умолчанию подход, заменяющий линейную регрессию «черным ящиком» из-за благоприятных свойств штрафа  $L_1$ . Использование  $l = 0$  и отсутствие масштабирования сводятся к линейной регрессии, и единственное возможное преимущество этого метода состоит в том, чтобы избежать масштабирования и получить более простую для объяснения модель. Но можно изменить масштаб  $w$  и  $b$  (которые соответственно поглощают мультипликативный и аддитивный члены масштабирования) лассо, чтобы получить тот же результат.

## 27.6. Регрессия ближайших соседей

По сравнению с классификацией здесь мы используем среднее значение вместо большинства. Алгоритм соответствует потере  $L_2$  при тех же условиях, что и для классификации (см. [27.4]). Та же теоретическая логика применима и к выбору  $k$ :

```

template<typename X = NUMERIC_X, typename INDEX = VpTree<X, double, typename
    EuclideanDistance<X>::Distance> > class KNNReg
{
    mutable INDEX instances;
    int k;
public:
    template<typename DATA> KNNReg(DATA const& data, int theK = -1): k(theK)
    {
        assert(data.getSize() > 0);
        if(k == -1) k = 2 * int(log(data.getSize())/2) + 1;
        for(int i = 0; i < data.getSize(); ++i)
            learn(data.getY(i), data.getX(i));
    }
    void learn(double label, X const& x){instances.insert(x, label);}
    double predict(X const& x) const
    {
        Vector<typename INDEX::NodeType*> neighbors = instances.kNN(x, k);
        IncrementalStatistics s;
        for(int i = 0; i < neighbors.getSize(); ++i)
            s.addValue(neighbors[i]->value);
        return s.getMean();
    }
};
typedef ScaledLearner<NoParamsLearner<KNNReg<>, double>, double> SKNNReg;

```

Что касается классификации, то поиск может занять  $\approx O(n)$  времени для больших  $D$ .

## 27.7. Дерево регрессии

По сравнению с деревом решений здесь мы предсказываем реальное значение  $y$ . Это предполагает, что кусочно-постоянная функция является хорошим приближением к реальной. Естественная потеря вычисляется как *сумма квадратов ошибок* (Sum of Squared Errors, SSE), минимизатором которой является среднее значение значений  $y$ . Она обновляется постепенно, что позволяет эффективно рассчитывать точки разделения.

Что касается классификации, каждое поддерево должно быть лучше, чем его корень, чтобы его не обрезали. Это проверяется путем сравнения квадратов ошибок с использованием критерия знака с  $z$ -оценкой по умолчанию, равной 0,25, которая кажется наилучшей по результатам экспериментов:

```
struct RegressionTree
{
    struct Node
    {
        union
        {
            double split; // для внутренних узлов
            double label; // для листовых узлов
        };
        int feature; // для внутренних узлов
        Node *left, *right;
        bool isLeaf() {return !left;}
        Node(int theFeature, double theSplit): feature(theFeature),
            split(theSplit), left(0), right(0) {}
    } * root;
    Freelist<Node> f;
    double SSE(double sum, double sum2, int n) const
    {return sum2 - sum * sum/n;}
    template<typename DATA> struct Comparator
    {
        int feature;
        DATA const& data;
        double v(int i) const {return data.data.getX(i, feature);}
        bool operator()(int lhs, int rhs) const {return v(lhs) < v(rhs);}
        bool isEqual(int lhs, int rhs) const {return v(lhs) == v(rhs);}
    };
    void rDelete(Node* node)
    {
        if(node)
        {
            rDelete(node->left);
            f.remove(node->left);
            rDelete(node->right);
            f.remove(node->right);
        }
    }
}
```

```

double classifyHelper(NUMERIC_X const& x, Node* current) const
{
    while(!current->isLeaf()) current = x[current->feature] <
        current->split ? current->left : current->right;
    return current->label;
}

template<typename DATA> Node* rHelper(DATA& data, int left, int right,
    double pruneZ, int depth, bool rfMode)
{
    int D = data.getX(left).getSize(), bestFeature = -1,
        n = right - left + 1;
    double bestSplit, bestScore, sumY = 0, sumY2 = 0;
    Comparator<DATA> co = {-1, data};
    for(int j = left; j <= right; ++j)
    {
        double y = data.getY(j);
        sumY += y;
        sumY2 += y * y;
    }
    double ave = sumY/n, sse = max(0.0, SSE(sumY, sumY2, n));
    Bitset<> allowedFeatures;
    if(rfMode)
    { // признаки выборки для случайного леса
        allowedFeatures = Bitset<>(D);
        allowedFeatures.setAll(0);
        Vector<int> p = GlobalRNG().sortedSample(sqrt(D), D);
        for(int j = 0; j < p.getSize(); ++j) allowedFeatures.set(p[j], 1);
    }
    if(sse > 0) for(int i = 0; i < D; ++i) // поиск лучшего признака и разделение
        if(allowedFeatures.getSize() == 0 || allowedFeatures[i])
        {
            co.feature = i;
            quickSort(data.permutation.getArray(), left, right, co);
            double sumYLeft = 0, sumYRight = sumY, sumY2Left = 0,
                sumY2Right = sumY2;
            int nRight = n, nLeft = 0;
            for(int j = left; j < right; ++j)
            { // инкрементный перебор счетчиков
                int y = data.getY(j);
                ++nLeft;
                sumYLeft += y;
                sumY2Left += y * y;
                --nRight;
                sumYRight -= y;
                sumY2Right -= y * y;
            }
            double fLeft = data.getX(j, i), score =
                SSE(sumYLeft, sumY2Left, nLeft) +
                SSE(sumYRight, sumY2Right, nRight),
                fRight = data.getX(j + 1, i);
            if(fLeft != fRight && // равные значения не разделяются
                (bestFeature == -1 || score < bestScore))

```

```

        {
            bestScore = score;
            bestSplit = (fLeft + fRight)/2;
            bestFeature = i;
        }
    }

    if(n < 3 || depth <= 1 || sse <= 0 || bestFeature == -1)
        return new(f.allocate())Node(-1, ave);
    // примеры делятся на правые и левые
    int i = left - 1;
    for(int j = left; j <= right; ++j)
        if(data.getX(j, bestFeature) < bestSplit)
            swap(data.permutation[j], data.permutation[++i]);
    if(i < left || i > right) return new(f.allocate())Node(-1, ave);
    Node* node = new(f.allocate())Node(bestFeature, bestSplit);
    // рекурсивное вычисление потомков
    node->left = rHelper(data, left, i, pruneZ, depth - 1, rfMode);
    node->right = rHelper(data, i + 1, right, pruneZ, depth - 1, rfMode);
    // попытка выполнить обрезку
    double nodeWins = 0, treeWins = 0;
    for(int j = left; j <= right; ++j)
    {
        double y = data.getY(j), eNode = ave - y, eTree =
            classifyHelper(data.getX(j), node) - y;
        if(eNode * eNode == eTree * eTree)
        {
            nodeWins += 0.5;
            treeWins += 0.5;
        }
        else if(eNode * eNode < eTree * eTree) ++nodeWins;
        else ++treeWins;
    }
    if(!rfMode && signTestAreEqual(nodeWins, treeWins, pruneZ))
    {
        rDelete(node);
        node->left = node->right = 0;
        node->label = ave;
        node->feature = -1;
    }
    return node;
}

Node* constructFrom(Node* node)
{
    Node* tree = 0;
    if(node)
    {
        tree = new(f.allocate())Node(*node);
        tree->left = constructFrom(node->left);
        tree->right = constructFrom(node->right);
    }
}

```



```

        return tree;
    }
public:
    template<typename DATA> RegressionTree(DATA const& data, double pruneZ =
        0.25, int maxDepth = 50, bool rfMode = false): root(0)
    {
        assert(data.getSize() > 0);
        int left = 0, right = data.getSize() - 1;
        PermutedData<DATA> pData(data);
        for(int i = 0; i < data.getSize(); ++i) pData.addIndex(i);
        root = rHelper(pData, left, right, pruneZ, maxDepth, rfMode);
    }
    RegressionTree(RegressionTree const& other)
    {root = constructFrom(other.root);}
    RegressionTree& operator=(RegressionTree const& rhs)
    {return genericAssign(*this, rhs);}
    double predict(NUMERIC_X const& x) const
    {return root ? classifyHelper(x, root) : 0;}
};

```

Время выполнения = дерево решений.

## 27.8. Регрессия случайного леса

Единственное отличие от классификации здесь заключается в том, что для объединения используется среднее значение вместо большинства:

```

class RandomForestReg
{
    Vector<RegressionTree> forest;
public:
    template<typename DATA> RandomForestReg(DATA const& data,
        int nTrees = 300){addTrees(data, nTrees);}
    template<typename DATA> void addTrees(DATA const& data, int nTrees)
    {
        assert(data.getSize() > 1);
        for(int i = 0, D = getD(data); i < nTrees; ++i)
        {
            PermutedData<DATA> resample(data);
            for(int j = 0; j < data.getSize(); ++j)
                resample.addIndex(GlobalRNG().mod(data.getSize()));
            forest.append(RegressionTree(resample, 0, 50, true));
        }
    }
    double predict(NUMERIC_X const& x) const
    {
        IncrementalStatistics s;
        for(int i = 0; i < forest.getSize(); ++i)
            s.addValue(forest[i].predict(x));
        return s.getMean();
    }
};

```

## 27.9. Нейронная сеть

Структура здесь аналогична классификации, но с некоторыми изменениями:

- ◆ перекрестно проверьте количество скрытых нейронов — учетверение от 1 до 64 кажется разумным;
- ◆ перекрестно проверьте начальную скорость обучения для SGD, чтобы спуск не расходился. Используйте тот же диапазон, что для лассо в реальном времени;
- ◆ не используйте функцию активации для блока вывода.

```
class HiddenLayerNNReg
{
    Vector<NeuralNetwork> nns;
public:
    template<typename DATA> HiddenLayerNNReg(DATA const& data,
        Vector<double>const& p, int nGoal = 100000, int nNns = 5):
        nns(nNns, NeuralNetwork(getD(data), true, p[0]))
    { // структура
        int nHidden = p[1], D = getD(data),
            nRepeats = ceiling(nGoal, data.getSize());
        double a = sqrt(3.0/D);
        for(int l = 0; l < nns.getSize(); ++l)
        {
            NeuralNetwork& nn = nns[l];
            nn.addLayer(nHidden);
            for(int j = 0; j < nHidden; ++j)
                for(int k = -1; k < D; ++k)
                    nn.addConnection(0, j, k, k == -1 ? 0 :
                        GlobalRNG().uniform(-a, a));
            nn.addLayer(1);
            for(int k = -1; k < nHidden; ++k)
                nn.addConnection(1, 0, k, 0);
        }
        // обучение
        for(int j = 0; j < nRepeats; ++j)
            for(int i = 0; i < data.getSize(); ++i)
                learn(data.getX(i), data.getY(i));
    }
    void learn(NUMERIC_X const& x, double label)
    {
        for(int l = 0; l < nns.getSize(); ++l)
            nns[l].learn(x, Vector<double>(1, label));
    }
    double evaluate(NUMERIC_X const& x) const
    {
        double result = 0;
        for(int l = 0; l < nns.getSize(); ++l)
            result += nns[l].evaluate(x)[0];
        return result/nns.getSize();
    }
    int predict(NUMERIC_X const& x) const{return evaluate(x);}
};
```

```
struct NoParamsNNReg
{
    HiddenLayerNNReg model;
    template<typename DATA> static Vector<double> findParams(DATA const&
        data, int rLow = -15, int rHigh = 5, int hLow = 0, int hHigh = 6)
    {
        Vector<Vector<double> > sets(2);
        for(int i = rLow; i <= rHigh; i += 2) sets[0].append(pow(2, i));
        for(int i = hLow; i <= hHigh; i += 2) sets[1].append(pow(2, i));
        return gridMinimize(sets,
            RRiskFunctor<HiddenLayerNNReg, Vector<double>, DATA>(data));
    }
    template<typename DATA> NoParamsNNReg(DATA const& data):
        model(data, findParams(data)) {}
    double predict(NUMERIC_X const& x) const{return model.predict(x);}
};
typedef ScaledLearner<NoParamsLearner<NoParamsNNReg, double>, double, EMPTY,
    ScalerMQ> SNNReg;
```

Как и в случае с классификацией, время выполнения каждого примера составляет  $O(W)$ . Если используется глобальная оптимизация и правильно выбрано количество скрытых блоков, нейронная сеть совместима с потерями  $L_2$  (см. [27.4]).

## 27.10. Выбор признаков

Что касается классификации, поиск в оболочке со случайным лесом полезен для выбора признаков:

```
template<typename SUBSET_LEARNER = RandomForestReg> struct SmartFSLearnerReg
{
    typedef FeatureSubsetLearner<SUBSET_LEARNER> MODEL;
    MODEL model;
public:
    template<typename DATA> SmartFSLearnerReg(DATA const& data,
        int subsampleLimit = 20): model(data, selectFeaturesSmart(
            RRiskFunctor<MODEL, Bitset<>, DATA>(data), getD(data),
            subsampleLimit)) {}
    double predict(NUMERIC_X const& x) const{return model.predict(x);}
};
```

## 27.11. Сравнение производительности

При использовании средней изогнутой метрики expStd случайный лес кажется лучшим алгоритмом  $A$  (рис. 27.2).

mqLasso	mqOnlineLasso	RegTree	RFRegTree	B_RF_Lasso	sKNNReg	mqNNcv
0.62	0.41	0.5	0.81	0.9	0.73	0.74

Рис. 27.2. Сравнение производительности

Объединение этого метода с лассо дает еще лучшие результаты за счет дополнительной перекрестной проверки:

```
class SimpleBestCombinerReg
{
    BestCombiner<double> c;
public:
    template<typename DATA> SimpleBestCombinerReg(DATA const& data)
    {
        c.addNoParamsClassifier<RandomForestReg>(data, RRiskFunctor<
            NoParamsLearner<RandomForestReg, double>, EMPTY, DATA>(data));
        c.addNoParamsClassifier<SLasso>(data, RRiskFunctor<
            NoParamsLearner<SLasso, double>, EMPTY, DATA>(data));
    }
    double predict(NUMERIC_X const& x) const{return c.predict(x);}
};
```

В настоящее время в литературе не приведено всестороннего сравнения современных регрессионных методов, основанных на многих наборах данных. В работе [27.9] приводится сравнение некоторых алгоритмов на некоторых наборах данных (рис. 27.3).

SVR	LinReg	RegTree	NN	MARS	AdditiveModel	ProjectionPursuit	RF	Boosting
2.83	6.82	6.83	2.75	5.5	4.33	3.08	2.83	4.67

Рис. 27.3. Преобразование в ранги Фридмана и средние ошибки  $L_2$  от [27.9] дают следующие средние ранги. SVR, бустинг и MARS обсуждаются в комментариях. Также про бустинг, проекционное преследование (ProjectionPursuit) и аддитивные модели (AdditiveModel) можно почитать в работе [27.5]

По результатам этого исследования случайный лес и нейронные сети, по-видимому, являются предпочтительными алгоритмами, и оба намного более эффективны, чем SVR. В сочетании с моими экспериментами случайный лес кажется лучшим вариантом  $A$ . В имитационном исследовании LOESS (обсуждается в комментариях) и MARS плохо работали на данных большой размерности (см. [27.3]).

Но простые модели часто бывают достаточно хороши — например, линейная регрессия хорошо подходит для моделирования цен на жилье из-за естественной аддитивности таких признаков, как качество ближайшей школы и удобство транспорта. Также оценочные коэффициенты логически не подвержены парадоксу Симпсона из-за естественной аддитивности.

## 27.12. Примечания по реализации

Использование СКО как процента стандартного отклонения — это оригинальная идея, и она имеет смысл в качестве масштабной меры.

Реализация лассо является наиболее сложной из-за необходимости проработки математики. Сочетание координатного спуска и обычного SGD кажется хорошим способом его оптимизации.

Реализация дерева регрессии точно отражает реализацию дерева классификации, потому что использование знакового теста для обрезки работает одинаково в обоих случаях.

Другие алгоритмы также в основном являются расширением алгоритмов классификации, хотя и с потенциально другими параметрами.

## 27.13. Комментарии

В задаче регрессии большое значение имеют затраты, потому что завышенная оценка может выйти вам меньше или дороже, чем заниженная, но смоделировать это таким образом, чтобы получить эффективный  $A$ , сложно.

Еще одна интересная мера сложности — это *масштабное измерение*, которое реализуется более сложно и не дает получения более полезной верхней границы (см. [27.10]). Зато это позволяет получить нижнюю границу, аналогичную размерности  $VC$  (см. [27.1]). Эту оценку также следует воспринимать с долей скептицизма, потому что наложение дополнительных условий на  $G$ , таких как ограничение размеров чисел, обычно снижает любую меру сложности.

Вопросы разреженности в настоящий момент активно исследуются (см. [27.6]). Для регрессии лассо существуют и другие приемы — например, применение более простой формулы спуска по координатам при использовании шкалы среднего отклонения и начала с предыдущего решения  $l$  при выборе  $l$ . Но делать это с перекрестной проверкой неуклюже.

Линейная регрессия во многих отношениях ненадежна, но и у надежных решений есть свои проблемы. В работе [27.15] приведен хороший обзор. Это также относится к разреженной регрессии. Популярный вариант, называемый *наименьшей медианой квадратов* (Least Median of Squares, LMS), имеет высокую степень разбивки, но экспоненциально неэффективен по  $D$  для вычислений. Таким образом, кажется, что хороших вариантов надежной регрессии на все случаи жизни нет (см. [27.7]). Но в простых вычислительных исследованиях — например, при изучении асимптотики алгоритмов, которые работают с несколькими переменными (обычно с одной или двумя) и могут генерировать данные по запросу, LMS кажется хорошим методом.

В *гребневой регрессии* используется штраф  $L_2$ , который также устраняет проблемы линейной регрессии и имеет аналитическое решение, но не приводит к разреженности. Кроме того, аналитическое решение занимает время  $O(\min(n, D)^3)$ , которое не масштабируется. Возможно, самая большая проблема с лассо заключается в том, что при наличии коррелированных переменных алгоритм может выбрать любую из них. Интересным предложением для решения этой проблемы является *эластичная сеть*, т. е. использование комбинации штрафов  $L_1$  и  $L_2$ . У этой сети тоже есть аналитическое решение, а также дополнительный параметр комбинации весов, о котором пока известно мало.

*Регрессия опорного вектора ядра* (Kernel Support Vector Regression, SVR) ведет к попаданию вместо линии в  $\varepsilon$ -трубу для некоторого числа  $\varepsilon$  в расширенном пространстве признаков (см. [27.10]). Это дает разреженность в количестве опорных векторов, потому что примеры внутри трубы не могут быть опорными векторами. Но SMO-подобное решение более сложное, чем классификационное (см. [27.8]), и для перекрестной проверки также необходимо выбирать  $\varepsilon$ . Тем не менее, по существу, используется один и тот же алгоритм, т. е. итеративно решается последовательность 2D-задач. Разница в основном состоит в расчете градиентов и решении 2D-задач. Для параметров при прочих равных нужны маленькие  $C$ , большие  $\gamma$  и большие  $\varepsilon$ .

*Регрессия гребня ядра* является прямым продолжением гребневой регрессии (см. [27.10]). Она также имеет аналитическое решение, но время выполнения  $O(n^3)$ . Кроме того, в отличие от SVR, решения не являются редкими по количеству опорных векторов.

*Ядерная регрессия Надарая — Ватсона* — это более старый метод, который, несмотря на тривиальное обобщение на  $D > 1$ , ведет себя экспоненциально плохо с увеличением  $D$  (см. [27.3]). По сути, пропускная способность локального ядра фиксирована, но, например, расстояния между ближайшими соседями адаптивны.

Для дерева регрессии было предложено много критериев обрезки, хотя и меньше, чем для дерева решений. Можно также применять сокращение сложности затрат.

В качестве альтернативы можно рассмотреть потери  $L_1$  для дерева, минимизатором которого является медиана. Медиана может быть более надежной, чем среднее, но подходит не для всех случаев, а ее инкрементальный расчет немного сложен. Один из способов сделать это эффективно — использовать сбалансированное дерево поиска с увеличением размера поддеревья, где ключ представляет собой комбинацию абсолютной ошибки и номера примера (чтобы разрешить не уникальные элементы в дереве).

Интересной альтернативой является модельное дерево с линейной регрессией на листьях, использующее сглаживание (см. [27.16]). Но, несмотря на заявления о превосходстве над обычным деревом регрессии, у него, похоже, нет варианта использования:

- ◆ это дерево гораздо менее интерпретируемо;
- ◆ случайный лес лучше выполняет сглаживание;
- ◆ усреднение на листе с меньшей вероятностью переобучается на небольшом количестве данных;
- ◆ существуют некоторые недостатки конструкции — такие как необходимость параметра степени сглаживания и подгонка линейной модели к листьям с несколькими примерами.

Для случайного леса в исходной статье рекомендуется использовать  $D/3$  признаков. Но, по-видимому, нет никаких причин выбирать это значение вместо  $\sqrt{D}$ .

В-сплайны (см. комментарии в *главе 23. Численные алгоритмы: работа с функциями*) с приближением методом наименьших квадратов можно использовать для регрессии в одномерном случае. Но даже здесь возникают трудности с зашумленными данными. Штрафные методы в некоторой степени помогают решить это, но для  $D > 1$  проблема все еще есть (см. [27.3]). Расширение сплайнов регрессии до  $D > 1$  требует тензорных произведений сплайнов (см. [27.5]), что по существу сводится к наложению сетки на диапазон данных и подгонке сплайн-произведений к каждой ячейке. Несмотря на универсальную согласованность (см. [27.4]), этот метод не масштабируется.

Разумным решением являются *сплайны многомерной адаптивной регрессии* (Multivariate Adaptive Regression Splines, MARS) (см. [27.13, 27.5]). Линейные сплайны могут быть эквивалентно представлены суммой шарнирных функций, подобных SVM (нелинейное обобщенное сходство), а идея жадного подхода MARS состоит в добавлении шарнирной функции за раз, так что добавление представляет собой шарнир или произведение шарнира на функцию, которая уже является частью суммы. Несмотря на использование вычислительных трюков (см. [27.5]), время выполнения оказывается медленнее, а производительность хуже, чем у случайного леса.

Методы локальной регрессии обобщают  $k$ -NN несколькими способами. Типичным решением является использование ядра для моделирования локальной области нескольких ближайших соседей вместо среднего. Такие методы, как *LOESS* (см. [27.12, 27.5]), позволяют сделать это, но *SVR* сохраняет только опорные векторы и поэтому кажется намного лучше. Кроме того, *LOESS* требует больше данных, чем другие методы, для хорошей оценки даже для  $D = 1$  (см. [27.3]).

Многие методы, такие как вейвлеты (см. [27.5]), можно использовать в одномерном случае, но до больших  $D$  они не масштабируются. У них есть варианты использования в специализированных областях, особенно для создания визуализаций.

Практически любой локализованный базисный набор страдает от проклятия размерности. Все хорошо работающие методы делают некоторые допущения, когда эффекты данных распространяются далеко в пространстве признаков, — например, разделение дерева разделяет все пространство признаков на две части, а не только области, для которых есть данные. Эта зависимость между локальными регионами требует предположений, которых нельзя избежать.

Другой специальной моделью можно назвать *обобщенную линейную модель* (Generalized Linear Model, GLM) (см. [27.5]). Логистическая регрессия является ее частным случаем. Идея состоит в том, чтобы использовать функцию связи, такую как логит-преобразование, потому что результат потом будет легче оценить. Подгонять такие модели можно с помощью ряда методов, в том числе SGD. Было разработано множество специальных алгоритмов, но простые модификации, такие как штраф за лассо, делают многие из них неработоспособными. Например, рассмотрим *тест прогонов* — для хорошего соответствия примерно одинаковое количество примеров появится ниже и выше гиперплоскости регрессии. Но с лассо это уже не так. Фундаментальный недостаток такой более *общей аддитивной модели* (General Additive Model, GAM) заключается в том, что отдельные переменные по-прежнему обрабатываются независимо, даже если для них используются некоторые преобразования ядерной регрессии. Так, для цифровых преобразований данных ничего не изменится.

Как правило, *полупараметрические модели регрессии*, такие как GAM, делают ограниченные параметрические предположения. Например, логика предметной области может предполагать, что значение отклика является аддитивным по отношению к значениям компонентов, но имеет убывающую отдачу. Все еще приходится делать некоторые субъективные выборы относительно оставшейся параметрической формы, и не хочется пробовать слишком много форм или выбирать неправильную.

Интересным применением регрессии является повышение эффективности очень дорогих симуляций. Например, можно разработать более совершенную систему моделирования автокатастроф, в которой исследуется столкновение автомобиля с куклами-пассажирами со стеной. Чтобы сэкономить автомобили, создайте регрессионную модель (называемую *метамоделью*, или *суррогатной моделью*) и оптимизируйте ее, чтобы узнать, какую конфигурацию параметров проектирования следует пробовать следующей. По сути, все, что можно сделать с реальной моделью, можно выполнить и с суррогатной, причем последняя обновляется после каждой оценки реальной модели. Случайный лес является хорошей суррогатной моделью, но может учитывать *SVR*, если для целей анализа требуется дифференцируемость.

Использование метамодели противоречит идее прямого решения проблемы, но, например, прямое применение метода оптимизации приводит к гораздо большему количеству реальных оценок модели. Обычные методы оптимизации не предназначены для очень дорогих симуляций и стремятся сделать среду выполнения  $O$ -масштабируемой. Процессы Кригинга/Гаусса (погуглите) часто используются в качестве метамodelей в опубликованной литературе, но случайный лес и SVR намного быстрее и, как известно, хорошо работают для регрессии.

Иногда даже имеют место метаметамодели. Например, первая реальная модель — это, возможно, автомобильный двигатель, метамодель — это его высокоточная вычислительная спецификация, а метаметамодель — это случайный лес, основанный на некоторых ключевых переменных. Также полезно различать модели регрессии, которые предназначены для потенциально зашумленных данных, и модели интерполяции, предполагающие отсутствие шума.

Еще одним типичным применением является регрессия логарифмических шансов, которая по сути представляет собой достоверную классификацию. Хотя отсутствие прямой классификации приводит к более высокой ошибке оценки, а логарифмические шансы могут быть оценены не очень хорошо, это полезно в некоторых приложениях, особенно когда ошибка теста СКО в логарифмических шансах достаточно мала. Классы 0/1 нужно кодировать как 0,001/0,999 (особенности вычислений).

## 27.14. Советы по дополнительной подготовке

- ◆ В качестве оценочной метрики для окончательного измерения производительности вместо СКО поэкспериментируйте с усеченным средним значением |ошибки|. То же самое справедливо для 90-процентного квантиля |ошибки|. Дают ли они разные результаты?
- ◆ Внедрите эластичную чистую регрессию и поэкспериментируйте с ней. Лучше ли этот алгоритм работает для выбора коррелированных признаков по сравнению с лассо? Расширение — примените к классификации с логистической трансформацией.
- ◆ Внедрите метод SVR и сравните его производительность и эффективность со случайным лесом.
- ◆ Исследуйте использование численного бустинга с деревом регрессии для обучения с асимметричными функциями потерь, такими как `linex`.
- ◆ Поэкспериментируйте с линейной моделью, а затем со случайным лесом в качестве стратегии бустинга.
- ◆ Исследование и внедрение LMS. Для больших  $D$  попробуйте использовать глобальную оптимизацию или жадный алгоритм, если найдете хороший вариант. Расширьте модель, чтобы использовать разреженность, как для обычной регрессии. Используйте результат для анализа производительности алгоритма (начните с некоторых алгоритмов сортировки) с удвоением входных размеров.



## 27.15. Список рекомендуемой литературы

- 27.1. Anthony M., & Bartlett P. L. (2009). *Neural Network Learning: Theoretical Foundations*. Cambridge University Press.
- 27.2. Berk R. A. (2020). *Statistical Learning from a Regression Perspective*. Springer.
- 27.3. Clarke B., Fokoue E., & Zhang H. H. (2009). *Principles and Theory for Data Mining and Machine Learning*. Springer.
- 27.4. Györfi L., Kohler M., Krzyzak A., & Walk H. (2002). *A Distribution-Free Theory of Nonparametric Regression*. Springer.
- 27.5. Hastie T., Tibshirani R., & Friedman J. (2009). *The Elements of Statistical Learning*. Springer.
- 27.6. Hastie T., Tibshirani R., & Wainwright M. (2015). *Statistical Learning with Sparsity: The Lasso and Generalizations*. CRC Press.
- 27.7. Huber P. J., & Ronchetti E. M. (2009). *Robust Statistics*. Wiley.
- 27.8. Liao S. P., Lin H. T., & Lin C. B. (2002). A note on the decomposition methods for support vector regression. *Neural Computation*, 14(6), 1267–1281.
- 27.9. Meyer D., Leisch F., & Hornik K. (2003). The support vector machine under test. *Neurocomputing*, 55(1), 169–186.
- 27.10. Mohri M., Rostamizadeh A., & Talwalkar A. (2018). *Foundations of Machine Learning*. MIT Press.
- 27.11. Wikipedia (2013). Regression analysis. [http://en.wikipedia.org/wiki/Regression\\_analysis](http://en.wikipedia.org/wiki/Regression_analysis). Accessed May 18, 2013.
- 27.12. Wikipedia (2016a). Local regression. [https://en.wikipedia.org/wiki/Local\\_regression](https://en.wikipedia.org/wiki/Local_regression). Accessed August 16, 2016.
- 27.13. Wikipedia (2016b). Multivariate adaptive regression splines. [https://en.wikipedia.org/wiki/Multivariate\\_adaptive\\_regression\\_splines](https://en.wikipedia.org/wiki/Multivariate_adaptive_regression_splines). Accessed August 16, 2016.
- 27.14. Wikipedia (2018). Simpson's paradox. [https://en.wikipedia.org/wiki/Simpson%27s\\_paradox](https://en.wikipedia.org/wiki/Simpson%27s_paradox). Accessed February 18, 2018.
- 27.15. Wilcox R. R. (2016). *Introduction to Robust Estimation and Hypothesis Testing*. Academic Press.
- 27.16. Witten I. H., Frank E., & Hall M. A. (2016). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.

## 28. Машинное обучение: кластеризация

### 28.1. Введение

В этой главе описываются реализации основных алгоритмов кластеризации с несколькими оригинальными вариантами алгоритмов. Некоторые из методов являются более теоретическими, но все же нуждаются в обсуждении из-за их популярности в литературе по машинному обучению. Наиболее интересным результатом можно считать алгоритм  $k$ -метода с выбором  $k$  по упрощенной метрике силуэта.

### 28.2. Постановка

Кластеризация при  $D \leq 3$  хорошо объясняется визуально. Задача сводится к выбору различных форм и дальнейшему выводу о том, что некоторые образцы похожи друг на друга. Для алгоритма даже решение о том, образует ли набор примеров одну группу или несколько, сложно, потому что, в отличие от человека, он не может использовать предварительные знания, чтобы определить природу данных и выявить закономерность (рис. 28.1).

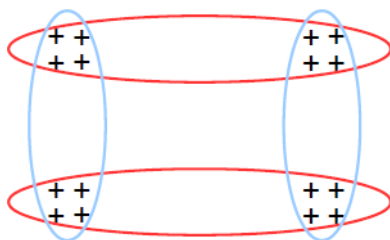


Рис. 28.1. Неясно, какой из двух кластеров предпочесть

Таким образом, математическая кластеризация — это некорректная задача, в которой нет ясности:

- ♦ выбор между альтернативными группами, предложенными по разным признакам — например, по росту или по весу. Выбор осуществляется на основе знания предметной области. Разные  $A$  предполагают конкретные модели, в которых задача корректна, но такие модели не обязательно хорошо подходят для данных. В частности, многие  $A$  предполагают специфичную для предметной области функцию расстояния  $d$ ;
- ♦ руководство по выбору числа классов  $k$ . Обычно принято максимизировать некоторую меру производительности, но истинное  $k$  обнаруживается редко и может не существовать, поэтому некоторые кластеры разделяются или объединяются;

- ◆ способ выбора признаков. Схожие эвристики, например выбор  $k$ , можно использовать с поиском-оберткой, но результат имеет гораздо большую дисперсию. Кроме того, обычно функции масштабируются, чтобы не оказывать чрезмерного предварительного влияния на некоторые из них. В отличие от контролируемых задач, для выбора хороших функций требуется много знаний в предметной области.

Несмотря на то что мы делаем достаточно предположений о модели, чтобы признать кластеризацию с выбранной моделью корректной, в найденных кластерах все же есть нестабильность, поскольку иногда находится одно многообещающее решение, а иногда другое (см. рис. 28.1). Кроме того, ни один  $A$  не может обнаружить кластеры, образованные разделенными подкластерами так, что между ними есть другие кластеры. Напротив, для задач с учителем отдельные данные обычно не представляют проблемы, если работать с сильными моделями.

Если вся информация о наборе данных доступна через  $d$ , ни один алгоритм не может удовлетворить всем аксиомам Клейнберга (см. [28.18]):

- ◆ масштабирование  $d$  приводит к получению такой же кластеризации;
- ◆ возможно любое разделение;
- ◆ изменение  $d$  такое, что примеры в одном и том же кластере ближе, а в разных кластерах дальше, приводит к получению такой же кластеризации.

Последнее, однако, имеет смысл только для хорошо разделенных кластеров (на рис. 28.2. кластеры таковыми не являются). Это желательно, а не обязательно, хотя и слегка связано с ошибкой аппроксимации.

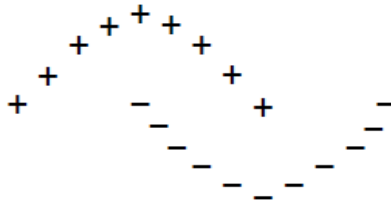


Рис. 28.2. Недостаточно разделенные кластеры

Несмотря на проблемы, некоторые  $A$  создают из некоторых данных полезные кластеры. Существует много неразмеченных данных, и автоматическое обнаружение кластеров в легко разделяемых данных — это очень ценное явление. Результаты кластеризации носят исследовательский характер. Существует мнение, что они просты или не интересны.

Кластеризация сильно отличается от автоматического кодирования. Оба метода являются неконтролируемыми, но цель кластеризации состоит не в том, чтобы реконструировать  $X$ , а в том, чтобы сгруппировать связанные  $x$  с использованием расстояния реконструкции.

Теоретически, если все данные известны, то известна и их плотность. Но пользы от этого нет, потому что даже если рассматривать режимы PDF как кластеры, все равно неясно, что такое  $k$  или как именно выполнить разделение на  $k$  режимов. Все  $A$  нуждаются в дополнительных предположениях.

## 28.3. Внешняя оценка

Нужно сравнить кластеризацию размеченных данных с метками. Что касается классификации, все показатели выводятся из матрицы непредвиденных обстоятельств. Количество строк =  $k$ , а количество столбцов = количеству классов данных. Ячейка (строка, столбец) = количеству примеров с меткой = столбцу и разделу кластера = строке:

```
template<typename DATA> Matrix<int> clusterContingencyMatrix(
    Vector<int> const& assignments, DATA const& data)
{ // строка - это раздел, а столбец - это метка
    int n = assignments.getSize();
    assert(n == data.getSize());
    int k = valMax(assignments.getArray(), n) + 1,
        nClasses = findNClasses(data);
    Matrix<int> counts(k, nClasses);
    for(int i = 0; i < n; ++i)
        ++counts(assignments[i], data.getY(i));
    return counts;
}
```

В качестве метрики качества можно использовать точность, которая была бы получена, если бы кластеризация использовалась для предсказания меток. Простой, но не лучший

вариант — это *чистота*  $= \frac{\sum_r \max_c n_{rc}}{n}$ . Она предполагает, что метка каждого кластера

является меткой большинства и взвешивает кластеры по размеру:

```
double clusterPurity(Matrix<int> const& counts)
{
    int sum = 0, total = 0;
    for(int i = 0; i < counts.rows; ++i)
    {
        int maxI = 0;
        for(int j = 0; j < counts.columns; ++j)
        {
            total += counts(i, j);
            maxI = max(maxI, counts(i, j));
        }
        sum += maxI;
    }
    return sum * 1.0/total;
}
```

Чистота не учитывает идентификацию кластера, т. е. явно не штрафует разбиение метки на несколько кластеров. Штрафование выполняется неявно, если  $k$  = количеству меток. Для  $A$ , который выбирает  $k$  автоматически, чистота неприемлема, потому что большее  $k$  обычно имеет лучшую чистоту случайно.

Таким образом, чистота не является хорошей сравнительной мерой, но хорошей описательной, потому что в некоторых приложениях переоценка  $k$  стоит намного меньше, чем смешивание кластеров. В частности, если требуется сокращение данных, высокая чистота может привести к успеху, даже если  $k$  сильно завышено, — это может быть важно, если классы нелегко разделить.

Вы можете вычислить ошибку классификации непосредственно путем сведения к задаче о назначениях (см. главу 11. *Алгоритмы графов*). Это избавляет от необходимости перечислять все назначения классов меткам, чтобы найти лучший. Чтобы разрешить решатель задач с поиском кратчайшего пути Дейкстры, затраты корректируются так, чтобы они были неотрицательными, используя формулу  $\text{cost}_{rc} = n - n_{rc}$ :

```
double clusterClassificationAccuracy(Matrix<int> const& counts)
{
    int n = 0, sum = 0;
    for(int i = 0; i < counts.rows; ++i)
        for(int j = 0; j < counts.columns; ++j) n += counts(i, j);
    Vector<pair<pair<int, int>, double> > allowedMatches;
    for(int i = 0; i < counts.rows; ++i)
        for(int j = 0; j < counts.columns; ++j) allowedMatches.append(
            make_pair(make_pair(i, counts.rows + j), n - counts(i, j)));
    Vector<pair<int, int> > matches = assignmentProblem(counts.rows,
        counts.columns, allowedMatches);
    assert(matches.getSize() == min(counts.rows, counts.columns));
    for(int i = 0; i < matches.getSize(); ++i)
        sum += counts(matches[i].first, matches[i].second - counts.rows);
    return sum * 1.0/n;
}
```

У ошибки классификации есть некоторые недостатки (см. [28.1]), поэтому лучше не использовать ее в качестве основного показателя сравнения. В частности, его интуитивное значение становится неясным, если оценочное значение  $k$  не совпадает с количеством меток. Но это также хорошая описательная мера.

Другой подход называется *индексом Рэнда* — это доля всех возможных пар примеров, которые одинаковы в обеих кластеризациях. Пара считается одинаковой, если оба примера принадлежат либо к одному, либо к разным кластерам. Этот индекс можно вычислить из матрицы случайности (см. [28.1]) как:

$$1 - \frac{m_1 + m_2 - \sum_{rc} \binom{n_{rc}}{2}}{M},$$

где

$$m_1 = \sum_r \binom{\sum_c n_{rc}}{2}, \quad m_2 = \sum_c \binom{\sum_r n_{rc}}{2} \quad \text{и} \quad M = \binom{n}{2}.$$

Полученное значение  $\in [0, 1]$ .

Хотя индекс Рэнда — это полезная метрика, но пары, которые различаются, имеют тенденцию доминировать в значении. В частности, для больших  $k$  ожидаемое значение высокое. Нормализацию можно выполнить, создав *скорректированный индекс*

*Рэнда*  $= \frac{R - E[R]}{1 - E[R]}$ .  $E[R]$  вычисляется при условии, что суммы строк и столбцов посто-

яны, а количество ячеек гипергеометрическое — такое, что

$$E\left[\sum_{rc} \binom{n_{rc}}{2}\right] = \frac{m_1 m_2}{M}.$$

Остальные компоненты формулы для  $R$  остаются постоянными, поэтому после упрощения математически скорректированный индекс Рэнда равен

$$\frac{\sum \sum \binom{n_{rc}}{2} - \frac{m_1 m_2}{M}}{(m_1 + m_2) / 2 - m_1 m_2 / M}.$$

Верхняя граница по-прежнему равна 1, но значение может быть ниже 0 (случайная кластеризация) или даже  $-1$ :

```
double nChoose2(int n){return n * (n - 1)/2;}
double AdjustedRandIndex(Matrix<int> const& counts)
{
    int total = 0, SumRC2 = 0, SumR2 = 0, SumC2 = 0;
    for(int i = 0; i < counts.rows; ++i)
    {
        int sumI = 0;
        for(int j = 0; j < counts.columns; ++j)
        {
            int c = counts(i, j);
            total += c;
            sumI += c;
            SumRC2 += nChoose2(c);
        }
        SumR2 += nChoose2(sumI);
    }
    for(int j = 0; j < counts.columns; ++j)
    {
        int sumJ = 0;
        for(int i = 0; i < counts.rows; ++i) sumJ += counts(i, j);
        SumC2 += nChoose2(sumJ);
    }
    double EV = SumR2 * SumC2 * 1.0/nChoose2(total);
    return (SumRC2 - EV)/(0.5 * (SumR2 + SumC2) - EV);
}
```

$E$ [скорректированный индекс Рэнда] асимптотически инвариантно в  $k$  (см. [28.13]), что, несмотря на отсутствие асимптотической инвариантности, позволяет сравнивать кластеризации с разными  $k$ . Интуитивно понятно, что для фиксированного  $k$  корректировка представляет собой линейное преобразование, поскольку ожидаемое значение является постоянным, но для конкретного  $k$  это другое линейное преобразование.

Скорректированный индекс Рэнда является хорошим сравнительным и описательным показателем. Для интерпретации представляется полезным следующее эмпирическое правило: значение  $\geq 0,8$  соответствует степени А,  $0,6 \leq \text{значение} < 0,8$  соответствует степени В и т. д.

Еще одна интересная метрика — точность классификатора, обученного на основе кластеризации, на другом наборе тестов. Она, вероятно, наиболее полезна при работе со случайным лесом, т. к. сводит к минимуму предвзятость введения классификатора. Также может быть полезно обучить дерево решений и вручную проверить его, чтобы попытаться понять логику или просто подсчитать количество узлов:

```
template<typename CLASSIFIER, typename DATA> double findTestCAcc(
    DATA const& train, Vector<int> assignments, DATA const& test)
{
    assert(train.getSize() == assignments.getSize());
    RelabeledData<DATA> rd(train);
    rd.labels = assignments;
    CLASSIFIER c(rd);
    Vector<int> assignmentsTest(test.getSize());
    for(int i = 0; i < test.getSize(); ++i)
        assignmentsTest[i] = c.predict(test.getX(i));
    return clusterClassificationAccuracy(
        clusterContingencyMatrix(assignmentsTest, test));
}
```

Кластеризация может быть произвольной, но поддающейся обучению, — например, разделение по одной переменной, так что это может быть хорошей мерой для обнаружения такой ситуации.

## 28.4. Внутренняя оценка и выбор количества кластеров

У новых данных нет помеченных примеров, на которых можно было бы провести внешнюю оценку. Поэтому нужно вычислить статистику, которая дает подсказки о качестве кластеризации. Самая очевидная из них — универсальный MDL. Для вероятностных моделей  $MDL = \text{количество битов для кодирования данных} + \text{количество битов для кодирования параметров} = \text{константа} + \log(\text{вероятность данных}) + \text{количество оцениваемых параметров} \times \text{среднее количество битов на параметр}$  (этот множитель постоянный, потому что вероятность пропорциональна только вероятности заданных параметров). Поскольку в лучшем случае усреднения параметры сходятся со скоростью  $O(1/\sqrt{n})$  по CLT, для каждого параметра требуется около  $\lg(n)/2$  бита. Выражая битовый размер в натуральном виде,  $2(MDL\text{-константа}) = \text{статистика } BIC = 2LL + p \ln(n)$ , где  $LL$  = логарифмическая вероятность, а  $p$  = количество оцениваемых параметров. Но метрика BIC доступна только у вероятностных моделей и не обязательно является наиболее эффективной мерой.

Хотя количество кластеров у многих предметных областей часто известно (например, десять), нам нужна возможность определять его автоматически. При наличии статистического показателя качества, такого как BIC, который должным образом штрафует большее оценочное значение  $k$ , можно начать с  $k = 2$  и увеличивать его по мере улучшения статистики до безопасного предела, такого как  $\sqrt{n}$ .

*Индекс силуэта* (см. [28.1]) — это общая метрика, равная среднему силуэту по всем примерам. Здесь  $\text{силуэт} = \frac{b-a}{\max(b,a)}$ , где  $a$  = среднее расстояние от примера до других

примеров в своем кластере, а  $b = \min_c$  — среднее расстояние от примера до примеров в классе  $c$ , не содержащем этот пример:

```
template<typename DATA, typename DISTANCE> double clusterSilhouette(
    DATA const& data, Vector<int> const& assignments, DISTANCE const& d)
```

```

{
    int n = assignments.getSize();
    assert(n > 0);
    int k = valMax(assignments.getArray(), n) + 1;
    double sum = 0;
    for(int i = 0; i < n; ++i)
    {
        int c = assignments[i];
        Vector<double> ds(k);
        Vector<int> sizes(k);
        for(int j = 0; j < n; ++j) if(i != j)
        {
            int c2 = assignments[j];
            ++sizes[c2];
            ds[c2] += d(data.getX(i), data.getX(j));
        }
        for(int j = 0; j < k; ++j) if(sizes[j]) ds[j] /= sizes[j];
        double ai = ds[c], bi = numeric_limits<double>::infinity();
        for(int j = 0; j < k; ++j) if(j != c) bi = min(bi, ds[j]);
        sum += (bi - ai)/max(bi, ai);
    }
    return sum/n;
}

```

Но вычисления выполняются дорого — за время  $O(n^2k)$ . Поэтому лучше использовать *упрощенный силуэт*, основанный на расстояниях до выбранных репрезентативных точек кластеров, а не на средних значениях. Эта метрика считается недостаточно общей (см. [28.15]) — она полезна для репрезентативных алгоритмов. Если  $d$  — это евклидово расстояние, для любого  $x \notin$  кластеру с центроидом  $c$ , образованным  $x_i$ :

$$\sum d(x, x_i)^2 = n_c \left( d(x, c) + \sum d(c, x_i)^2 \right).$$

Доказательство: для одномерного случая

$$\begin{aligned}
 \sum d(x - x_i)^2 &= nc \left( x_i^2 + \frac{\sum x_i^2}{nc} - 2xc \right) = \\
 &= nc \left( (x - c)^2 + \frac{\sum x_i^2}{nc} - c^2 \right) = nc \left( (x - c)^2 + \text{Var}(x_i) \right)
 \end{aligned}$$

обобщается на  $D > 1$ , потому что квадрат евклидова расстояния является аддитивным. Таким образом, упрощение эффективно игнорирует дисперсию кластера — та же интерпретация должна применяться для других  $d$ , где такое разложение не выполняется. Эксперименты, описанные в работе [28.24], показывают, что производительность упрощенной версии практически эквивалентна:

```

template<typename DATA, typename DISTANCE, typename REPS>
double clusterSimplifiedSilhouette(DATA const& data,
    Vector<int> const& assignments, REPS const& r, DISTANCE const & d)
{
    int n = assignments.getSize();
    assert(n > 0);

```



```

int k = valMax(assignments.getArray(), n) + 1;
double sum = 0;
for(int i = 0; i < n; ++i)
{
    int c = assignments[i];
    double ai = d(data.getX(i), r[c]),
        bi = numeric_limits<double>::infinity();
    for(int j = 0; j < k; ++j) if(j != c)
        bi = min(bi, d(data.getX(i), r[j]));
    sum += (bi - ai)/max(bi, ai);
}
return sum/n;
}

template<typename DATA> double clusterSimplifiedSilhouetteL2(DATA const& data,
    Vector<int> const& assignments)
{
    int k = valMax(assignments.getArray(), assignments.getSize()) + 1;
    return clusterSimplifiedSilhouette(data, assignments, findCentroids(data,
        k, assignments), EuclideanDistance<NUMERIC_X>::Distance());
}

```

Если есть результат кластеризации, полезно иметь как назначения, так и значение внутреннего индекса. Вычисление последнего обычно стоит меньше, если  $A$  и индекс подобраны правильно. Если нет, значение индекса устанавливается равным  $\infty$  и вычисляется позже:

```

struct ClusterResult
{
    Vector<int> assignments;
    double comparableInternalIndex;
    ClusterResult(Vector<int> const& theAssignments, double theCIP =
        numeric_limits<double>::infinity()): assignments(theAssignments),
        comparableInternalIndex(theCIP) {}
};

```

Это позволяет оценить  $k$ :

1. Начать с  $k = 2$ .
2. До достижения максимального  $k = \sqrt{n}$  :
3. Сгруппировать с текущим  $k$  и вычислить значение индекса.
4. Если есть худшее значение, вернуть предыдущую кластеризацию.

```

template<typename CLUSTERER, typename DATA, typename PARAMS> ClusterResult
    findClustersAndK(DATA const& data, CLUSTERER const& c, PARAMS const& p,
    int maxK = -1)
{
    if(maxK == -1) maxK = sqrt(data.getSize());
    Vector<int> dummy;
    ClusterResult best(dummy);
    for(int k = 2; k <= maxK; ++k)
    {
        ClusterResult result = c(data, k, p);
    }
}

```

```

        if(isfinite(result.comparableInternalIndex) &&
           result.comparableInternalIndex < best.comparableInternalIndex)
            best = result;
        else break;
    }
    return best;
}

template<typename CLUSTERER, typename PARAMS = EMPTY> struct FindKClusterer
{
    CLUSTERER c;
    PARAMS p;
    FindKClusterer(PARAMS const& theP = PARAMS()): p(theP){}
    template<typename DATA> ClusterResult operator()(DATA const& data, int k)
        const{return c(data, k, p);}
    template<typename DATA> ClusterResult operator()(DATA const& data)const
        {return findClustersAndK(data, c, p);}
};

```

Хотя параметры обычно не передаются в кластеризацию  $A$ , сама по себе такая возможность полезна. Что касается контролируемых задач, следующая оболочка допускает фиктивные параметры:

```

template<typename CLUSTERER> struct NoParamsClusterer
{
    CLUSTERER c;
    template<typename DATA> ClusterResult operator()(DATA const& data, int k,
        EMPTY const& p)const{return c(data, k);}
    template<typename DATA> ClusterResult operator()(DATA const& data,
        EMPTY const& p)const{return c(data);}
};

```

Технически можно использовать внутренние индексы для выбора значений других параметров и, возможно, даже выполнять поиск оболочки для выбора функций, но это может привести к переобучению из-за слишком большого объема работы с одними и теми же данными. Метрику силуэта можно использовать потому, что ее ожидаемое значение увеличивается вместе с  $D$  (см. [28.23]), поэтому поиск обертки с использованием упрощенного силуэта вызывает сомнения.

## 28.5. Расчет устойчивости

Общий алгоритм устойчивости, основанный на функции потерь, здесь не применяется, потому что даже для разумных потерь, таких как  $k$ -среднее, неясно, как сформировать основной предиктор.

Базовый метод с использованием алгоритма кластеризации  $A$  (см. [28.28]):

1.  $B$  раз, где  $B = 100$ :
2. Кластеризация двух ресемплов данных с начальной загрузкой.
3. Рассчитать внешний индекс, используя образцы, общие для обоих случаев.
4. Устойчивость = среднее значение показателей.

Обратите внимание, что *неустойчивость* ( $= 1 - \text{устойчивость}$ ) естественным образом растет с ростом  $k$ , поэтому ее интерпретация может быть затруднена. Ранее предлагались некоторые методы нормализации, но они сомнительны. Таким образом, скорректированный индекс Рэнда кажется хорошим вариантом из-за возможности коррекции:

```
Matrix<int> clusterOnlyContingencyMatrix(Vector<int> const& assignments1,
    Vector<int> const& assignments2)
{
    int n = assignments1.getSize();
    assert(n == assignments2.getSize());
    int k1 = valMax(assignments1.getArray(), n) + 1,
        k2 = valMax(assignments2.getArray(), n) + 1;
    Matrix<int> counts(k1, k2);
    for(int i = 0; i < n; ++i) ++counts(assignments1[i], assignments2[i]);
    return counts;
}

template<typename CLUSTERER, typename PARAMS, typename DATA> double
findStability(CLUSTERER const& c, PARAMS const& p, DATA const& data,
    int k = -1, int B = 100)
{
    double sum = 0;
    int n = data.getSize();
    for(int j = 0; j < B; ++j)
    { // отрисовка и начальная загрузка кластеров
        Vector<int> assignments[2] = {Vector<int>(n, -1), Vector<int>(n, -1)};
        for(int l = 0; l < 2; ++l)
        {
            PermutedData<DATA> dataP(data);
            for(int i = 0; i < n; ++i) dataP.addIndex(GlobalRNG().mod(n));
            Vector<int> pAssignments = k == -1 ? c(dataP, p).assignments :
                c(dataP, k, p).assignments;
            for(int i = 0; i < n; ++i)
                assignments[l][dataP.permutation[i]] = pAssignments[i];
        } // вычисление пересечений
        for(int i = n - 1; i >= 0; --i)
            if(assignments[0][i] == -1 || assignments[1][i] == -1)
                for(int l = 0; l < 2; ++l)
                {
                    assignments[l][i] = assignments[l].lastItem();
                    assignments[l].removeLast();
                }
        // нужно больше 1 примера, чтобы избежать NaN.
        // На самом деле, ГОРАЗДО больше
        if(assignments[0].getSize() > 1) sum += AdjustedRandIndex(
            clusterOnlyContingencyMatrix(assignments[0], assignments[1]));
    }
    return sum/B;
}
```

Обязательным теоретическим требованием любого хорошего  $A$  является сходимость в устойчивости, т. е. для любого набора данных и разумного определения устойчивости

при  $n \rightarrow \infty$  устойчивость  $A$  сходится к фиксированному значению. Вероятно, это лучшее, на что можно надеяться, потому что алгоритм можно заставить случайным образом чередоваться между двумя одинаково хорошими кластеризациями по мере добавления новых данных и привести к неустойчивости, но сама устойчивость должна сходиться. Обратите внимание, что устойчивость является свойством как  $A$ , так и набора данных.

Это определение устойчивости в каком-то смысле более правильное, чем общее, потому что оно учитывает структуру, а главный предиктор — нет. Например, для задачи классификации очень разные разбиения могут иметь одинаковую точность.

Использование устойчивости для выбора  $k$  и, возможно, других параметров — это интересный, но потенциально очень медленный метод. Также (см. [28.28]):

- ◆ не существует теоретических гарантий или обширных экспериментальных доказательств хорошей производительности;
- ◆ если  $A$  неустойчив, например когда его предположения существенно неверны, результаты непредсказуемы.

Таким образом, устойчивость кажется просто полезной метрикой качества.

## 28.6. Кластеризация в евклидовом пространстве

Для данного  $k$  и векторного пространства  $X$  простой и эффективной является метрика  $k$ -средних (рис. 28.3):

1. Выбрать начальные кластеры.
2. Пока алгоритм не сойдется или не будет достигнут некоторый предел итераций:
3.     Вычислить центроиды текущих кластеров.
4.     Назначить каждую точку кластеру ближайшему центроиду.

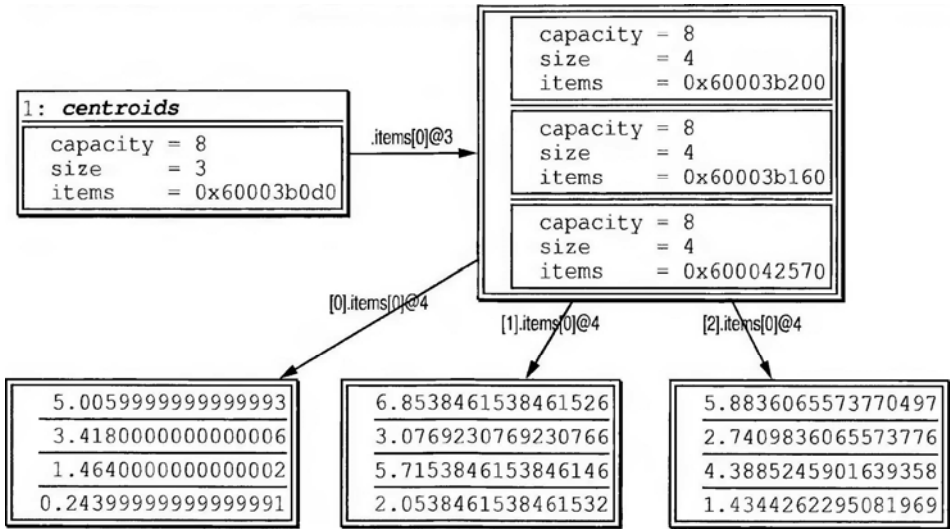


Рис. 28.3. Схема памяти кластеризации  $k$ -средних на данных о цветках ириса

Хорошая инициализация особенно важна, потому что оптимизация, выполняемая алгоритмами кластеризации, является локальной. Поиск оптимальных центроидов эквивалентен NP-полному местоположению объекта, для которого  $k$ -средние являются хорошим локальным поиском, — цель никогда не увеличивается (но технически  $k$ -средние могут не найти локальный минимум, если они имеют равные расстояния (см. [28.18])). Алгоритм аппроксимации с коэффициентом аппроксимации  $E[O(\ln(k))]$  эффективен для инициализации (см. [28.3]). Идея состоит в том, чтобы выбрать центроиды, которые отличаются друг от друга:

1. Выбрать случайную выборку в качестве первого центроида.
2.  $k - 1$  раз:
3. Выбрать пример с относительной вероятностью  $\min d(x_i, c_j)^2$  в качестве следующей центроиды, где  $c_j$  — это уже вычисленный центроид.

Метод применяется к метрике  $d$  и гарантирует тот же ожидаемый коэффициент аппроксимации (с точностью до константы). Единственное изменение в вычислениях заключается в том, что не нужно возводить расстояния в квадрат.

Чтобы присвоить каждый пример ближайшему центроиду, используйте поиск ближайшего соседа по дереву VP. Судя по моим экспериментам, этот метод работает немного быстрее, чем дерево  $k$ -d, несмотря на то, что в последнем не нужно вычислять квадратные корни. При выборе модели используется упрощенный силуэт:

```
template<typename DATA, typename DISTANCE> Vector<int>
    findKMeansPPCentroids(DATA const& data, int k, DISTANCE const& d,
        bool isMetric = false)
{ // алгоритм аппроксимации для инициализации центроида
    int n = data.getSize();
    assert(n > 0 && k <= n);
    Vector<double> closestDistances(n, numeric_limits<double>::infinity());
    Vector<int> centroids(1, GlobalRNG().mod(n));
    for(int i = 1; i < k; ++i)
    { // повторное вычисление ближайших расстояний до центра
        for(int j = 0; j < n; ++j)
            closestDistances[j] = min(closestDistances[j],
                d(data.getX(j), data.getX(centroids.lastItem())));
        // выбор следующего центра пропорционально квадрату ближайшего расстояния
        Vector<double> probs(n);
        for(int j = 0; j < n; ++j)
        {
            probs[j] = closestDistances[j];
            if(!isMetric) probs[j] *= closestDistances[j];
        }
        normalizeProbs(probs);
        AliasMethod a(probs);
        centroids.append(a.next());
    }
    return centroids;
}

template<typename DATA> Vector<typename DATA::X_TYPE> assemblePrototypes(
    DATA const& data, Vector<int> const& medoids)
{ // вспомогательная функция для извлечения центров кластеров из данных и индексов
    Vector<typename DATA::X_TYPE> result(medoids.getSize());
```

```

    for(int i = 0; i < medoids.getSize(); ++i)
        result[i] = data.getX(medoids[i]);
    return result;
}

template<typename DATA> Vector<NUMERIC_X> findCentroids(DATA const& data,
    int k, Vector<int> const& assignments)
{
    Vector<int> counts(k);
    Vector<NUMERIC_X> centroids(k, data.getX(0) * 0);
    for(int i = 0; i < data.getSize(); ++i)
    {
        ++counts[assignments[i]];
        centroids[assignments[i]] += data.getX(i);
    }
    for(int i = 0; i < k; ++i) centroids[i] *= 1.0/counts[i];
    return centroids;
}

struct KMeans
{
    typedef EuclideanDistance<NUMERIC_X>::Distance EUC_D;
    template<typename DATA> static bool findAssignments(DATA const& data,
        Vector<NUMERIC_X> const& centroids, Vector<int>& assignments)
    { // присвоение всех примеров ближайшим центроидам
        VpTree<NUMERIC_X, int, EUC_D> t;
        bool converged = true;
        for(int i = 0; i < centroids.getSize(); ++i) t.insert(centroids[i], i);
        // назначение всех точек ближайшему центроиду
        for(int i = 0; i < data.getSize(); ++i)
        {
            int best = t.nearestNeighbor(data.getX(i))->value;
            if(best != assignments[i])
            { // выполняется, если присвоение не меняется
                converged = false;
                assignments[i] = best;
            }
        }
        return converged;
    }

    template<typename DATA> ClusterResult operator()(DATA const& data,
        int k, int maxIterations = 1000) const
    {
        assert(k > 0 && k <= data.getSize() && data.getSize() > 0);
        Vector<int> assignments(data.getSize());
        findAssignments(data, assemblePrototypes(data, findKMeansPPCentroids(
            data, k, EUC_D())), assignments);
        for(int m = 0; m < maxIterations; ++m) if(findAssignments(data,
            findCentroids(data, k, assignments), assignments)) break;
        return ClusterResult(assignments, -clusterSimplifiedSilhouetteL2(data,
            assignments));
    }
};

typedef FindKClusterer<NoParamsClusterer<KMeans> > KMeansGeneral;
```

Алгоритм нужно повторить несколько раз (число 10 кажется разумным), пока целевая функция не улучшится:

```
template<typename DATA> double kMeansSimpSil(DATA const& data,
    Vector<int> const& assignments)
{
    int n = assignments.getSize();
    assert(n > 0);
    int k = valMax(assignments.getArray(), n) + 1;
    Vector<NUMERIC_X> centroids = findCentroids(data, k, assignments);
    typename EuclideanDistance<NUMERIC_X>::DistanceIncremental d;
    double sum = 0;
    for(int i = 0; i < n; ++i)
        sum += d(data.getX(i), centroids[assignments[i]]);
    return sum;
}

struct RepeatedKMeans
{
    template<typename DATA> ClusterResult operator()(DATA const& data,
        int k, int maxIterations = 1000, int nRep = 10) const
    {
        KMeans km;
        ClusterResult best = km(data, k, maxIterations);
        double ss = kMeansSimpSil(data, best.assignments);
        while(--nRep)
        {
            ClusterResult result = km(data, k, maxIterations);
            double ssNew = kMeansSimpSil(data, result.assignments);
            if(ssNew < ss)
            {
                ss = ssNew;
                best = result;
            }
        }
        return best;
    }
};

typedef FindKClusterer<NoParamsClusterer<RepeatedKMeans> > RKMeansGeneral;
```

Самая дорогая операция в цикле — это поиск ближайших центроидов. Он занимает ожидаемое время  $O(n \lg(k))$  при условии эффективного поиска ближайшего соседа, но может выполняться за  $O(nDk)$ . Для сходимости может потребоваться экспоненциально много итераций (см. [28.17]), поэтому нужно ограничить количество итераций большой константой, что кажется эффективным на практике. Сглаженная сложность полиномиальна (см. [28.2]).

Метрика  $k$ -среднего ухудшается из-за высокой ошибки аппроксимации, так же как ближайший средний классификатор. В частности, она может различать только хорошо разделенные формы, но все же не обязательно хорошо, потому что маленькая форма может отнять поддержку у более крупной (рис. 28.4).

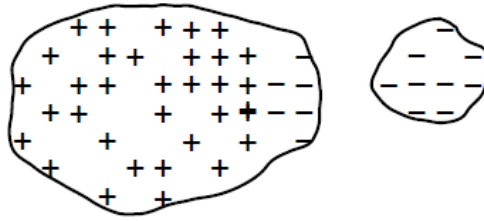


Рис. 28.4. Эффект формы на  $k$ -средних — большая группа забирает несколько экземпляров у меньшей

## 28.7. Кластеризация в метрическом пространстве

Для вычисления центроидов нужно векторное пространство. Расширение метрики  $X$  требует модификаций, таких как  $k$ -медоиды (см. [28.13]), где репрезентативные точки (называемые *медоидами*) выбираются из данных. Мы ищем одну из  $\binom{n}{k}$  комбинаций точек таких, что сумма расстояний примеров до их ближайшего медоида является минимальной.

Алгоритм локального поиска лучшей комбинации:

1. Начать с некоторого начального решения — в нашем случае используется инициализация  $k$ -средних ++.
2. До тех пор, пока не будут удовлетворены некоторые критерии завершения:
3. Выбрать медоид и немедоид и менять их местами, пока целевой критерий не снизится.

Выбор пар для обмена осуществляется несколькими способами:

- ♦ оригинальный *алгоритм РАМ* (см. [28.15]) рассматривает все возможные пары, выбирает наилучшие и обменивает их до тех пор, пока не будет достигнута сходимость, но это дорого с точки зрения вычислений;
- ♦ более масштабируемый метод (см. [28.1]) заключается в выборе случайных пар и обмене на некоторое количество итераций.

Лучшая стратегия (нечто среднее) состоит в рассмотрении всех пар, образованных случайным немедоидом и всеми медоидами. При надлежащем кэшировании расстояний стоимость итерации получается примерно такая же, как и для случайной пары, потому что узким местом является вычисление всех расстояний от выбранного немедоида до других выборок, а результат повторно используется при рассмотрении всех медоидов. В частности, для любого примера нужно сохранять его расстояние до каждого текущего медоида. Для этого требуется  $O(nk)$  памяти, что обычно ненамного больше минимального  $O(n)$ .

При рассмотрении пар для любой пары есть два случая, в зависимости от того, ближе ли она к:

- ♦ медоиду — проверить все остальные медоиды и кандидатов, чтобы найти ближайший;
- ♦ кандидату — самый близкий кандидат.



Используйте случайную перестановку для управления порядком, в котором рассматриваются медоиды:

- ◆ малое  $n$  — сходится, если медоиды не изменились после полного прохождения перестановки. В каждом проходе нужно учитывать разные случайные перестановки, чтобы разнообразить локальный поиск;
- ◆ большое  $n$  — по умолчанию предельное число итераций равно 1000. В задаче сортировки купонов необходимо  $E[k \log(k)]$  примеров, чтобы получить все представленные классы при условии сбалансированных данных, поэтому для разумного  $k$  значения 1000 должно быть достаточно, а найденные медоиды должны быть достаточно близки к истинным.

Чтобы выбрать  $k$ , можно использовать упрощенный силуэт:

```
template<typename DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
struct KMedoids
{
    static bool isIMedoid(int i, Vector<int> const& medoids)
    {
        for(int j = 0; j < medoids.getSize(); ++j)
            if(medoids[j] == i) return true;
        return false;
    }
    template<typename DATA> ClusterResult operator()(DATA const& data, int k,
        int maxRounds = 1000) const
    {return findClusters(data, k, maxRounds).first;}
    template<typename DATA> static pair<ClusterResult, Vector<int>> >
        findClusters(DATA const& data, int k, int maxRounds = 1000)
    { // инициализация текущих медоидов
        int n = data.getSize();
        assert(k > 0 && k <= n && n > 0);
        DISTANCE d;
        Vector<int> perm(n), medoids(findKMeansPPCentroids(data, k, d, true)),
            assignments(n);
        Vector<Vector<double>> dCache(n, Vector<double>(k));
        for(int i = 0; i < n; ++i)
        { // вычисление, присвоение и кеширование расстояний
            for(int j = 0; j < k; ++j)
                dCache[i][j] = d(data.getX(i), data.getX(medoids[j]));
            int best = argMin(dCache[i].getArray(), k);
            assignments[i] = best;
        }
        for(int i = 0; i < n; ++i) perm[i] = i; // инициализация перестановки
        bool converged = false;
        while(!converged)
        {
            converged = true;
            GlobalRNG().randomPermutation(perm.getArray(), n);
            for(int i = 0; i < n && maxRounds > 0; ++i)
            {
                if(isIMedoid(perm[i], medoids)) continue;
                Vector<double> tempDs(n);
```

```

    for(int l = 0; l < n; ++l)
        tempDs[l] = d(data.getX(perm[i]), data.getX(l));
    int bestJ = -1;
    double bestDiff;
    for(int j = 0; j < k; ++j)
    {
        double DSumDiff = 0;
        for(int l = 0; l < n; ++l)
        {
            double dOld = dCache[l][assignments[l]];
            if(assignments[l] == j)
            {
                dCache[l][j] = tempDs[l];
                DSumDiff += valMin(dCache[l].getArray(), k) - dOld;
                dCache[l][j] = dOld;
            }
            else if(tempDs[l] < dOld) DSumDiff += tempDs[l] - dOld;
        }
        if(bestJ == -1 || DSumDiff < bestDiff)
        {
            bestDiff = DSumDiff;
            bestJ = j;
        }
    }
    if(bestDiff < 0)
    {
        converged = false;
        for(int l = 0; l < n; ++l)
        {
            dCache[l][bestJ] = tempDs[l];
            if(assignments[l] == bestJ) assignments[l] =
                argMin(dCache[l].getArray(), k);
            else if(tempDs[l] < dCache[l][assignments[l]])
                assignments[l] = bestJ;
        }
        medoids[bestJ] = perm[i];
    }
    --maxRounds;
}

return make_pair(ClusterResult(assignments,
    -clusterSimplifiedSilhouette(data, assignments,
        assemblePrototypes(data, medoids), d)), medoids);
}

};

template<typename> DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
using KMedGeneral = FindKClusterer<NoParamsClusterer<KMedoids<DISTANCE> > >;

Что касается k-средних, повторение может оказаться полезным:

template<typename> DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
struct RepeatedKMedoids

```

```

{
    template<typename DATA> static double kMedS(DATA const& data,
        Vector<int> const& assignments, Vector<int> const& medoids)
    {
        int n = assignments.getSize();
        assert(n > 0);
        int k = valMax(assignments.getArray(), n) + 1;
        Vector<typename DATA::X_TYPE> m = assemblePrototypes(data, medoids);
        DISTANCE d;
        double sum = 0;
        for(int i = 0; i < n; ++i) sum += d(data.getX(i), m[assignments[i]]);
        return sum;
    }
    template<typename DATA> ClusterResult operator()(DATA const& data,
        int k, int maxRounds = 1000, int nRep = 10) const
    {
        KMedoids<DISTANCE> km;
        pair<ClusterResult, Vector<int> > best =
            KMedoids<DISTANCE>::findClusters(data, k, maxRounds);
        double s = kMedS(data, best.first.assignments, best.second);
        while(--nRep)
        {
            pair<ClusterResult, Vector<int> > result =
                KMedoids<DISTANCE>::findClusters(data, k, maxRounds);
            double sNew = kMedS(data, result.first.assignments, result.second);
            if(sNew < s)
            {
                s = sNew;
                best.first = result.first;
            }
        }
        return best.first;
    }
};

template<typename DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
using RKMedGeneral = FindKClusterer<NoParamsClusterer<RepeatedKMedoids<DISTANCE> > >;

```

Для очень больших  $n$ , если у вас нет места  $O(nk)$  для хранения кеша, вы можете хранить его на диске и использовать инкрементный алгоритм, такой как SGD. То есть прочитайте страницу ввода/вывода с примерами и их соответствующими расстояниями и перебирайте все примеры таким образом.

$k$ -медоиды дают лучшее понимание данных по сравнению с другими  $A$ , потому что кластеры определяются из конкретных примеров, которые обычно могут быть легко проверены человеком. Так что это, а также возможность работать с общими данными, делает  $k$ -медоиды первым  $A$ , который можно попробовать в качестве дерева решений для классификации.

## 28.8. Спектральная кластеризация

Пусть  $W$  — это матрица подобия, созданная из обучающих данных с использованием ядер или какого-либо другого метода. Можно представить, что это взвешенная матрица

смежности некоторого графа. Его *матрица степеней*  $D$  является диагональной матрицей, где  $D[i, i] = \sum_j W[i, j]$ . *Лапласиан графа*  $L = D - W$ . Это преобразование полезно, потому что если  $L$  имеет  $k$  компонентов связности, каждый из которых соответствует определенному кластеру, его  $k$  наименьших собственных значений равны 0, а соответствующие собственные векторы являются индикаторными векторами для строк, принадлежащих каждому кластеру (см. [28.1]). Таким образом, в этом случае они формируют набор признаков, который получается лучше исходного.

Кластеры редко бывают четко разделены, а метод генерации подобия точен по  $L$  и имеет  $k$  связанных компонентов. Но если компоненты связаны между собой соединениями с малым весом, взятие «первых»  $k$  собственных векторов по-прежнему позволяет получить хороший набор признаков.

Ядра часто используются для генерации  $W$ , но матрица получается плотной, и требуется знать параметры ядра. Можно сделать  $W$  разреженной, отрезав записи  $< \max(W[i, i], W[j, j]) \epsilon$  для  $\epsilon = 0,001$  или около того, но все же необходимо выбрать параметр, т. к. задача может быть чувствительной к масштабу. Более простой вариант — использовать симметричный генератор ближайших соседей, где  $W[i, j] = W[j, i]$ , если  $i$  — один из нескольких соседей  $j$ , или наоборот (оба не обязательно верны). Количество проверяемых соседей можно установить равным значению по умолчанию классификатора  $k$ -NN (см. главу 26. *Машинное обучение: классификация*).

Полученный алгоритм можно в некоторой степени подстроить:

1. Создать  $W$ , используя ближайших соседей или какой-либо другой метод.
2. Создать нормализованную  $L = I - D^{-0,5} W D^{-0,5}$  (обратите внимание, что  $D^{-0,5} W D^{-0,5} \neq D^{-1} W$ ).
3. Найти первые  $k$  собственные пары.
4. Создать новый набор данных, в котором вектор признаков  $x$  для примера  $i = \{\text{собственный вектор } [j][i] \text{ для } 0 \leq j < k\}$  нормирован так, что  $\|x\| = 1$ .
5. Используйте любой алгоритм кластеризации, такой как  $k$ -средних.

Симметризация позволяет получить более эффективный решатель собственных значений (см. главу 22. *Численные алгоритмы: введение и матричная алгебра*) и сохраняет свойство связного компонента вплоть до линейного преобразования. Нормализация  $x$  защищает от возможного отсутствия масштаба компонентов собственного вектора в одной и той же координате (см. [28.27]).

Когда нужно выбрать  $k$ , требуется подходящий индекс, основанный на окончательных заданиях. Среди используемых индексов лучшим выбором кажется силуэт (упрощенный силуэт не повышает эффективности), поэтому по умолчанию следует использовать его. Его вычисление выполняется намного быстрее, чем вычисление собственного значения. Обратите внимание, что использование индекса окончательного алгоритма кластеризации (здесь  $k$ -средних) — плохая идея, потому что он работает в другом пространстве признаков и принимает решения на основе последнего. Также выбор  $k$  собственных векторов решает несколько иную проблему с каждым  $k$ , поэтому его нельзя использовать для выбора  $k$ . Для вычисления  $L$  нужно применять алгебру разреженных матриц, хотя для получения собственных векторов необходимо преобразовать их в плотные:

```

template<typename DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
struct SpectralClusterer
{ // собственные числа и перестановка для сортировки
    typedef pair<pair<Vector<double>, Matrix<double> >, Vector<int> > EIGS;
    template<typename DATA> SparseMatrix<double> createLaplacian(
        DATA const& data) const
    { // настройка лапласиана kNN
        int n = data.getSize(), nNeighbors = lgFloor(n)/2 + 1;
                                                // kNN по умолчанию

        SparseMatrix<double> W(n, n);
        typedef VpTree<typename DATA::X_TYPE, int, DISTANCE> TREE;
        TREE tree;
        for(int i = 0; i < n; ++i) tree.insert(data.getX(i), i);
        for(int i = 0; i < n; ++i)
        {
            Vector<typename TREE::NodeType*> neighbors =
                tree.kNN(data.getX(i), nNeighbors + 1);
            for(int j = 0; j < neighbors.getSize(); ++j)
            { // первые nn обычно собственные
                int l = neighbors[j]->value;
                if(l != i)
                {
                    W.set(i, l, 1);
                    W.set(l, i, 1);
                }
            }
        }
        return W;
    }
    EIGS findLaplacianEigs(SparseMatrix<double> const& W) const
    { // нормализация
        int n = W.getRows();
        SparseMatrix<double> Dm05(n, n);
        for(int i = 0; i < n; ++i)
        {
            double di = 0;
            for(int j = 0; j < n; ++j) di += W(i, j);
            Dm05.set(i, i, 1/sqrt(di));
        }
        // поиск собственных
        EIGS eigs(QREigenSymmetric(toDense<double>(SparseMatrix<double>::
            identity(n) - Dm05 * W * Dm05)), Vector<int>(n));
        // сортировка перестановки
        for(int i = 0; i < n; ++i) eigs.second[i] = i;
        quickSort(eigs.second.getArray(), 0, n - 1,
            IndexComparator<double>(eigs.first.first.getArray()));
        return eigs;
    }
    template<typename DATA> ClusterResult operator()(DATA const& data, int k,
        EIGS const& eigs) const // вызов после k поисков
    { // создание новых признаков
        int n = data.getSize();
        assert(k > 0 && k < n);
    }

```

```

InMemoryData<NUMERIC_X, int> data2;
for(int i = 0; i < n; ++i)
{ // собственные векторы - это строки
    Vector<double> x(k);
    for(int j = 0; j < k; ++j)
        x[j] = eigs.first.second(eigs.second[j], i);
    double xNorm = norm(x); // нормализация
    if(xNorm > 0) x *= 1/xNorm;
    data2.addZ(x, 0);
} // кластер
RepeatedKMeans km;
ClusterResult result = km(data2, k);
result.comparableInternalIndex =
    -clusterSilhouette(data, result.assignments, DISTANCE());
return result;
}

template<typename DATA> ClusterResult operator()(DATA const& data, int k)
const // если k известно
{return operator()(data, k, findLaplacianEigs(createLaplacian(data)));}
template<typename DATA> ClusterResult operator()(DATA const& data) const
{ // если k не известно
    return findClustersAndK(data, *this,
        findLaplacianEigs(createLaplacian(data)));
}
};

```

Алгоритм является статистически согласованным при некоторых условиях, когда известно  $k$  (см. [28.27]). Его также можно рассматривать как попытку разрезать граф на  $k$  сбалансированных связанных компонентов оптимальным образом. Даже приближение оптимального сбалансированного разреза к постоянному коэффициенту является  $NP$ -трудным (см. [28.27]), поэтому такая эвристика имеет смысл. Поскольку для больших  $n$  (для моего компьютера это примерно  $> 5000$ ) будет исчерпана память, смешайте алгоритм с повторяющимися  $k$ -медоидами, чтобы получить хорошие результаты для произвольных расстояний:

```

template<typename DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
struct SpectralSmart
{
    RKMedGeneral<DISTANCE> km;
    SpectralClusterer<DISTANCE> s;
    template<typename DATA> bool useKMed(DATA const& data) const
        {return data.getSize() > 5000;} // для памяти
    // экономность и эффективность
    template<typename DATA> ClusterResult operator()(DATA const& data,
        int k) const
    {
        if(useKMed(data)) return km(data, k);
        else return s(data, k);
    }
    template<typename DATA> ClusterResult operator()(DATA const& data) const
    {
        if(useKMed(data)) return km(data);

```

```

    return s(data);
}
};

```

## 28.9. Эксперименты

Сравнения выполнены на обучающих наборах из 15 различных наборов данных (тех же, что и для классификации). Из-за исследовательского характера и сложности кластеризации результаты являются скорее доказательством концепции. Сообщается о средних рангах Фридмана. Рандомизированные алгоритмы запускались 5 раз (рис. 28.5).

	kMeansPP01	kMeansPP01R10	kMed01	kMed01R10	Spectral
A_Rand	3	3.07	3.13	2.87	2.33
Устойчивость	2.93	1.27	3.07	2.73	Нет данных

Рис. 28.5. Сравнение производительности с известными  $k$

Для угадывания значения  $k$  индекс Рэнда подходит лучше, чем относительная ошибка в найденном  $k$ , потому что, например, завышенная оценка обычно лучше, чем заниженная (рис. 28.6).

	kMeansPP01Gap	kMeansPPSimpSil	kMeansPPR10SimpSil	kMed01SimpSil	kMed01R10SimpSil	Spectral
A_Rand	3.67	3.4	3.6	3.2	2.87	3.2
Устойч.	4.47	2.73	1.4	3.4	3	Н/Д
AbsKDiff	2.6	2.67	2.33	2.73	2.47	1.8

Рис. 28.6. Сравнение производительности с угаданным  $k$  (разрыв обсуждается в комментариях)

Когда  $k$  известно, спектральный анализ с  $k$ -медианным переключением для больших  $n$  является явным победителем и работает с произвольными расстояниями. Когда  $k$  неизвестно, это также первый вариант, который нужно попробовать, — хотя  $k$ -медиана лучше работает с таблицей, спектральный метод работает для определенных чистых наборов данных, таких как цифры.

Учитывая все атрибуты, кажется, что в настоящее время  $k$ -средние,  $k$ -медианы и их повторяющиеся версии можно использовать безопасно во всех задачах. Другим требуется оценка устойчивости, которая может быть слишком дорогой, если у вас нет знаний предметной области, которые могут помочь достичь хорошей производительности. Интересная идея состоит в том, чтобы рассматривать другие  $A$  только тогда, когда оценка их устойчивости выполняется достаточно быстро. Повторные  $k$ -средние почти всегда являются наиболее устойчивыми и гораздо реже лучшими, поэтому эта стратегия нуждается в дополнительных исследованиях.

## 28.10. Примечания по реализации

Учебники [28.1 и 28.13] охватывают большинство алгоритмов, которые я реализовал, но не все. Некоторые из них были доступны только из первичной литературы. Выбор алгоритма был основной трудностью как при оценке, так и при фактической кластеризации.

Реализация  $k$ -медианы является оригинальной, хотя и несколько очевидной золотой серединой между более экстремальными альтернативами большего или меньшего объема работы за итерацию. Также в этой реализации хорошо используется память и эффективно выполняется кэширование. Немало алгоритмов я испытал и отбросил, когда впервые начал исследовать тему, потому что многие из них поначалу кажутся многообещающими, но плохо работают с простыми числовыми данными, которые я считал за эталон.

## 28.11. Комментарии

Иногда в целях сбора обучающих данных для классификатора предлагается использовать кластеризацию. Идея кажется интересной, но запутанной, потому что классификатор может подобрать любой шаблон, достаточный для разделения данных, каким бы глупым этот шаблон ни был.

Скорректированный индекс Рэнда превосходит многие другие показатели, особенно те, которые также учитывают пары (см. [28.1]). Он был тщательно протестирован и использовался с момента его появления в 1985 году, и его поведение достаточно понятно. Возможно, информационные метрики, такие как *нормализованная взаимная информация*, являются единственным потенциальным конкурентом (см. [28.1, 28.13, 28.26]), но требуют дальнейшего изучения из-за некоторых проблем:

- ◆ у ненормализованной взаимной информации есть свои недостатки (см. [28.1]). Но нормализация суммы делает эту метрику эквивалентной конкретной нормализации *вариации информации*, которая, согласно источникам, хорошо работает без нормализации. Сам факт того, что проблемы метрики исправляют нормализацией, выглядит подозрительно;
- ◆ нормализация суммы имеет менее жесткую границу знаменателя, чем минимальная (см. [28.26]), но неясно, обладает ли последняя такими же хорошими свойствами;
- ◆ обе упомянутые нормализации используют минимальное значение вместо ожидаемого значения, в отличие от скорректированного индекса Рэнда, потому что, предположительно, последний трудно подсчитать (см. [28.1]). Но расчеты проводились по одной и той же гипергеометрической модели (см. [28.26]), и неясно, какая из них лучше.

Рекомендуемой мерой (см. [28.1]) является *расстояние Ван Донгена*  $= 1 - \text{среднее значение чистоты и чистоты(случайность)}^T$ . Несмотря на то что при нормализации эта метрика удовлетворяет некоторым хорошим свойствам, она теряет информацию, т. к. не обращает внимания на немаксимальные элементы случайности (см. [28.1, 28.13]).

Чистота(случайность)<sup>T</sup> может быть полезной описательной метрикой, но, похоже, она не исследована. При использовании описательных метрик, таких как чистота и ошибка классификации, интересно рассмотреть балансировку по размерам маркированных классов, хотя, в отличие от классификации, этот вопрос, похоже, не изучался. Обратите внимание, что балансировка по размеру обнаруженного кластера вместо этого не имеет смысла, поскольку случайно обнаруженные небольшие кластеры будут иметь слишком большое влияние.

Существует несколько методов оценки стабильности (см. [28.28, 28.13]). Несколько более простая и эффективная стратегия начальной загрузки заключается в сравнении



кластеризации всех данных с кластеризацией начальной загрузки, но она может иметь более высокую дисперсию, как и кластеризация со всеми данными.

Простой, но несовершенный способ измерения устойчивости — выполнение нескольких прогонов и вычисление дисперсии некоторой меры качества. Способ фиксирует только дисперсию из-за рандомизации в  $A$ .

Использование точности классификации вместо скорректированного индекса Рэнда для устойчивости имеет некоторые теоретические преимущества (см. [28.28]), но поправка на случайность кажется более важной.

Для  $k$ -средних реализация дерева VP немного медленнее, чем *алгоритм Хамерли* (самый простой из нескольких методов неравенства треугольников — см. [28.11, 28.8]), но недостаточно, чтобы оправдать сложность реализации последнего (если только у вас нет реализации дерева VP).

$k$ -медоиды, разработанные с учетом надежности (см. [28.15]), не являются устойчивыми (см. [28.13]). Но нужны эксперименты, чтобы это подтвердить и решить, полезны ли на практике устойчивые варианты, т. е. можно ли повысить производительность ценой небольшого повышения вычислительных затрат. Можно использовать  $k$ -средние, основанные на медианах (называемые  $k$ -медианами или иногда, ошибочно,  $k$ -медоидами), но есть и другие варианты (см. [28.13]).

Для  $k$ -средних было предложено и сравнено много других методов инициализации (см. [28.6]). Инициализация  $k$ -средних++ превосходит случайную инициализацию. Некоторые более сложные методы работают заметно лучше, но не гарантируют ожидаемого коэффициента аппроксимации. Кроме того, инициализация  $k$ -средних++ хорошо работает с повторением.

Существует популярный в академических кругах алгоритм, который я считаю бесполезным, — *кластеризация ЕМ*. Он пытается уменьшить ошибку аппроксимации  $k$ -средних, подбирая многомерный гауссиан для каждого кластера. Затем вероятность данных выражается как смесь гауссианов в соответствии с  $k$  вероятностями смеси. Для вычисления параметров используется *алгоритм максимизации ожидания* (Expectation Maximization, EM). Он итеративно вычисляет вероятность и параметры смеси/гауссиана до сходимости, аналогично числовой итерации с фиксированной точкой (см. [28.10, 28.1]). Некоторые опасения:

- ◆ время выполнения одной итерации равно  $O(k(n + D)D^2)$ , что является в  $O(D)$  дороже, чем у  $k$ -средних. Таким образом, даже при  $D > 100$  вычисление может оказаться невозможным, известно также, что оценка ковариационных матриц с большим  $D$  проблематична;
- ◆ исследования производительности в литературе ничего хорошего не несут. ЕМ выполняется аналогично  $k$ -средним для смоделированных данных (см. [28.20]) и речевых данных (см. [28.16]). По некоторым конкретным данным ЕМ намного лучше (см. [28.25]), но такие случаи, как правило, необычны (см. [28.21]). Плохая производительность угадывания  $k$  с помощью ВИС описывалась в работе [28.20]). Возможно, сама многомерная модель Гаусса сомнительна для практических наборов данных.

Мягкие/нечеткие  $c$ -средние (см. [28.1]) — это расширение  $k$ -средних, где членство в кластере является вероятностным, как и у ЕМ. Далее этот метод не рассматривается, потому что:

- ◆  $c$ -средние в  $O(k)$  раз медленнее, чем  $k$ -средние. Кроме того, обнаружение сходимости в этом случае сложнее, поскольку у этого метода непрерывные параметры;
- ◆ в некоторых исследованиях, где сравниваются  $c$ -средние и  $k$ -средние, сообщается о смешанных результатах. У этих исследований есть недостатки, связанные с использованием небольшого количества наборов данных, неправильных показателей сравнения, однократного прогона, смоделированных данных или случайной инициализации  $k$ -средних;
- ◆ в нечеткой кластеризации есть специализированные внутренние метрики оценки, и  $k$ -средние++ технически в этом случае не применяются.

Для спектральной кластеризации с большими разреженными матрицами используется неявно перезапускаемая итерация Ланцоша (см. главу 22. Численные алгоритмы: введение и матричная алгебра). В работе [28.27] делается предположение, что это может быть подходящим приближением, но здесь этот метод не реализован.

Нормализованный симметричный лапласиан — это не единственный вариант. Существует еще один алгоритм, у которого есть некоторые теоретические преимущества (см. [28.27]), но он гораздо менее популярен и далее не рассматривается.

Интересной статистикой при выборе  $k$  для данных векторного пространства является *разрыв* (см. [28.22, 28.12]). Подумайте, лучше ли кластеризация, чем случайные значения. Для работы требуется нулевая гипотеза, описывающая, что именно является случайным. В этом случае используются случайные данные о положении, предполагая некоторое распределение в векторном пространстве, такое как многомерная однородность в диапазоне данных. Вы должны сгенерировать и сгруппировать множество случайных выборок для получения нулевого распределения.

Таким образом, с  $k$  классами  $\text{gap}_k = E[\text{статистика кластеризации случайных данных}] - \text{статистика кластеризации данных}$ . Стандартное отклонение статистики случайных

данных также вычисляется как  $s_k \sqrt{1 + \frac{1}{B}}$ , где для вычисления ожидания выполняется

$B$  симуляций. Поправочный коэффициент кажется чисто эвристическим. Авторами в эксперименте используется значение  $B = 20$  (см. [28.12]). Значение  $k$  выбирается с помощью правила одного стандартного отклонения: выберите максимальное  $k$  так, чтобы  $\text{gap}_k > \text{gap}_{k+1} + s_{k+1}$ .

Статистика, используемая в методе разрыва, равна  $\log(W)$ , где для данных центроидов  $c_j$ :  $W = \sum_j n_j \sum_{\text{кластер}(i)=j} d(x_i, c_j)$  (см. [28.12]). Более общее определение нецентроида вычисляется за  $O(n^2)$  времени для вычисления. Функция  $\log(W)$  используется вместо  $W$  из-за ее лучшей нормальности, поэтому кажется, что вместо  $\log(W)$  может применяться любая статистика, которая распределена приблизительно нормально. Это позволяет задействовать разрыв для других случаев, таких как кластеризация ЕМ. Однако этот метод не распространяется на метрическое пространство из-за необходимости иметь возможность выборки.

Приятной особенностью разрыва является возможность сравнивать  $k = 1$  без кластеризации с  $k > 1$ . Но многомерное равномерное распределение слишком случайно, если характеристики данных сильно коррелированы, а лежащее в основе измерение коррелирует намного меньше, что часто бывает у многомерных данных с множеством признаков. Возможно, поэтому в моих тестах упрощенный силуэт работает лучше. К тому же он работает в  $B$  раз быстрее.

Для выбора  $k$  в задаче спектральной кластеризации можно использовать *эвристику собственного зазора* (см. [28.27]), где остановка алгоритма происходит тогда, когда собственные значения начинают резко расти. К сожалению, этот метод не позволяет оптимизировать алгоритм. Некоторые возможности, которые я тестировал, работают не очень хорошо (во всех случаях требуется  $k \leq \sqrt{n}$ ):

1. В нескольких онлайн-источниках предлагалось использовать наибольший зазор, но даже согласно примерам, приведенным в работе [28.27], не нужно выбирать первый собственный зазор.
2. Первый собственный зазор  $>$  следующий.
3. Первый собственный зазор  $\geq 95\%$  собственного зазора.
4. Найти  $k$ , используя пункт (1), а затем снова найти его из  $[1, k]$  с использованием индекса окончательного алгоритма кластеризации на  $k$  признаках.

В этом вопросе требуется больше исследований. Кроме того, собственный зазор является относительным индексом, т. е. полезен только для выбора  $k$ , тогда как силуэт измеряет качество назначения независимо от  $A$  или цели.

Еще один способ реализации ЕМ заключается в обеспечении положительных собственных значений ковариационной матрицы. Не исключено, что можно ограничить их диапазон, чтобы ограничить количество условий или просто потребовать определенных минимальных небольших значений. Но представленная эвристика работает хорошо.

Использование ядер для кластеризации обычно проблематично, потому что без перекрестной проверки нет хорошего способа выбрать такие параметры, как ширина по Гауссу. Кроме того, прямое использование ядер, как в *ядерном  $k$ -среднем*, приводит к проблемам из-за высокой нелинейности — оптимизация с помощью локального поиска  $k$ -средних здесь гораздо менее эффективна (см. [28.19]). Спектральные методы исправляют это, одновременно уменьшая размерность, и к тому же быстрее работают в случае разреженных матриц. В работе [28.7] рассказывается о попытках масштабировать оптимизацию.

Интересным методом является *иерархическая кластеризация*. Слияние двух групп, находящихся на самом близком расстоянии друг от друга, называется *одинарной связью*. Идея состоит в том, чтобы многократно объединять два ближайших примера (или группы примеров) до тех пор, пока не получится одна группа. Это может быть эффективно реализовано с кучей индексов, но выполняется за немасштабируемое время

$O(n^2 \lg(n))$  и требует  $O(n^2)$  памяти из-за необходимости рассматривать все  $\binom{n}{2}$  пар примеров в первом проходе.

Хотя этот метод похож на ближайшего соседа и способен обнаруживать скопления произвольной формы, он страдает от *образования цепочек*. Зашумленные примеры могут привести к раннему слиянию несвязанных кластеров (по той же логике кажется, что он не сходится в устойчивости). Таким образом, из нескольких возможностей наилучшей кажется *средняя связь*, т. е. среднее расстояние между группами. На документах среднее связывание данных работает лучше, чем другие иерархические методы, но хуже, чем  $k$ -средние (см. [28.14]). Иерархическая кластеризация, как правило, наиболее полезна для визуализации кластеризации небольших наборов данных (путем построения иерархии — см. [28.13]).

Существует и другой подход — *кластеризация на основе плотности*. Найдите соединенные плотные участки точек и объявите их кластерами. Наиболее репрезентативным и простым  $A$  является *DBSCAN* (см. [28.9]). Но мои недолгие эксперименты с ним дали плохие результаты. Возможно, дело в том, что для измерения плотности используется один глобальный  $\varepsilon$ . Использовать *DBSCAN* имеет смысл только тогда, когда плотность меняется незначительно. Учитывая, что даже в случайной выборке плотности имеют тенденцию меняться, этот подход не кажется надежным.

*Самоорганизующаяся карта* на основе нейронных сетей (Self-Organizing Map, SOM), похоже, утратила свою популярность. Этот метод также нуждается в настройке параметров (см. [28.16]).

В работе [28.1] приведены специализированные методы для конкретных типов данных. В работе [28.13] обсуждаются многие другие, менее полезные методы кластеризации, например:

- ◆ *сеточные методы* аналогичны методам на основе плотности. Они разделяют  $X$  на сетку и пытаются найти связанные ячейки. Но у этих методов экспоненциальное время выполнения по  $D$ , они сложны в реализации и не работают в метрике  $X$ ;
- ◆ *поточные методы* и методы больших данных, как правило, перепроектируются, чтобы работать в конкретных моделях. На практике чаще используется распараллеливание хорошо зарекомендовавшего себя  $A$  с такими инструментами, как Hadoop и Spark. Например, метод  $k$ -средних легко адаптируется, хотя инициализация требует некоторой осторожности (см. [28.4]);
- ◆ ансамблевые методы не применяются так же, как и для классификации, потому что они не могут формировать основной предиктор. Даже использование классификаторов на результатах кластеризации не помогает, потому что неясно, как объединить последние. Вместо этого найдите *консенсусное решение*, обычно такое, которое обладает максимальной устойчивостью по отношению к другим решениям и в каком-то смысле является их «центроидом». Для расчета стабильности найдите среднее сходство каждой базовой кластеризации по отношению ко всем остальным, что работает, потому что все базовые кластеры  $A$  содержат одни и те же данные. Можно комбинировать различные алгоритмы таким образом, что бы это было похоже на голосование большинства за классификацию, но преимущества не особенно очевидны и требуют дальнейших экспериментов.

Обширное экспериментальное сравнение методов кластеризации еще не проводилось.

## 28.12. Советы по дополнительной подготовке

- ◆ В спектральном методе проверьте, если есть разница в производительности между обычным и упрощенным силуэтом.
- ◆ В спектральном методе попытайтесь неявно перезапустить Ланцоша для вычисления собственных векторов.

## 28.13. Список рекомендуемой литературы

- 28.1. Aggarwal C. C., & Reddy C. K. (Eds). (2014). Data Clustering: Algorithms and Applications. CRC.
- 28.2. Arthur D., Manthey B., & Röglin H. (2011). Smoothed analysis of the k-means method. Journal of the ACM (JACM), 58(5), 19.

- 28.3. Arthur D., & Vassilvitskii S. (2007). k-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (pp. 1027–1035). SIAM.
- 28.4. Bahmani B., Moseley B., Vattani A., Kumar R., & Vassilvitskii S. (2012). Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7), 622–633.
- 28.5. Bouveyron C., & Brunet-Saumard C. (2014). Model-based clustering of high-dimensional data: A review. *Computational Statistics & Data Analysis*, 71, 52–78.
- 28.6. Celebi M. E., Kingravi H. A., & Vela P. A. (2013). A comparative study of efficient initialization methods for the k-means clustering algorithm. *Expert Systems with Applications*, 40(1), 200–210.
- 28.7. Chitta R. (2015). *Kernel-based Clustering of Big Bata* (Doctoral dissertation, Michigan State University).
- 28.8. Drake J. (2013). *Faster k-means Clustering*. Baylor University.
- 28.9. Ester M., Kriegel H. P., Sander J., & Xu X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd* (Vol. 96, № 34, pp. 226–231).
- 28.10. Gupta M. R., & Chen Y. (2010). *Theory and Use of the EM Algorithm*. Now Publishers.
- 28.11. Hamerly G., & Drake J. (2015). Accelerating Lloyd's algorithm for k-means clustering. In *Partitional Clustering Algorithms* (pp. 41–78). Springer.
- 28.12. Hastie T., Tibshirani R., & Friedman J. (2009). *The Elements of Statistical Learning*. Springer.
- 28.13. Hennig C., Meila M., Murtagh F., & Rocci R. (Eds). (2016). *Handbook of Cluster Analysis*. CRC.
- 28.14. Karypis M. S. G., Kumar V., & Steinbach M. (2000). A comparison of document clustering techniques. In *TextMining Workshop at KDD2000*.
- 28.15. Kauffman L., & Rousseeuw P. J. (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley.
- 28.16. Kinnunen T., Sidoroff I., Tuononen M., & Fränti P. (2011). Comparison of clustering methods: A case study of text-independent speaker modeling. *Pattern Recognition Letters*, 32(13), 1604–1617.
- 28.17. Russell S. J., Norvig P. (2020). *Artificial Intelligence: a Modern Approach*. Prentice Hall.
- 28.18. Shalev-Shwartz, S., & Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.
- 28.19. Sugiyama M. (2016). *Introduction to Statistical Machine Learning*. Morgan Kaufmann.
- 28.20. Steinley D., & Brusco M. J. (2011a). Evaluating mixture modeling for clustering: recommendations and cautions. *Psychological Methods*, 16(1), 63–78.
- 28.21. Steinley D., & Brusco M. J. (2011b). K-Means Clustering and Mixture Model Clustering: Reply to McLachlan (2011) and Vermunt (2011). *Psychological Methods*, 16(1), 89–92.
- 28.22. Tibshirani R., Walther G., & Hastie T. (2001). Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(2), 411–423.
- 28.23. Tomašev N., & Radovanović M. (2016). Clustering Evaluation in High-Dimensional Data. In *Unsupervised Learning Algorithms* (pp. 71–107). Springer.
- 28.24. Vendramin L., Campello R. J., & Hruschka E. R. (2010). Relative clustering validity criteria: A comparative overview. *Statistical Analysis and Data Mining*, 3(4), 209–235.
- 28.25. Vermunt J. K. (2011). K-means may perform as well as mixture model clustering but may also be much worse: Comment on Steinley and Brusco (2011).
- 28.26. Vinh N. X., Epps J., & Bailey J. (2010). Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *Journal of Machine Learning Research*, 11(Oct), 2837–2854.
- 28.27. von Luxburg U. (2007). A tutorial on spectral clustering. *Statistics and computing*, 17(4), 395–416.
- 28.28. von Luxburg U. (2010). *Clustering Stability*. Now Publishers.

## 29. Машинное обучение: прочие задачи

### 29.1. Введение

В этой главе кратко рассматриваются задачи машинного обучения, которые не обсуждались в предыдущих главах. Основная из них — обучение с укреплением, о котором написано много книг, но полезные практические приложения этого метода ограничены.

### 29.2. Обучение с подкреплением

Требуется найти *максимизирующую значение политику выбора шага* для агента, который перемещается в некотором пространстве состояний и получает *вознаграждение* (состояние) при входе в состояние. *Значение* состояния =  $E[\text{вознаграждение} + \text{значение состояния, в которое агент перейдет дальше}] = E[\sum \text{вознаграждение за выбранную последовательность состояний, начиная с некоторого состояния}]$ . *Эпизодом* называется последовательность, которая приводит к терминальному состоянию. Вознаграждения *обесцениваются*, так что вознаграждение  $r$  на расстоянии в  $k$  состояний стоит  $ry^k$  для  $0 < y \leq 1$ . Для неэпизодических пространств состояний  $y < 1$  предотвращает  $\sum \text{вознаграждения} \rightarrow \infty$ . Это позволяет смоделировать множество проблем:

- ♦ игры вроде шахмат — любая последовательность ходов приводит к мату или отсутствию материального обмена на 50 ходов. То же самое и в блэкджеке — принятие решения о том, брать другую карту или нет, учитывая сыгранные карты. Обычно  $r$  равно 1 для победы,  $1/2$  для ничьей и  $-1$  — для поражения. Оптимальная политика максимизации вознаграждения определяет идеального игрока;
- ♦ управление сложными системами — такими как автопилот. Шаги определяют возможные настройки навигации и других элементов управления. Конечного состояния не существует, и вознаграждения являются сигналами опасности и качества исполнения;
- ♦ *задача многорукого бандита*. Бандит — это игровой автомат казино с множеством рычагов, каждый из которых имеет неизвестное  $E[\text{выплата}]$ . Вознаграждение  $r$  — это выигрыш после нажатия на рычаг;
- ♦ распределение ресурсов. Предположим, некто, инвестирующий в облигации, постоянно выбирает между денежными средствами и облигациями с различными процентными ставками, сроками погашения и рисками дефолта после каждого события, связанного с получением дохода.

Моделирование является самой простой и наиболее общей стратегией решения. Учитывая текущее состояние, из каждого следующего состояния сгенерируйте множество случайных эпизодов и выберите состояние, максимизирующее среднее вознаграждение

за последовательность  $\sum$ . Для неэпизодических задач выполняется фиксированное количество шагов или столько, пока  $r$  не станет слишком маленьким.

Для повышения эффективности вы можете выполнить больше моделирований многообещающих ходов на основе первоначальных симуляций. Критерий UCB1 после  $n$  моделирований моделирует шаг:

$$i = \arg \max_i \left( \text{ave}(i) + \sqrt{\frac{2 \log(n)}{n(i)}} \right),$$

где  $r$  принадлежит  $[0,1]$ , а шаг  $i$  был смоделирован  $n(i)$  раз. Для наград с разным масштабом константа  $\neq 2$ , поэтому сначала выполняется масштабирование. Любой выбор гарантирует асимптотически оптимальное значение  $E[\text{проигрыш от невыполнения лучшего хода, обнаруженного после } n \text{ попыток}]$  для любого  $n$  (см. [29.7]):

```
double UCB1(double averageValue, int nTries, int totalTries)
{ return averageValue + sqrt(2 * log(totalTries)/nTries); }
```

Применение UCB1 более чем к одному шагу не влияет на оптимальность (см. [29.8]), поэтому после эпизода, если у вас достаточно памяти, нужно сохранить и обновить средние значения и количество посещений. Вы можете сбросить любую сохраненную информацию после совершения хода. Отслеживание максимальных/минимальных значений позволяет выполнять динамическое масштабирование, но это только приближительное значение, поскольку при текущем масштабе предварительное решение не обязательно должно быть оптимальным, если оно масштабируется по-разному.

Если выполнить масштабирование до  $[0, 1]$  невозможно, используйте ОСВА (см. главу 21. *Вычислительная статистика*) — в отличие от CLT, масштабирование здесь не требуется.

## 29.3. Функция, присваивающая значения

Еще один масштабируемый подход заключается в создании функции, которая присваивает значение каждому состоянию и переходит в состояние с максимальным значением. Значение (текущее состояние) = вознаграждение (текущее состояние) + среднее наблюдаемое значение (следующее состояние)  $V_{\text{следующее}}$ . Инкрементальное среднее обновление  $\text{ave}_n = \text{ave}_{n-1} + \frac{1}{n}(x_n - \text{ave}_{n-1})$  дает  $V_{\text{текущее}} += \frac{1}{n}(r + V_{\text{следующее}} - V_{\text{текущее}})$ . Алгоритм устанавливает все  $V = 0$  и многократно запускает эпизод с начального состояния, выбирая следующее состояние за счет текущего  $V$  и правил обновления. Доказуемо, что алгоритм сходится к значениям, удовлетворяющим  $V_{\text{текущее}} = r + E[V_{\text{следующее}}]$  для исследованных состояний (см. [29.6]). Изменение скорости обучения  $1/n$  на более медленный рост  $n^{-0.501}$  может привести к более быстрой сходимости. Если сделать начальные значения оптимистичными/пессимистичными, это стимулирует исследование/эксплуатацию и сходится быстрее, если подсказка полезна. Сходящиеся значения не обязательно должны быть оптимальными, если следующий выбор состояния является жадным и ничего не исследует. Использование UCB1 или запуск эпизодов из случайных состояний обеспечивает сходимость к оптимальным значениям:

```
template<typename PROBLEM> void TDLearning(PROBLEM& p)
{
    while(p.hasMoreEpisodes())
```

```

{
    double valueCurrent = p.startEpisode();
    while(!p.isInFinalState())
    {
        double valueNext = p.pickNextState();
        p.updateCurrentStateValue(p.learningRate() * (p.reward() +
            p.discountRate() * valueNext - valueCurrent));
        p.goToNextState();
        valueCurrent = valueNext;
    }
    p.updateCurrentStateValue(p.learningRate() *
        (p.reward() - valueCurrent));
}
}

```

В небольших пространствах состояний  $V$  является дискретным и запоминает значение каждого состояния:

```

struct DiscreteValueFunction
{
    Vector<pair<double, int> > values;
    double learningRate(int state){return 1.0/values[state].second;}
    void updateValue(int state, double delta)
    {
        ++values[state].second;
        values[state].first += delta;
    }
    DiscreteValueFunction(int n): values(n, n, make_pair(0.0, 1)){}
};

```

В огромных пространствах состояний, таких как шахматы,  $V$  является приближением. Доказано, что линейная комбинация характеристик состояния сходится к наилучшему возможному линейному приближению для скорости Роббинса — Манро (см. [29.6]). Другие представления, такие как нейронная сеть, могут давать лучшие результаты, но могут не сходиться:

```

struct LinearCombinationValueFunction
{
    Vector<double> weights;
    int n;
    double learningRate(){return 1.0/n;}
    void updateWeights(Vector<double> const& stateFeatures, double delta)
    { // установка одного из признаков состояний равным 1,
      // чтобы получить вес отклонения
      assert(stateFeatures.getSize() == weights.getSize());
      for(int i = 0; i < weights.getSize(); ++i)
          weights[i] += delta * stateFeatures[i];
      ++n;
    }
    LinearCombinationValueFunction(int theN): weights(theN, theN, 0), n(1) {}
};

```

Обучение функцией значения аналогично выполнению регрессии, но требуется политика для присвоения конечного значения промежуточным состояниям и обучения



в реальном времени для повышения эффективности. Ознакомьтесь с работой [29.11]. О применении к играм можно почитать в работе [29.9]. По неизвестной причине определенные игры — такие как нарды, больше подходят для этого, чем некоторые другие — такие как шахматы. В последние годы впечатляющие успехи игровых программ AlphaGo и AlphaZero возродили интерес к обучению с подкреплением, особенно когда для присвоения значений состояния используется глубокая нейронная сеть.

## 29.4. Поиск часто встречающихся комбинаций предметов

Существуют системы рекомендаций — типа как в разделе Amazon «с этим также покупают». В базе данных такая покупка сохраняется, если это происходит в других корзинах достаточное количество раз.

*Алгоритм Апериори* позволяет избежать комбинаторного взрыва, сортируя элементы в корзине по id и добавляя комбинацию, если  $k = 1$  или последний элемент и комбинация из  $k - 1$  первых предметов достаточно часты. Алгоритм обрабатывает корзины по раундам, при этом раунд  $k$  учитывает  $k$  комбинаций. Сортировка гарантирует, что комбинации из  $k - 1$  элементов обрабатываются раньше комбинаций из  $k$ :

```
struct APriori
{
    LCPTreap<Vector<int>, int> counts;
    void noCutProcess (Vector<Vector<int> >const& baskets, int nRounds)
    {
        for(int k = 1; k <= nRounds; ++k)
            for(int i = 0; i < baskets.getSize(); ++i)
                processBasket(baskets[i], k);
    }
};
```

Обработка корзины возвращает количество добавленных товаров, чтобы вызывающий код мог остановиться, если в текущем раунде ничего не добавлено, или отрегулировать отсечки:

```
int processBasket(Vector<int> const& basket, int round,
    int rPrevMinCount = 0, int r1MinCount = 0)
{
    int addedCount = 0;
    if(basket.getSize() > round)
    {
        Combinator c(round, basket.getSize());
        do // подготовка текущей комбинации, сортировка не нужна
        { // корзина уже отсортирована
            Vector<int> key, single;
            for(int i = 0; i < round; ++i) key.append(basket[c.c[i]]);
            quickSort(key.getArray(), key.getSize());
            int* count = counts.find(key);
            if(count) ++*count; // комбинация встречается часто
            else if(round == 1) // если round = 1
```

```

    {
        counts.insert(key, 1);
        ++addedCount;
    }
    else // комбинация встречается часто,
         // если последний элемент
    { // и комбинация без него встречаются часто
        single.append(key.lastItem());
        if(*counts.find(single) >= r1MinCount)
        {
            key.removeLast();
            if(*counts.find(key) >= rPrevMinCount)
            {
                key.append(single[0]);
                counts.insert(key, 1);
                ++addedCount;
            }
        }
    }
}
}while(!c.next());
}
return addedCount;
}

```

## 29.5. Полуконтролируемое обучение

Нам требуется классифицировать использование как помеченных, так и непомеченных образцов, потому что для маркировки обычно требуется мнение человека. Маркированные наборы данных почти никогда не бывают слишком большими, если только некоторые данные не помечены автоматическим источником. Но немаркированных данных предостаточно.

Немаркированные образцы позволяют лучше оценить свойства распределения на  $X$ , т. к. в LDA есть ковариационная матрица без использования меток, поэтому немаркированные образцы помогают. Но, как правило, это плохой классификатор даже с хорошей оценкой ковариационной матрицы. Уменьшение размера PCA тоже работает без необходимости маркировки.

Также вы можете кластеризовать все образцы и использовать классификатор, обученный на размеченных данных, чтобы классифицировать их и обучить окончательный классификатор на результате. Этот метод может работать хорошо, но не тогда, когда кластеры не соответствуют классам.

Существует несколько нерешенных проблем:

- ◆ трудно выбрать модель, потому что она основывает решение в основном на помеченных образцах;
- ◆ эксперименты показывают, что не существует хорошего метода «черного ящика» (см. [29.3]), в отличие от обучения с учителем. Также выбор алгоритма требует понимания данных.

## 29.6. Оценка плотности

Требуется распределение вероятностей по известной выборке. Задача не вполне корректна в том смысле, что  $\Pr(x) = (x \in \text{data} ? 1/n : 0)$ , поэтому мы должны назначать вероятность  $x \notin \text{данным}$ , и для этого в лучшем случае можно использовать что-то вроде ORM.

Несколько алгоритмов хорошо показали себя для небольших  $D$  (обычно  $\leq 3$  (см. [29.10])):

- ♦ *гистограмма* — разделить  $X$  на сетку, а для любой ячейки оценить ее вероятность по доле содержащихся примеров. Используйте  $o(n)$  ячеек, чтобы сбалансировать ошибки приближения и оценки, и начните с 1 для сглаживания. Можно рассмотреть возможность использования дерева  $k$ -d или VP для расширения до больших  $D$  за счет адаптивного разделения пространства. Затем, чтобы оценить  $\Pr(x)$ , можно пройти по дереву и усреднить примерные пропорции текущего узла;
- ♦ *оценщик плотности ядра* (Kernel Density Estimator, KDE) — выразить PDF как сумму функций ядра, аналогично SVM. Обычно используется ядро Гаусса.

Если вы не знаете истинной плотности, которую нужно оценить, не существует хорошей функции потерь  $L$ , которую можно оценить. Поэтому оценки сравнивать трудно, даже по параметрам выбора.

С учетом того, как будет использоваться оценка плотности, решение этой проблемы напрямую лучше из-за меньшей ошибки оценки и менее некорректной постановки. Например, оВ сводит классификацию к оценке плотности, но на практике решает прямую задачу классификации. Теоретические аспекты оценки плотности обсуждаются в работе [29.4].

Эмпирическое распределение передискретизации начальной загрузки обычно работает лучше для любой задачи, которую необходимо выполнить с оценкой плотности, поэтому сначала стоит рассмотреть этот вариант.

Интересной практической задачей является оценка максимальной скорости запросов пользователей, скажем, к сайту или другому сервису. Например, в задаче торговли акциями можно выбрать время открытия. Необходимо будет как оценивать, так и максимизировать. Также могут потребоваться доверительные интервалы. Оценка плотности с равночастотными корзинами для гистограммы является одним из решений.

## 29.7. Обнаружение выпадающих значений

*Выпадающее значение* — это выборка, созданная механизмом, отличным от механизма создания остальных данных, и значение в этом случае отличается. Их обнаружение очень полезно с экономической точки зрения — любая мошенническая деятельность, например с кредитными картами, является выпадающей по сравнению с обычными транзакциями. Но проблема некорректна, потому что в ней нарушается допущение о независимости и случайности, а мы хотим обнаружить нарушения. Такие задачи обычно лучше решаются классификацией, возможно, с несбалансированными данными.

Классическое эмпирическое правило заключается в том, что наблюдения с  $\geq 3$  стандартными отклонениями от среднего являются выпадающими (см. [29.1]). Эвристи-

чески это работает только для нормально распределенных данных, а число «3» здесь несколько произвольно. Оно, видимо, переключало из ранних методов управления промышленными процессами, таких как контрольные карты. Более свободный от предположений подход состоит в том, чтобы рассмотреть распределение соответствующей статистики порядка из распределения базовой модели и сделать вывод о том, что значение является выпадающим, если оно более экстремально, чем 99,9% значений, которые могут быть сгенерированы. Но это также условно, и использование правила на многих выборках подлежит многократной проверке. Таким образом, современные методы рассматривают обнаружение выбросов как классификацию на классы «нормальное» и «выпадающее».

Невозможно обнаружить выпадающее значение без знаний предметной области в конкретной предполагаемой модели. Другая точка зрения состоит в том, что любая «небольшая» группа примеров, оказывающая слишком большое влияние на модель, является исключением. Так что в лучшем случае можно определить только потенциальные выпадающие значения. Несколько более сложных методов:

- ◆ использовать смешанную модель Гаусса для кластеризации данных, а затем объявить маловероятные примеры выпадающими;
- ◆ использовать DBSCAN, чтобы найти выпадающие значения;
- ◆ оценить плотность и объявить выборки в регионах с низкой плотностью;
- ◆ использовать обратного ближайшего соседа — для каждой точки отметить  $k$  ближайших соседей и считать выбросами любые неотмеченные или мало отмеченные точки.

Если цель обнаружения выбросов состоит в том, чтобы сделать оценки более устойчивыми к неверным данным, лучше использовать надежный метод.

Анализ выпадающих значений лучше всего использовать в качестве предварительного задания. В конечном итоге, особенно при попытке оценить эффективность обнаружения, вы получите размеченные данные и превратите проблему в классификацию, возможно, с несбалансированными данными. Это позволит более эффективно делать выводы в предметной области.

## 29.8. Примечания по реализации

Все реализации очень точно следуют учебникам и очень полны в деталях без каких-либо существенных усилий. Алгоритмы в основном просты, но несколько принятых решений не описаны. Но представлено не так много алгоритмов, а не описанные алгоритмы намного сложнее.

## 29.9. Комментарии

У кратко рассмотренных здесь методов нет богатства общедоступных наборов данных, в отличие от основных задач машинного обучения. Но ситуация в этом вопросе улучшается, особенно для обучения с усилением.

Обучение с усилением — это большая область знаний, по которой написано несколько учебников, но из описанного тут полезны только моделирование и обучение на основе

функции ценности. Другие типы алгоритмов основаны на политике, т. е. оценивают действие не только по состоянию, к которому оно приводит, — см. работу [29.11], если любопытно. Учитывая мой шахматный опыт, могу ответственно заявить, что такие алгоритмы бесплодные и ведут к лишнему ненужному комбинаторному взрыву. Идея разделения вознаграждения за конечную цель между всеми действиями является основным компонентом склонности к усилению и хорошо отражена в функциях ценности. Хорошая функция значения — ключ к тому, чтобы все работало хорошо.

Для теоретического понимания обучения очень важно предположение о независимости и случайности. Такие методы, как *активное обучение* (алгоритм  $A$  учится в реальном времени и выбирает следующий пример), нарушают его и тем самым ослабляют теоретические гарантии. В активном обучении используется набор немаркированных примеров, которые помечаются по одному, и поиск следующего лучшего может быть неэффективным. Сложно реализовать простую генерацию примеров, т. к. они не должны соответствовать чему-либо допустимому, т. е. изображение не обязательно должно быть цифрой. Активное обучение  $\neq$  планирование экспериментов, потому что в последнем случае выбор всех параметров допустим или имеет четкие ограничения по диапазону.

Чтобы узнать больше о поиске частых комбинаций предметов, ознакомьтесь с работой [29.2]. Эта задача имеет коммерческую ценность, но не так популярна, как другие. У существующих доступных наборов данных из-за неконтролируемого характера и отсутствия теории оценки трудно проверить качество алгоритмов.

*Трансферное обучение* — это еще одна проблемная модель. Идея состоит в том, что примеры из одной задачи могут помочь в изучении другой задачи. Так у людей, но научить компьютер этому сложнее.

## 29.10. Список рекомендуемой литературы

- 29.1. Aggarwal C. C. (2017). Outlier Analysis. Springer.
- 29.2. Aggarwal C. C., & Han J. (Eds). (2014). Frequent Pattern Mining. Springer.
- 29.3. Chapelle O., Schölkopf B., Zien A. (2006). Semi-supervised Learning. MIT Press.
- 29.4. Devroye L. & Lugosi G. (2001). Combinatorial Methods in Density Estimation. Springer.
- 29.5. Hastie T., Tibshirani R., & Friedman J. (2009). The Elements of Statistical Learning. Springer.
- 29.6. Russell S. J., Norvig P. (2020). Artificial Intelligence: a Modern Approach. Prentice Hall.
- 29.7. Auer P., Cesa-Bianchi N., & Fischer P. (2002). Finite-time analysis of the multiarmed bandit problem. Machine Learning, 47(2–3), 235–256.
- 29.8. Kocsis L., & Szepesvári C. (2006). Bandit based Monte Carlo planning. In Machine Learning: ECML 2006 (pp. 282–293). Springer.
- 29.9. Mandziuk J. (2010). Knowledge-Free and Learning-Based Methods in Intelligent Game Playing. Springer.
- 29.10. Scott D. W. (2015). Multivariate Density Estimation: Theory, Practice, and Visualization. Wiley.
- 29.11. Sutton R. S., & Barto A. G. (2018). Reinforcement Learning: An Introduction. MIT Press.

## 30. Отстойник: не слишком полезные алгоритмы и структуры данных

### 30.1. Введение

Существуют алгоритмы и структуры данных, которые не кажутся полезными, но слишком хороши, чтобы их выбрасывать, несмотря на то, что тому есть веские причины. Они либо хорошо обоснованы теоретически, либо проходятся в стандартной учебной программе, либо просто удовлетворяют определенное любопытство. Я создал большое количество таких экспериментальных реализаций и хочу поделиться ими с читателями, которые могут потом провести свои собственные исследования. Для каждой реализации будут приведены комментарии, ссылки на работы и причины, почему алгоритм плох.

С учетом сказанного, еще раз отметим, что эти реализации не поддерживаются! У них, скорее всего, нет хороших наборов тестов, описаний функциональности (даже с точки зрения дальнейших ссылок), продуманной структуры, надежности или ссылок. Особенно это касается тем предыдущих глав, потому что я так и не отшлифовал материал настолько, чтобы опубликовать его в первом черновом издании. Все, что вы увидите здесь, взято из моих резервных копий. Однако в этой главе представлено не все, что у меня есть, — лишь те реализации, которые прошли базовые тесты и достаточно интересны. Для алгоритмов, не имеющих приличных описаний, код приведен только на веб-сайте. Вы можете самостоятельно исследовать алгоритмы, представленные (и не представленные) ссылками для получения дополнительной информации. Наконец, здесь рассмотрены только те реализации, которые полезны с учетом ссылок, — например, несколько реализаций, не прошедших базовые тесты, были убраны.

### 30.2. Сортировка связанного списка

Базовый связанный список:

```
template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM> >
struct LinkedList
{
    struct Node
    {
        ITEM item;
        Node* next;
        Node(ITEM const& theItem, Node* theNext)
            : item(theItem), next(theNext) {}
    }*root;
    int size;
    Freelist<Node> freelist;
```

```

COMPARATOR c;
LinkedList(COMPATOR theC = COMPATOR()): root(0), size(0), c(theC) {}
void prepend(ITEM const& item)
{
    root = new(freelist.allocate())Node(item, root);
    ++size;
}
};

```

Сортировка слиянием — лучший алгоритм для списков, потому что он не требует произвольного доступа и позволяет манипулировать указателями списков без временного массива. Полезной оптимизацией является использование размера списка вместо вычисления середины (учитывая, что один указатель работает в два раза быстрее, чем другой, и гарантируя, что медленный указатель находится в середине, когда быстрый достигает конца), а это дает короткий эффективный алгоритм. Следующие функции являются членами списка.

Объединение списков похоже на объединение массивов:

```

Node* advanceSmaller(Node* a, Node* b)
{
    Node* smaller = c(b->item, a->item) ? b : a;
    Node* nodeToAppend = smaller;
    smaller = smaller->next;
    return nodeToAppend;
}

Node* merge(Node* a, Node* b)
{ // выбор головы
    Node* head = advanceSmaller(a, b), *tail = head;
    // добавление из наименьшего, пока он не иссякнет
    while(a && b) tail = tail->next = advanceSmaller(a, b);
    // добавление оставшегося из списка
    tail->next = a ? a : b;
    return head;
}

```

Идея вычисления местоположения середины без произвольного доступа состоит в том, чтобы запомнить указатель на начало нужного массива и обновить его во время рекурсивных вызовов:

```

Node* mergesort(Node* list, int n, Node*& nextAfterLast)
{
    if(n==1)
    {
        nextAfterLast = list->next;
        list->next = 0;
        return list;
    }
    int middle = n/2;
    Node *secondHalf, *m1 = mergesort(list, middle, secondHalf);
    return merge(m1, mergesort(secondHalf, n - middle, nextAfterLast));
}

```

```

void sort()
{
    if(size > 1)
    {
        Node* dummy;
        root = mergesort(root, size, dummy);
    }
}

```

### 30.3. Частичная сортировка

Чтобы выполнить инкрементную сортировку, сортируйте следующие элементы только при необходимости и повторно используйте правильные связанные значения из предыдущих вызовов, хранящихся в стеке. Алгоритм работает за оптимальное ожидаемое время  $O(n + k \lg(k))$  после выбора  $k$  элементов (см. [30.14]) и работает, потому что выбор осуществляется слева направо, вследствие чего быстрый выбор всегда выполняет присвоение  $\text{right} = j$ . Перед первым вызовом стек должен содержать крайний правый индекс массива, а после каждого вызова выталкивать его, чтобы обеспечить соблюдение  $\text{left} + 1 \leq \text{right}$  для следующего вызова. Массив сортируется, когда стек пуст и  $\text{left} \geq \text{right}$ . Невозможно изменить стек и вектор между вызовами:

```

template<typename ITEM, typename COMPARATOR> ITEM incrementalQuickSelect(
    ITEM* vector, int left, Stack<int>& s, COMPARATOR comparator)
{
    for(int right, i, j; left < (right = s.getTop()); s.push(j))
        partition3(vector, left, right, i, j, comparator);
    s.pop();
    return vector[left];
}

```

Маловероятно худший случай равен  $O(n^2)$ . Чтобы отсортировать список полностью с помощью частичной сортировки:

```

template<typename ITEM, typename COMPARATOR> void incrementalSort(
    ITEM* vector, int n, COMPARATOR const& c)
{
    Stack<int> s;
    s.push(n - 1);
    for(int i = 0; i < n; ++i) incrementalQuickSelect(vector, i, s, c);
}

```

Затруднительно придумать хороший вариант использования для этого алгоритма, потому что динамически отсортированные последовательности предлагают лучшую функциональность, а обычная сортировка более эффективна для статических данных. Но с концептуальной точки зрения это интересная идея, открытие которой, впрочем, заняло немало времени.

### 30.4. Сжатое префиксное дерево

Некоторые деревья работают только с ключами, которые представляют собой последовательности битов, а не сопоставимые объекты. Они полезны концептуально, но не на



практике, за исключением применений в аппаратных средствах, поскольку извлечение битов происходит относительно медленно, а операции сопоставления используют  $w$  из них, где  $w$  — размер слова. Кроме того, для использования требуется интерпретировать объекты как битовые последовательности, что неудобно для типов, отличных от слов, и не допускает портативность из-за порядка следования байтов. Таким образом, концептуально несущественные операции, такие как конструкторы копирования и итераторы, для таких последовательностей не предусмотрены.

Ключи легко интерпретировать как последовательности байтов, но это не портативно:

```
template<typename KEY> struct DefaultRank
{
    unsigned char* array;
    int size;
    DefaultRank(KEY const& key)
    { // допустимы лишь простые структуры данных
        array = (unsigned char*)&key;
        size = sizeof(key);
    }
};
```

В архитектуре с прямым порядком байтов самый младший значащий байт помещается в позицию 0 массива, чего вы, возможно, не ожидали. Для операций сопоставления это приводит к ускорению, если младшие байты различаются сильнее, чем старшие, но результирующая битовая строка не отсортирована в лексикографическом порядке, что препятствует поддержке префиксных операций. Портативное ранжирование (т. е. преобразование в битовую последовательность) будет использовать деление для извлечения байтов в обратном порядке, но для каждого типа ключа потребуется собственная реализация.

В каждом узле есть индекс следующего битового блока, который надо проверить. Реализация немного сложна, особенно для удаления. Ни один элемент не может быть префиксом другого элемента, т. к. все элементы имеют одинаковый размер  $w$ . Поэтому инкрементный поиск не имеет смысла.

Все разветвления в узле имеют один и тот же битовый путь до этого узла. Это следует из алгоритма вставки. Элементы со смысловой точки зрения не связаны с внутренними узлами. Но в операциях вроде удаления у них есть важное свойство, заключающееся в том, что обратные указатели на них расположены в их поддеревьях, а  $\text{lcp}$  с каждым дочерним узлом является как минимум индексом узла, в котором они хранятся.

Подумайте о случае, когда ключи являются целыми числами и каждое из них используется (а их всего  $2^{32}$ ). Будет два ключа, совпадающих по 31-му биту и различающихся по 32-му. Узлы, которые различаются между собой, будут разветвляться на 32-м бите, как и дочерний узел.

Дополнительное пространство составляет три слова на ключ, как в декартовом дереве, и на два слова меньше, чем для LCPTreap:

```
template<typename KEY, typename ITEM, typename RANK = DefaultRank<KEY> >
class PatriciaTrie
{
    enum{B = 8, TABLE_SIZE = 2, S = B-1};
```

```

int extract(unsigned char* key, int i)
{
    return (unsigned char) (key[i/B] << (i%B)) >> S;
}

struct Node
{
    KEY key;
    ITEM item;
    int index;
    Node *next[2];
    Node(KEY const& theKey, ITEM const& theItem, int theTo, bool bit,
        Node* other): key(theKey), item(theItem), index(theTo)
    {
        next[!bit] = other;
        next[bit] = this;
    }
}

Node* root;
Freelist<Node> freelist;

template<typename ACTION>
void forEachHelper(Node* t, ACTION& action)
{
    for(int i = 0; t && i < 2; ++i)
    {
        Node* child = t->next[i];
        if(child)
        {
            if(t->index >= child->index) action(child);
            else forEachHelper(child, action);
        }
    }
}

struct AppendAction
{
    Vector<Node*>& result;
    AppendAction(Vector<Node*>& theResult):result(theResult){}
    void operator() (Node* node){result.append(node);}
};

Node** findForwardPointer(Node* query, Node** tree)
{
    RANK rank(query->key);
    Node* node;
    while((node = *tree) != query)
    {
        tree = &node->next[extract(rank.array, node->index)];
    }
    return tree;
}

Node** findBackwardPointer(Node* query, Node** tree)
{
    RANK rank(query->key);
    int prevIndex = -1;
    Node** pointer = tree;
    Node* node;

```

```

    while((node = *pointer) && prevIndex < node->index)
    {
        prevIndex = node->index;
        pointer = &node->next[extract(rank.array, prevIndex)];
    }
    return pointer;
}

void removeSingleChildNode(Node** forwardPointer)
{
    Node* node = *forwardPointer;
    *forwardPointer = node->next[!node->next[0]];
    freelist.remove(node);
}

public:
    PatriciaTrie(): root(0){}
    // перебор узлов в лексикографическом порядке битов
    template<typename ACTION> void forEach(ACTION& action)
    {forEachHelper(root, action);}
    Node* findLongestMatch(KEY const& key)
    { // алгоритм не сработает, если ключ является префиксом другого
        if(!root) return 0;
        RANK rank(key);
        int prevIndex = -1;
        Node* node = root;
        while(prevIndex < node->index)
        {
            prevIndex = node->index;
            Node* next = node->next[extract(rank.array, prevIndex)];
            if(!next) break;
            node = next;
        }
        return node;
    }
    /*
    Самые длинные значения совпадают, потому что все узлы непроверенных битов на пути
    имеют одинаковые биты, и будут совпадать и дальше. Другие узлы не могут быть
    исключены непроверенными битами, потому что они такие же, т. к. их биты
    не проверены, или проверенными битами, потому что в этом случае поиск пойдет
    по другому пути
    */
}

ITEM* find(KEY const& key)
{
    Node* node = findLongestMatch(key);
    return node && key == node->key ? &node->item : 0;
}

void insert(KEY const& key, ITEM const& item)
{ // 1. Найти индекс ключа, который нужно вставить
    Node* lcpNode = findLongestMatch(key);
    RANK rank(key);
    int theIndex = 0;
    if(lcpNode)
    {
        if(key == lcpNode->key){lcpNode->item = item; return;}
    }
}

```

```

    RANK rank2(lcpNode->key);
    while(extract(rank.array, theIndex) ==
           extract(rank2.array, theIndex)) ++theIndex;
    if(theIndex == lcpNode->index && theIndex < rank.size * B)
        ++theIndex;
} // 2. Создать и вставить узел
Node **pointer = &root, *node;
int prevIndex = -1;
// новый узел идет перед существующим,
// если node->index >= theIndex, или
// самоуказывающий узел, если prevIndex >= node->index
while((node = *pointer) && node->index < theIndex &&
       prevIndex < node->index)
{
    prevIndex = node->index;
    pointer = &node->next[extract(rank.array, prevIndex)];
}
*pointer = new(freelist.allocate())
    Node(key, item, theIndex, extract(rank.array, theIndex), node);
}
Vector<Node*> prefixFind(KEY const& key, int minLCP)
{
    RANK rank(key);
    int prevIndex = -1;
    Node* node = root;
    while(node && node->index < minLCP && prevIndex < node->index)
    {
        prevIndex = node->index;
        node = node->next[extract(rank.array, prevIndex)];
    }
    Vector<Node*> result;
    if(node && node->index >= minLCP - 1) // && findlcp - отлично
    {
        if(prevIndex >= node->index) result.append(node);
        else
        {
            AppendAction action(result);
            forEachHelper(node, action);
        }
    }
    return result;
}
void remove(KEY const& key)
{
    RANK rank(key);
    int prevIndex = -1;
    Node** pointer = &root, **parentForwardPointer = &root;
    Node* node, *parent = 0;
    while((node = *pointer) && prevIndex < node->index)
    {
        prevIndex = node->index;
        parentForwardPointer = pointer;
    }
}

```

```

    pointer = &node->next[extract(rank.array, prevIndex)];
}
if(node && key == node->key)
{
    Node* parent = *parentForwardPointer;
    *pointer = 0;
    // если узел указывает на себя, удалите и свяжите прямой
    // указатель с другим потомком
    if(node == parent)
        removeSingleChildNode(findForwardPointer(node, &root));
    else
    { // если элемент не заменяется родительским,
      // родительский указатель переадресовывается на него,
      // и родитель удаляется как один дочерний узел
        node->key = parent->key;
        node->item = parent->item;
        *findBackwardPointer(parent, parentForwardPointer) = node;
        removeSingleChildNode(parentForwardPointer);
    }
}
}
};

```

Я поэкспериментировал с расширением сжатого префиксного дерева (Patricia trie), пытаясь научиться разделять его на несколько битов за раз для повышения эффективности (рис. 30.1). Следующая затем специальная нулевая ссылка позволяет снять требование префикса. Но это расширение занимает больше места для узлов и вынуждает отказаться от удаления, потому что замена удаленного узла узлом, который указывает на него, может привести к тому, что листовый узел будет иметь два обратных указателя на узлы, отличные от него самого, а это означает, что этот лист не может быть удален. Это происходит, когда узел удаляется и заменяется другим узлом с прямым указателем, превращающимся в обратный указатель.

Поддерживаются только слабые удаления (см. главу 12. *Разные алгоритмы и методы*) путем создания узлов по мере удаления и периодического восстановления. В качестве альтернативы можно хранить элементы во внешних узлах, что позволяет удалять их, но это делает реализацию более громоздкой и менее эффективной.

	LLRBT	Treap	CHT	Patricia	LCPTreap
int	18,4 (37/36)	16(38/36)	8,0 (33/32)	34(37/36)	20(43/42)
double					23(61/60)
Struct10	54(90/90)	72(90/90)	49(86/85)	45(90/90)	26(96/96)
Struct10_2	19(90/90)	21(90/90)	49(86/85)		26(96/96)
Struct10_4				54(90/90)	44(96/96)
Struct10_5					145(96/96)

**Рис. 30.1.** Некоторые сравнения производительности LLRBT, сжатого префиксного дерева и лучших структур данных (время, память<sub>1</sub>, память<sub>2</sub>)

## 30.5. Хеширование кукушкой

Поиск и удаление в худшем случае выполняются за  $O(1)$ , а вставка ожидается за  $O(1)$ . Прежде чем сильно радоваться, обратите внимание, что хеширование кукушкой опасно, потому что ожидаемая вставка  $O(1)$  может оказаться бесконечным циклом, если вы плохо выберите хеш-функцию. Даже универсальная  $h$  может привести к проблемам, и чтобы быть теоретически безопасными, достаточно  $\lg(n)$ -независимых хешей (см. [30.6]). Так что этот метод хеширования хорош для тестирования  $h$ , но не для общего использования.

Хеш-таблица кукушки состоит из двух массивов элементов, хешей и массивов состояний. Вставка помещает элемент  $x$  в ячейку  $h_1(x)$  таблицы<sub>1</sub>, и если эта ячейка занята, элемент  $y$  внутри нее выбрасывается в ячейку  $h_2(x)$  таблицы<sub>2</sub>, далее, возможно, элемент  $z$  выбрасывается в ячейку  $h_1(z)$  таблицы<sub>1</sub> и т. д., пока элемент не будет помещен в пустую ячейку, или это не произойдет  $k$  раз (часто используется значение  $k = 50$ , на практике требуется  $O(\lg(n))$ ), после чего все элементы перефразируются с использованием другой пары случайно выбранных хешей. Требуется выполнение условия  $a < 0,5$  для ожидаемой производительности  $O(1)$ . Поиском и удалением нужно проверить только ячейки  $h_1(x)$  таблицы<sub>1</sub> и ячейку  $h_2(x)$  таблицы<sub>2</sub>, чтобы найти  $x$  или сделать вывод, что его там нет.

В качестве меры оптимизации можно использовать одну таблицу и проверить, не пуста ли ячейка перед первым удалением во время первой вставки. В работе [30.11] описывается и анализируется несколько вариантов. Хеширование кукушкой имеет лишь одно практическое преимущество: оно адаптируется к данным в том смысле, что вынуждает выбрать хорошую функцию  $h$ , и работает за ожидаемое время  $O(1)$ . Из-за необходимости небольшого размера требуется больше памяти, чем для цепного или линейного зондирования:

```
template<typename KEY, typename VALUE, typename HASHER = EHash<BUHash> >
class CuckooHashTable
{
    enum{NOT_FOUND = -1};
    int capacity, size;
    typedef KVPair<KEY, VALUE> Node;
    Node* table;
    bool* isOccupied;
    HASHER h1, h2;
    int hash2(KEY const& key, int hash1)
    {
        // чтобы запомнить, какая хеш-функция использовалась для хеширования ключа.
        // Хитрость заключается в использовании h1 = hash1(key) и
        // h2= (hash2(key) - h1) & capacity, потому что таким образом
        // для заданной ячейки другая ячейка (hash2 (key) - cell) & capacity
        int result = h2(key) - hash1;
        if(result < 0) result += capacity;
        return result;
    }
    int findNode(KEY const& key)
    {
        for(int i = 0, cell = h1(key); i < 2; ++i)
```

```

    {
        if(isOccupied[cell] && table[cell].key == key) return cell;
        if(i == 0) cell = hash2(key, cell);
    }
    return NOT_FOUND;
}

void allocateTable()
{
    table = rawMemory<Node>(capacity);
    isOccupied = new bool[capacity];
    setGoodState();
}

void setGoodState()
{
    h1 = HASHER(capacity);
    h2 = HASHER(capacity);
    size = 0;
    for(int i = 0; i < capacity; ++i) isOccupied[i] = false;
}

static void cleanUp(Node* theTable, int theCapacity, bool* isOccupied)
{
    for(int i = 0; i < theCapacity; ++i)
        if(isOccupied[i]) theTable[i].~Node();
    rawDelete(theTable);
    delete[] isOccupied;
}

bool insertHelper(KEY const& key, VALUE const& value, bool duringResize)
{
    // коэффициент загрузки должен быть < 0,5, что является
    // фазовым переходом для универсальной хэш-функции на фазе удаления
    if(size > capacity * 0.4) resize(true);
    // вы можете сначала проверить, пусты ли оба места, а не только первое.
    // Это ценой дополнительного кода дает ускорение вставки примерно на 17%,
    // но не стоит того, т. к. хеширование кукушкой не используется
    for(Node node(key, value);;resize(false))
    {
        int cell = h1(node.key);
        // неясно, какое предельное максимальное значение
        // лучше использовать, но 50 вполне подходит
        for(int limit = 0; limit < 50; ++limit)
        {
            if(!isOccupied[cell])
            {
                ++size;
                new(&table[cell])Node(node);
                isOccupied[cell] = true;
                return true;
            }
            if(limit < 2 && table[cell].key == node.key)
            {
                table[cell].value = node.value;
            }
        }
    }
}

```

```

        return true;
    }
    swap(node, table[cell]);
    cell = hash2(node.key, cell);
}
if(duringResize) return false;
}

void resize(bool increaseSize)
{
    Node* oldTable = table;
    int oldCapacity = capacity;
    bool* oldIsOccupied = isOccupied;
    if(increaseSize) capacity = nextPowerOfTwo(max(4 * size, 8));
    allocateTable();
    for(int i = 0; i < oldCapacity; ++i)
    {
        if(oldIsOccupied[i] && !insertHelper(oldTable[i].key,
            oldTable[i].value, true))
        {
            setGoodState();
            i = -1;
        }
    }
    cleanUp(oldTable, oldCapacity, oldIsOccupied);
}

public:
    CuckooHashTable(int initialSize = 8): capacity(nextPowerOfTwo(max(
        initialSize, 8))), h1(capacity), h2(capacity) {allocateTable();}
    VALUE* find(KEY const& key)
    {
        int result = findNode(key);
        return result == NOT_FOUND ? 0 : &table[result].value;
    }
    ~CuckooHashTable() {cleanUp(table, capacity, isOccupied);}
    void insert(KEY const& key, VALUE const& value)
    {insertHelper(key, value, false);}
    void remove(KEY const& key)
    {
        int result = findNode(key);
        if(result != NOT_FOUND)
        {
            table[result].~Node();
            isOccupied[result] = false;
            if(--size < capacity * 0.1) resize(true);
        }
    }
    typedef Node NodeType;
    void getSize() {return size;}
    template<typename ACTION> void forEach(ACTION& action)

```



```

{
    for(int i = 0; i < capacity; ++i)
        if(table[i].isOccupied) action(&table[i]);
}
class Iterator
{
    int i;
    CuckooHashTable& hashTable;
    void advance()
        {while(i < hashTable.capacity && !hashTable.isOccupied[i])++i;}
public:
    Iterator(CuckooHashTable& theHashTable): i(0), hashTable(theHashTable)
        {advance();}
    bool hasNext()
        {return i < hashTable.capacity && hashTable.isOccupied[i];}
    NodeType* next()
    {
        Node* result = hasNext() ? &hashTable.table[i++] : 0;
        advance();
        return result;
    }
};
};

```

## 30.6. Немного приоритетных очередей

Первая альтернатива индексированной куче на основе репортеров — использовать указатели. В ней теряется часть эффективности из-за промахов кеша, но реализация проще:

```

template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM> >
class IndexedPointerHeap
{
    COMPARATOR c;
    struct Item
    {
        ITEM item;
        int index;
        Item(ITEM const& theItem, int theIndex):item(theItem),index(theIndex){}
    };
    Freelist<Item> freelist;
    Vector<Item*> items;
    int getParent(int i){return (i-1)/2;}
    int getLeftChild(int i){return 2*i+1;}
    void report(Item* item, int i){item->index = i;}
    void moveUp(int i)
    {
        Handle temp = items[i];
        for(int parent; i > 0 && c(temp->item,
            items[parent = getParent(i)]->item); i = parent)
            report(items[i] = items[parent], i);
        report(items[i] = temp, i);
    }
}

```

```

void moveDown(int i)
{
    Handle temp = items[i];
    for(int child; (child = getLeftChild(i)) <
        items.getSize(); i = child)
    {
        int rightChild = child + 1;
        if(rightChild < items.getSize() && c(items
            [rightChild]->item, items[child]->item)) child = rightChild;
        if(!c(items[child]->item, temp->item)) break;
        report(items[i] = items[child], i);
    }
    report(items[i] = temp, i);
}

void heapify()
{for(int i = getParent(items.getSize()-1); i >= 0; --i) moveDown(i);}

void remove(int i)
{
    freelist.remove(items[i]);
    if(items.getSize() > 1)
    {
        items[i] = items.lastItem();
        moveDown(i);
    }
    items.removeLast();
}

public:
    typedef Item* Handle;
    bool isEmpty() {return items.getSize() <= 0;}
    ITEM const& getMin() {assert(!isEmpty());return items[0]->item;}
    Handle insert(ITEM const& item)
    {
        Handle result = new(freelist.allocate())Item(item, items.getSize());
        items.append(result);
        moveUp(items.getSize()-1);
        return result;
    }
    ITEM deleteMin() {ITEM result = getMin();remove(0);return result;}
    void changeKey(Handle pointer, ITEM const& item)
    { // assert(указатель && указатель является полезным)
        bool decrease = c(item, pointer->item);
        pointer->item = item;
        decrease ? moveUp(pointer->index) : moveDown(pointer->index);
    }
    void decreaseKey(Handle index, ITEM const& item) {changeKey(index, item);}
    void deleteKey(Handle pointer) {remove(pointer->index);}
};

```

Основной обсуждаемый с теоретической точки зрения метод — это *куча спаривания*. Ему также нужно отображение от, скажем, номеров узлов графа до дескрипторов указателей, которая указана в обертке.

Куча представляет собой множество деревьев и указатель на то, чей корень имеет наименьший приоритет. Остальные деревья являются его дочерними элементами. Когда вставляется элемент, для него создается новое дерево, и в зависимости от его приоритета оно делается корневым или добавляется в качестве дочернего элемента корня. Удаление минимального элемента удаляет корень, линейно просматривает дочерние элементы, чтобы найти новый, и делает его родителем. Затем оставшиеся дочерние элементы объединяются путем разбиения на пары, и меньший член каждой пары становится родителем другого члена. Удаление отсекает поддерево, укорененное в узле дескриптора, и делает его дочерним по отношению к корню. Слияние делает больший корень дочерним элементом меньшего. Если есть указатели на правильного брата, самого старшего ребенка и самого младшего родителя, наименьший удаляется за амортизированное время  $\lg(n)$ , ключ уменьшается за время от  $O(\lg \lg(n))$  и  $2^{O(\sqrt{\log \log(n)})}$ , что на практике равно  $O(1)$ , и другие операции  $O(1)$ . В Википедии приведен более подробный анализ (рис. 30.2):

```
template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM> >
class PairingHeap
{
    /* Удаление в парной куче – это отвязка дерева с элементом, удаление в нем минимального
    значения, а затем, если удаленный элемент не был корневым, слияние с корнем.
    Чтобы увеличить ключ, нужно вырезать потомков и добавить в список братьев и сестер.
    Эти дополнительные операции не так полезны и не реализованы. */
    COMPARATOR c;
    int size;
    struct Node
    {
        ITEM element;
        Node* elder, *oldestChild, *youngerSibling;
        Node(ITEM const& theItem)
            :element(theItem), elder(0), oldestChild(0), youngerSibling(0){}
    } *root;
    Freelist<Node> freeList;
    PairingHeap(PairingHeap<ITEM> const&);
    PairingHeap& operator=(PairingHeap<ITEM> const&);
    void insertNode(Node* node)
    {
        if(c(root->element, node->element)) linkRoots(root, node);
        else{linkRoots(node, root);root = node;}
    }
    void linkRoots(Node* parent, Node* child)
    {
        // Текущие младшие братья и сестры игнорируются
        Node* oldestChild = parent->oldestChild;
        child->youngerSibling = oldestChild;
        if(oldestChild) oldestChild->elder = child;
        parent->oldestChild = child;
        child->elder = parent;
    }
    void pairUp()
    {
        for(Node* currentRoot = root->youngerSibling; currentRoot; )
```

```

    {
        Node* nextRoot = currentRoot->youngerSibling;
        if (!nextRoot)
        {
            insertNode(currentRoot);
            break;
        }
        Node* youngerSibling = nextRoot->youngerSibling;
        if (c(currentRoot->element, nextRoot->element))
            swap(nextRoot, currentRoot);
        linkRoots(nextRoot, currentRoot);
        insertNode(nextRoot);
        currentRoot = youngerSibling;
    }
    correctRoot();
}

void correctRoot() {root->elder = root->youngerSibling = 0;}

public:
    typedef Node* Handle;
    PairingHeap(): root(0), size(0){}
    Handle insert(ITEM const& theItem)
    {
        ++size;
        Node* node = new(freeList.allocate())Node(theItem);
        if (isEmpty()) root = node;
        else insertNode(node);
        return node;
        // assert(клиент не испортит кучу через newRoot)
    }
    Handle replaceMin(ITEM const& theItem)
    {
        deleteMin();
        return insert(theItem);
    }
    void decreaseKey(Handle node, ITEM const& newItem)
    {
        assert(node && !c(node->element, newItem) && !isEmpty());
        node->element = newItem;
        if (node != root)
        {
            Node *elder = node->elder, *youngerSibling = node->youngerSibling;
            if (youngerSibling) youngerSibling->elder = elder;
            (elder->oldestChild == node ?
             elder->oldestChild : elder->youngerSibling) = youngerSibling;
            insertNode(node);
            correctRoot();
        }
    }
}

bool isEmpty() {return !root;}
ITEM const& getMin() {assert(!isEmpty());return root->element;}
ITEM deleteMin()

```

```

{
    --size;
    ITEM result = getMin();
    Node* oldRoot = root;
    root = root->oldestChild;
    freeList.remove(oldRoot);
    if(root) pairUp();
    return result;
}
int getSize(){return size;}
// проверьте, что, когда требуется слияние, используется
// внешний свободный список
void merge(Node* otherRoot){insertNode(otherRoot);}
};

template<typename ITEM> struct IndexedPaHeap
{
    typedef typename PairingHeap<ITEM>::Handle HANDLE;
    LinearProbingHashTable<int, HANDLE> map;
    PairingHeap<ITEM> heap;
public:
    bool isEmpty(){return heap.isEmpty();}
    void insert(ITEM const& item, int handle)
        {map.insert(handle, heap.insert(item));}
    ITEM const& getMin(){return heap.getMin();}
    ITEM deleteMin(){return heap.deleteMin();}
    void changeKey(ITEM const& item, int handle)
    {
        HANDLE h = map.find(handle);
        if(h) heap.changeKey(h->item, Item(item, h));
        else insert(item, handle);
    }
    void deleteKey(int handle)
    {
        HANDLE h = map.find(handle);
        assert(h);
        heap.remove(*h);
    }
};

```

Наконец, интересным, но несколько не конкурентоспособным вариантом является *ведерная очередь* для элементов с приоритетами  $\in [0, N - 1]$ . Элемент вставляется путем помещения его в список, соответствующий его приоритету, и его индекс обновляется, если он является новым наименьшим. Удаление минимального элемента удаляет элемент из списка индекса и увеличивает индекс до тех пор, пока он не попадет в непустой список. Для удаления и изменения ключа элемента, на который указывает дескриптор, нужны двусвязные списки (здесь не реализовано). Удаление минимального выполняется за  $O(N)$ , а все остальные операции — за  $O(1)$  с наихудшим случаем, когда элементы с приоритетом 0 и  $N - 1$  многократно вставляются и удаляются. Слияние занимает время  $O(N)$ , если сохранить указатели на конец списка:

```
template<typename ITEM> class BucketQueue
{
    int capacity, minIndex;
    struct Node
    {
        ITEM item;
        Node* next;
        Node(ITEM const& theItem, Node* theNext):item(theItem), next(theNext){}
    }** buckets;
    Freelist<Node> freelist;
public:
    BucketQueue(int maxN): capacity(maxN + 1), buckets(new Node*[capacity]),
        minIndex(capacity){for(int i = 0; i < capacity; ++i)buckets[i] = 0;}
    void insert(int priority, ITEM const& item)
    {
        assert(priority < capacity);
        buckets[priority] =
            new(freelist.allocate())Node(item, buckets[priority]);
        if(priority < minIndex) minIndex = priority;
    }
    bool isEmpty(){return minIndex == capacity;}
    ITEM findMin(){assert(!isEmpty()); return buckets[minIndex]->item;}
    void deleteMin()
    {
        assert(!isEmpty());
        Node* temp = buckets[minIndex];
        buckets[minIndex] = temp->next;
        freelist.remove(temp);
        while(minIndex < capacity && !buckets[minIndex]) ++minIndex;
    }
    ~BucketQueue(){delete[] buckets;}
};
```

	Время сортировки $1,5 \times 10^7$ целых чисел % 10 (время, память <sub>1</sub> , память <sub>2</sub> )
Heap	1,4 (62/67)
IndexedHeap	7,2 (487/501)
IndexedArrayHeap	2,6 (179/198)
IndexedPointerHeap	3,1 (238/244)
PairingHeap	1,4 (297/296)
BucketQueue	0,6 (179/178)
PairingMapLP	5,2 (592/592)
PointerMapLP	8,2 (534/540)

Рис. 30.2. Сравнение производительности

## 30.7. Младший общий предок (LCA) и запрос с минимальным диапазоном (RMQ)

Оба алгоритма предполагают представление дерева с родителями и объединениями, основанное на массиве элементов, который легко изменить для работы с родительскими указателями и даже некоторыми краткими представлениями дерева.

LCA (Lowest Common Ancestor) — это, по сути, тот же алгоритм, что и в вопросе с собеседований о поиске точки слияния двух связанных списков: сначала отрегулировать разницу в высоте, затем увеличивать оба до слияния или добраться до корня без слияния:

```
template<typename ITEM> int LCA(ITEM* array, int i, int j)
{ // производительность O(h)
    int hDiff = 0;
    for(int k = i; array[k] != -1; k = array[k]) --hDiff;
    for(int k = j; array[k] != -1; k = array[k]) ++hDiff;
    if (hDiff < 0) {swap(i, j); hDiff = -hDiff;}
    while(hDiff--) j = array[j];
    while(i != j) {i = array[i]; j = array[j];}
    return i;
}
```

RMQ (Range-minimum Query) проще всего выполнять с деревом *минимума левого диапазона* (LRM, left Range Minimum). При известном размере массивов он вычисляет наименьшее левое значение для каждого. Чтобы поиск выполнялся только тогда, когда элементы слева уменьшаются, нужно использовать древовидную структуру:

```
template<typename ITEM> struct LRMTree
{ // предыдущее меньшее дерево значений
    Vector<int> parents;
    LRMTree(ITEM* array, int size): parents(size, -1)
    {
        for(int i = 1; i < size; ++i)
        { // расширение самой правой ветви, если наименьшее значение
          // не приводит к созданию новой самой правой ветви
            parents[i] = i - 1;
            while(parents[i] != -1 && array[parents[i]] >= array[i])
                parents[i] = parents[parents[i]];
        }
    }
    int RMQ(int left, int right)
    {
        int l = LCA(parents.getArray(), left, right);
        if(l == left) return l;
        while(parents[right] != l) right = parents[right];
        return right;
    }
};
```

В некоторых реализациях суффиксных индексов используются функциональные возможности RMQ, причем реализации настроены на представление индекса.

## 30.8. Знаковый ранговый критерий Уилкоксона для двух выборок

Предполагается, что выборки:

- ◆ распределены симметрично вокруг медианы (практически нормально, но могут быть толстые хвосты);
- ◆ происходят из непрерывного распределения.

Затем, когда наблюдения объединяются и преобразуются в ранги, нужно вычислить среднее значение и дисперсию распределения суммы *знакового ранга*, предполагая, что оно нормальное, и используя тест *z*-балла. Знаковый ранг = ранг | разница парных наблюдений | × знак, где знак = 1, если соответствующее первое групповое наблюдение > второго группового наблюдения, и -1 — в противном случае.

Ранги связанных наблюдений усредняются. Хороший способ справиться с нулевыми различиями — распределить их равномерно и отбросить единицу, если число нечетное (см. [30.5]). Теоретически связей быть не должно из-за предположения о непрерывности, но что касается теста на знаки, то связи случаются, и этот метод работает хорошо. Учитывая  $n$  оставшихся наблюдений, среднее = 0, а дисперсия =  $\sum rank_i^2$  (см. [30.7]).

Если есть равные ранги, разница на самом деле немного меньше, так что мы все равно должны контролировать ошибки типа I. Корректировка расчета дисперсии для увеличения мощности немного сложна, и здесь этого не делается:

```
struct SignedRankComparator
{
    typedef pair<double, double> P;
    int sign(P const& p) const { return p.first - p.second > 0 ? 1 : -1; }
    double diff(P const& p) const { return abs(p.first - p.second); }
    bool isLess(P const& lhs, P const& rhs) const
        { return diff(lhs) < diff(rhs); }
    bool isEqual(P const& lhs, P const& rhs) const
        { return diff(lhs) == diff(rhs); }
};

double signedRankZ(Vector<pair<double, double> > a)
{ // те же тесты, используйте версию Коновера
    SignedRankComparator c;
    quickSort(a.getArray(), 0, a.getSize() - 1, c);
    int nP = a.getSize(), i = 0;
    // если нулей нечетное количество, сначала удаляется значение,
    // а остальные распределяются равномерно
    while(i < a.getSize() && c.diff(a[i]) == 0) ++i;
    if(i % 2) --nP;
    double signedRankSum = 0, rank2Sum = 0;
    for(i = i % 2; i < a.getSize(); ++i)
    { // проверка ранга для поиска связей, вычисление суммы
        int j = i;
        while(i + 1 < a.getSize() && c.isEqual(a[i], a[i + 1])) ++i;
        double rank = (i + j)/2.0 + 1 + nP - a.getSize();
```



```

while(j <= i)
{
    signedRankSum += c.sign(a[j++]) * rank;
    rank2Sum += rank * rank;
}
}
return rank2Sum == 0 ? 0 : abs(signedRankSum)/sqrt(rank2Sum);
}

```

Если наблюдения распределены нормально, асимптотически критерий знакового ранга имеет 95% мощности  $t$ -критерия, а критерий знака — 50% мощности критерия знакового ранга (см. [30.7]).

Нормальная аппроксимация Уилкоксона является асимптотической и неточной в небольших выборках. Вместо этого можно использовать перестановочный тест с точными, а не ранговыми значениями. Однако эта аппроксимация лучше подходит для небольших выборок, если вы подозреваете, что есть выпадающие значения, и учитывается их дисперсия, уменьшенная на основе связей. Перестановочные тесты также хорошо работают с рангами и автоматически учитывают связи, но менее эффективны с точки зрения вычислений.

Техническая проблема аппроксимации Уилкоксона заключается в том, что никогда нельзя знать, симметрично ли распределение относительно медианы, поэтому в целом безопасным является только тест на знаки. Например, для распределения Бернулли критерий знаков является единственным допустимым вариантом, несмотря на то, что медиана здесь является плохой оценкой положения. Уилкоксон, как правило, предпочтительнее знакового теста.

Если симметрии не наблюдается, знакомый тест учитывает медиану разностей (ее оценку Ходжеса — Лемана — см. комментарии), которая  $\neq$  среднему значению. Таким образом, учитывая два распределения с одинаковым средним значением, где одно смещено влево, а другое — вправо, при  $n \rightarrow \infty$  тест на знаки покажет значительную разницу в медианах, хотя в средних значениях их нет.

## 30.9. Критерий Фридмана для согласованных выборок

Это обобщение знакового теста предполагает, что наблюдения происходят из непрерывного распределения (см. [30.16, 30.7]). Сначала нужно преобразовать наблюдения в ранги в каждой области, т. е. ранги должны быть уникальны с непрерывным распределением, но на практике получается одинаковое значение. Управление связями выполняется за счет усреднения одинаковых рангов. Требуется вычислить:

- ♦  $\forall 0 \leq j < k, r_j = \sum_i r_{ij}$  = сумма рангов для предметной области  $j$ ;
- ♦  $r_{ave} = (k + 1) / 2$  = ожидаемый ранг;
- ♦  $S = \sum_j (r_j - nr_{ave})^2$  = сумма квадратов разностей между суммами рангов альтернатив и их ожидаемым значением;

♦  $st = \sum_{ij} (r_{ij} - r_{ave})^2$  = сумма квадратов разностей между отдельными рангами и их ожидаемым значением.

Асимптотически  $(k-1)S/s_i \sim$  хи-квадрат с  $k-1$  степенями свободы при нулевом равенстве всех альтернатив. Также нужно вернуть  $gj$ , чтобы в дальнейшем можно было выполнить апостериорный попарный анализ:

```
pair<double, Vector<double> > FriedmanPValue(Vector<Vector<double> > const& a)
{ // a[i] - это вектор ответов для области i формулы
  assert(a.getSize() > 0 && a[0].getSize() > 1);
  int n = a.getSize(), k = a[0].getSize();
  double aveRank = (k + 1)/2.0, SSAlternative = 0, SSTotal = 0;
  Vector<double> alternativeRankSums(k);
  for(int i = 0; i < n; ++i)
  {
    assert(a[i].getSize() == k);
    Vector<double> ri = convertToRanks(a[i]);
    for(int j = 0; j < k; ++j)
    {
      alternativeRankSums[j] += ri[j];
      SSTotal += (ri[j] - aveRank) * (ri[j] - aveRank);
    }
  }
  for(int j = 0; j < k; ++j)
  {
    double temp = alternativeRankSums[j] - n * aveRank;
    SSAlternative += temp * temp;
  }
  double p =
    1 - evaluateChiSquaredCdf(SSAlternative * (k - 1)/SSTotal, k - 1);
  return make_pair(p, alternativeRankSums);
}
```

Асимптотическое распределение кажется достаточно хорошим даже для небольших выборок (см. [30.7]), но можно использовать перестановочное тестирование с  $S$ -статистикой, если нужен точный тест.

## 30.10. MADs-подобный алгоритм оптимизации

Более простой *случайный поиск с уменьшающимся шагом* (DSRS, Decreasing Step Random Search) может быть эффективным с  $O(S)$  оценок за итерацию:

1. Начать с некоторого начального шага размером  $s$ , обычно 1.
2. До сходимости, когда  $s$  становится слишком маленьким для указанной точности:
3. Создать случайное направление.
4. Сделать шаг в  $s \times$  направление, если это уменьшает значение функции.
5. В случае успеха удвоить  $s$ .
6. В противном случае уменьшить на какой-то коэффициент в диапазоне  $[0,5, 1]$ , обычно 0,8.

Влияние  $s$  оценивается из ряда Тейлора  $f$  как  $s(s + dd)$ , где  $dd$  = производная по направлению, оцененная от последнего перемещения. Эта стратегия завершения полезна, потому что, если выбрано неправильное направление, есть много шансов восстановиться до того, как  $s$  станет слишком маленьким:

```
template<typename FUNCTION> pair<Vector<double>, double>
    DSRS(Vector<double> const& x0, FUNCTION const& f,
    double step = 1, double factor = 0.8, int maxFEvals = 10000000,
    double yPrecision = numeric_limits<double>::epsilon())
{
    pair<Vector<double>, double> xy(x0, f(x0));
    for(double dd = 0; --maxFEvals && step * (dd + step) > yPrecision;)
    {
        Vector<double> direction = x0; // проверка ненулевого
                                     // направления
        for(int j = 0; j < direction.getSize(); ++j) direction[j] =
            GlobalRNG().uniform01() * GlobalRNG().sign();
        direction *= 1/norm(direction);
        double yNew = f(xy.first + direction * step);
        if(isELess(yNew, xy.second, yPrecision))
        {
            dd = (xy.second - yNew)/step;
            xy.first += direction * step;
            xy.second = yNew;
            step *= 2;
        }
        else step *= factor;
    }
    return xy;
}
```

## 30.11. Алгоритм классификации бустинга SAMME

Освежите свою память, вернувшись к посвященному бустингу (разд. 26.16):

```
template<typename LEARNER = NoParamsLearner<DecisionTree, int>,
    typename PARAMS = EMPTY, typename X = NUMERIC_X> class AdaBoostSamme
{
    Vector<LEARNER> classifiers;
    Vector<double> weights;
    int nClasses;
public:
    template<typename DATA> AdaBoostSamme(DATA const& data, PARAMS const&
        p = PARAMS(), int nClassifiers = 300): nClasses(findNClasses(data))
    {
        int n = data.getSize();
        assert(n > 0 && nClassifiers > 0 && nClasses > 0);
        Vector<double> dataWeights(n, 1.0/n);
        for(int i = 0; i < nClassifiers; ++i)
        {
            AliasMethod sampler(dataWeights);
```

```

    PermutedData<DATA> resample(data);
    for(int j = 0; j < n; ++j) resample.addIndex(sampler.next());
    classifiers.append(LEARNER(resample, p));
    double error = 0;
    Bitset<> isWrong(n);
    for(int j = 0; j < n; ++j) if(classifiers.lastItem().predict(
        data.getX(j)) != data.getY(j))
    {
        isWrong.set(j);
        error += dataWeights[j];
    }
    if(error >= 1 - 1.0/nClasses) classifiers.removeLast();
    else if(error == 0)
    { // замена ансамбля классификатором
        Vector<LEARNER> temp;
        temp.append(classifiers.lastItem());
        classifiers = temp;
        weights = Vector<double>(1, 1);
        break;
    }
    else
    {
        double expWeight = (nClasses - 1) * (1 - error)/error;
        weights.append(log(expWeight));
        for(int j = 0; j < n; ++j)
            if(isWrong[j]) dataWeights[j] *= expWeight;
        normalizeProbs(dataWeights);
    }
}

int predict(X const& x) const
{
    Vector<double> counts(nClasses, 0);
    for(int i = 0; i < classifiers.getSize(); ++i)
        counts[classifiers[i].predict(x)] += weights[i];
    return argMax(counts.getArray(), counts.getSize());
}
};

```

## 30.12. Бустинг в задаче регрессии

Что касается классификации, бустинг можно использовать для улучшения результатов, особенно дерева регрессии. Примените версию с повышением градиента (см. главу 26. *Машинное обучение: классификация*), чтобы изменить только  $y$  без выполнения повторной выборки. Для текущего  $F$  нужно найти  $a$  для последнего  $h$  такое, что  $\sum(F(x_i) + ah(x_i) - y_i)^2$  оказывается минимальна. Решение:  $a = \sum(y_i - F(x_i))h(x_i) / \sum h(x_i)$  (см. [30.17]). Для регрессии повышение интуитивно похоже на удвоение по Тьюки, т. е. сначала выполняется регрессия для  $y$ , а затем снова для ошибок:

```

template<typename LEARNER = NoParamsLearner<RegressionTree, double>,
        typename PARAMS = EMPTY, typename X = NUMERIC_X> class L2Boost

```

```

{
    Vector<LEARNER> classifiers;
    Vector<double> weights;
    double getWeight(int i) const {return 1/pow(i + 1, 0.501);}
    struct L2Loss
    {
        Vector<double> F;
        L2Loss(int n): F(n, 0) {}
        double getNegGrad(int i, double y) {return 2 * (y - F[i]);}
        double loss(int i, double y) {return (F[i] - y) * (F[i] - y);}
    };
public:
    template<typename DATA> L2Boost(DATA const& data,
        PARAMS const& p = PARAMS(), int nClassifiers = 300)
    {
        int n = data.getSize();
        assert(n > 0 && nClassifiers > 0);
        L2Loss l(n);
        RelabeledData<DATA> regData(data);
        for(int j = 0; j < n; ++j) regData.addLabel(data.getY(j));
        for(int i = 0; i < nClassifiers; ++i)
        { // поиск меток, назначение новых меток данных, обучение учащегося
          // и обновление F
            for(int j = 0; j < n; ++j)
                regData.labels[j] = l.getNegGrad(j, data.getY(j));
            classifiers.append(LEARNER(regData, p));
            Vector<double> h;
            for(int j = 0; j < n; ++j)
                h.append(classifiers.lastItem().predict(data.getX(j)));
            double sumH2 = 0, weight = 0;
            for(int j = 0; j < n; ++j)
            {
                sumH2 += h[j] * h[j];
                weight += (data.getY(j) - l.F[j]) * h[j];
            }
            if(weight > 0 && isfinite(weight/sumH2))
            {
                weights.append(weight/sumH2);
                for(int j = 0; j < n; ++j)
                    l.F[j] += weights.lastItem() * h[j];
            }
            else
            {
                classifiers.removeLast();
                break;
            }
        }
    }
    double predict(X const& x) const
    {
        double sum = 0;

```

```

    for(int i = 0; i < classifiers.getSize(); ++i)
        sum += classifiers[i].predict(x) * weights[i];
    return sum;
}
};

```

Можно попробовать веса RM вместо оптимальных или дроби RM (вместо неявной единицы), если возникают проблемы с переобучением. Из-за отсутствия повторной выборки возможно досрочное завершение 0, а значение в 300 деревьев может быть недостижимо, что может дать преимущество в эффективности по сравнению со случайным лесом.

## 30.13. Вероятностная кластеризация

Пожалуйста, освежите свою память, вернувшись к разделу комментариев в *главе 28. Машинное обучение: кластеризация*. Используйте алгоритм ЕМ для обучения смеси  $k$  гауссианов:

1. Выбрать начальные значения для параметров: смешать  $w$ , средние  $m$  и ковариации  $\Sigma$ .
2. До схождения:
3. Рассчитать вероятности принадлежности к классу образцов, используя параметры.
4. Рассчитать параметры, используя вероятности членства.
5. Назначить все выборки их кластерам максимального правдоподобия.

Шаг (3) — это Е-шаг, а (4) — это М-шаг, но соответствие не является интуитивным. Оценка принадлежности к классу — это ожидание специальной функции, и (4) использует максимальную вероятность для оценки параметров. Для многомерной нормы со средним значением  $m$  и ковариационной матрицей  $\Sigma$  для одного логарифмического правдоподобия  $x$ :

$$LL(x) = -\frac{1}{2} \left( D \ln(2\pi) + \ln(\det(\Sigma)) + (x - \bar{x}) \Sigma^{-1} (x - \bar{x}) \right).$$

Для смеси с кластерными весами  $w$ :

- ◆ логарифмически взвешенное правдоподобие  $lwl(w, x) = \ln(w) + LL(x)$ ;
- ◆ общая логарифмическая вероятность для всех данных  $\sum_i \ln \left( \sum_j e^{lwl(w_j, x_i)} \right)$ .

Уравнения обновления:

- ◆ вероятность членства  $g_{ij}$  для примера  $i$  и класса  $j$  пропорциональна  $lwl(w_j, x_i)$ ;
- ◆ общая поддержка для класса  $j$ ,  $n_j = \sum_i g_{ij}$ ;
- ◆  $w_j = \frac{n_j}{n}$ ;

$$\blacklozenge \bar{x}_j = \frac{\sum_i g_{ij} x_i}{n_j};$$

$$\blacklozenge \sum_j = \frac{\sum_i g_{ij} (x_i - \bar{x}_j) \otimes (x_i - \bar{x}_j)}{n_j}.$$

Значение  $\Sigma_j$  не зависит от степеней свободы, поскольку используются оценки максимального правдоподобия. Что касается  $k$ -средних, то логарифмическая вероятность никогда не уменьшается (см. [30.8]), поэтому алгоритм представляет собой надлежащий локальный поиск (правда, иногда численные ошибки или эвристики надежности могут сделать алгоритм неработоспособным).

В реализации начальные значения  $x$  с чертой определяются несколькими  $k$ -средними, и используется алгоритм ВИС, т. к. его поддержка в этом случае естественна. Для определения сходимости нужно проверить, что относительное изменение логарифмической вероятности  $\leq 10^{-5}$  (см. [30.9]). Для безопасности количество итераций ограничено константой, по умолчанию равной 1000.

Мне так и не удалось решить численные проблемы и получить достаточную надежность (хотя приведенная далее реализация хорошо работала на всех протестированных наборах данных) в следующих ситуациях:

- ◆ получение сингулярностей в  $\Sigma$  при таких условиях, как функции с нулевой дисперсией. Надежность можно повысить путем усреднения некоторой объединенной дисперсии, но лучший метод еще предстоит определить;
- ◆ даже с объединенной дисперсией можно получить кластеры с поддержкой менее одного примера  $n_j < 1$ . Например, большой компонент может полностью включать меньший, оставив последний с почти нулевой поддержкой. Хорошего решения для этой проблемы я не вижу, кроме отказа от испытанного количества кластеров;
- ◆ инвертирование суммы с помощью разложения Холецкого (наиболее устойчивый метод для симметричных и положительно определенных матриц, см. главу 22. *Численные алгоритмы: введение и матричная алгебра*) иногда приводило к численным ошибкам, несмотря на вычисление  $hwI$  безопасным способом.

Начальное значение  $m$  определяется повторными  $k$ -средними, веса  $w$  задаются равными  $1/k$ , а  $\Sigma$  равной идентичности  $\times$  объединенную дисперсию, где последняя основана

на начальном  $m$  и  $= \frac{\sum_i (x_i - m(i))^2}{(n - k)D}$ . Объединенная дисперсия предполагает диаго-

нальную сумму  $\Sigma$  с одинаковой дисперсией для каждой компоненты. Поскольку оценочные  $k$  усреднены, получаем  $n - k$  степеней свободы. Это упрощает реализацию, и правильные  $w$  и  $\Sigma$  вычисляются на следующей итерации. В случае численных проблем по умолчанию используется исходная кластеризация и возвращается ее логарифмическое правдоподобие.

Некоторые числовые проблемы еще предстоит решить, но они редко обсуждаются в литературе, и лучший способ их решения неясен. Используйте следующие эвристики:

- ◆ усреднять  $I \times$  объединенную дисперсию с поддержкой 1 в  $\Sigma$  (с поддержкой  $n_j$ ) для предотвращения сингулярностей в таких условиях, как функции с нулевой дисперсией. Объединенная дисперсия изменяется вместе с  $w$  и  $m$ , но ее обновление приве-

дет к утрате «априорности». Удаление признаков с нулевой дисперсией может показаться более правильным, но некоторые кластеры могут по-прежнему иметь нулевую дисперсию по конкретному признаку;

- ◆ если выполнить разложение Холецкого не удастся, алгоритм завершается. Это говорит о наличии проблемы, которую нельзя исправить;
- ◆ тот факт, что  $\sum e^{x_i} = e^b \sum e^{x_i - b}$ , где  $b = \max(x_i)$ , делает использование  $|w|$  численно более устойчивым, потому что первая часть обычно несет большую часть значения суммы и может обрабатываться отдельно;
- ◆ проверка на поддержку менее 1 примера поддерживает свертывание  $n_j < 1$  и завершает работу алгоритма, если условие выполнено. Объединенный вариант препятствует свертыванию, но не может его предотвратить, а большой компонент может полностью поглотить меньший, оставив последнему поддержку, близкую к нулю;
- ◆ проверка логарифмической вероятности бесконечности и NaN. Эта эвристика совершенно универсальна.

```
struct EMClustering
{
    static double normalLL(Vector<double> x, Vector<double> const& m,
        Cholesky<double> const& l)
    {
        x -= m;
        return -(x.getSize() * log(2 * PI()) + l.logDet() +
            dotProduct(l.solve(x), x))/2;
    }
    static double llim() {return log(numeric_limits<double>::min())/2;}
    static pair<Vector<double>, double> findILL(NUMERIC_X const& x,
        Vector<double> const& w, Vector<Vector<double> > const& m,
        Vector<Cholesky<double> > const &ls)
    {
        int k = w.getSize();
        Vector<double> temp(k);
        for(int j = 0; j < k; ++j)
            temp[j] = w[j] > 0 ? log(w[j]) + normalLL(x, m[j], ls[j]) : 0;
        double b = argMax(temp.getArray(), k);
        for(int j = 0; j < k; ++j) temp[j] -= b;
        return make_pair(temp, b);
    }
    template<typename DATA> static double findLL(DATA const& data,
        Vector<double> const& w, Vector<Vector<double> > const& m,
        Vector<Cholesky<double> > const &ls)
    {
        double ll = 0;
        for(int i = 0; i < data.getSize(); ++i)
        {
            pair<Vector<double>, double> temp = findILL(data.getX(i), w, m, ls);
            double kSum = temp.second;
            for(int j = 0; j < w.getSize(); ++j) kSum += exp(temp.first[j]);
            ll += log(kSum);
        }
    }
}
```



```

    return ll;
}

template<typename DATA> ClusterResult operator()(DATA const& data, int k,
    int maxIterations = 1000) const
{
    int n = data.getSize(), D = getD(data);
    assert(k > 0 && k <= n && n > 0);
    // начальные значения
    Vector<int> assignments =
        RepeatedKMeans()(data, k, maxIterations).assignments;
    Vector<Vector<double>> m = findCentroids(data, k, assignments),
        g(n, Vector<double>(k));
    Vector<double> w(k, 1.0/k);
    double pooledVar = 0;
    for(int i = 0; i < n; ++i)
    {
        Vector<double> diff = (data.getX(i) - m[assignments[i]]);
        pooledVar += dotProduct(diff, diff);
    }
    pooledVar /= (n - k) * D;
    Vector<Cholesky<double>> ls(k,
        Cholesky<double>(Matrix<double>::identity(D) * pooledVar));
    double ll = findLL(data, w, m, ls);
    while(maxIterations--)
    { // var E
        for(int i = 0; i < n; ++i)
        {
            pair<Vector<double>, double> temp = findILL(data.getX(i), w, m, ls);
            for(int j = 0; j < k; ++j) g[i][j] = exp(temp.first[j]);
            normalizeProbs(g[i]);
        }
        // var M
        bool isNumericIssue = false;
        for(int j = 0; j < k; ++j)
        { // nj
            double nj = 0;
            for(int i = 0; i < n; ++i) nj += g[i][j];
            if(nj < 1){isNumericIssue = true; break;}
            // w
            w[j] = nj/n;
            // m
            for(int i = 0; i < n; ++i) m[j] += data.getX(i) * g[i][j];
            m[j] *= 1/nj;
            // объединенная дисперсия
            Matrix<double> covar = Matrix<double>::identity(D) *
                pooledVar;
            for(int i = 0; i < n; ++i)
            {
                Vector<double> xm = data.getX(i) - m[j];
                covar += outerProduct(xm, xm) * g[i][j];
            }
        }
    }
}

```

```

        covar *= 1/(1 + nj);
        ls[j] = Cholesky<double>(covar);
        if (ls[j].failed){isNumericIssue = true; break;}
    }
    if (isNumericIssue) break;
    double newLL = findLL(data, w, m, ls);
    if (!isfinite(newLL) || !isELess(ll, newLL, 0.00001)) break;
    ll = newLL;
}
double BIC = -2 * ll + (k * (1 + D + D * (D + 1)/2) - 1) * log(n);
for(int i = 0; i < n; ++i)
    assignments[i] = argMax(g[i].getArray(), k);
return ClusterResult(assignments, BIC);
}
};

```

Для  $D > 100$  используется метод  $k$ -средних, потому что ЕМ становится неэффективным:

```

typedef FindKClusterer<NoParamsClusterer<EMClustering> > EMBIC;
struct EMSmart
{
    KMeansGeneral km;
    EMBIC em;
    template<typename DATA> bool useKMeans(DATA const& data) const
    {
        return getD(data) > 100; // для эффективности
        // и на случай числовых ошибок
    }
    template<typename DATA> ClusterResult operator()(DATA const& data,
        int k) const
    {
        if (useKMeans(data)) return km(data, k);
        else return em(data, k);
    }
    template<typename DATA> ClusterResult operator()(DATA const& data) const
    {
        if (useKMeans(data)) return km(data);
        return em(data);
    }
};

```

## 30.14. Иерархическая кластеризация

Пожалуйста, освежите свою память, вернувшись к разделу комментариев в *главе 28. Машинное обучение: кластеризация*. Вы можете эффективно вычислить среднюю связь, используя *формулу Ланса — Уильямса* (см. [30.1]): когда группы  $i$  и  $j$  объедине-

ны, а для других групп  $k$   $d(k, ij) = \frac{n_i d(k, i) + n_j d(k, j)}{n_i + n_j}$ .

Иерархическая кластеризация имеет ряд преимуществ по сравнению с другими методами (см. [30.18]). В частности, она оценивает ожидаемое групповое расстояние между любыми двумя группами и в этом смысле не зависит от количества примеров в каждом кластере. Мало метрик, которые обладают этим свойством. Она также работает с лю-

бой мерой несходства (в отличие от расстояния, несходство не обязано удовлетворять неравенству треугольника, усредненные расстояния обычно перестают быть расстояниями и становятся несходствами).

Приведенный далее код содержит ошибку, вызывающую сбой теста (см. *разд. Советы по дополнительной подготовке*):

```
template<typename DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
struct HierarchicalClustering
{
    struct Node
    {
        union{int i, sequenceNumber;};
        int size;
        Node *left, *right;
        bool isLeaf(){return !left;}
        Node(int theI): i(theI), left(0), right(0), size(1){}
        bool operator<(Node const& rhs)const // используется большее значение
            {return sequenceNumber > rhs.sequenceNumber;}
    } *root;
    Freelist<Node> f;
    void createAssignmentsHelper(Node* node, Vector<int>& assignments, int nextC)const
    {
        if(node->isLeaf()) assignments[node->i] = nextC;
        else
        {
            createAssignmentsHelper(node->left, assignments, nextC);
            createAssignmentsHelper(node->right, assignments, nextC);
        }
    }
    Vector<int> createAssignments(int k, int n)const
    {
        Vector<int> assignments(n);
        Heap<Node*, PointerComparator<Node> > h;
        h.insert(root);
        int nextC = 0;
        while(!h.isEmpty())
        {
            Node* node = h.deleteMin();
            if(node->isLeaf() || k < 2)
                createAssignmentsHelper(node, assignments, nextC++);
            else
            {
                --k;
                h.insert(node->left);
                h.insert(node->right);
            }
        }
        return assignments;
    }
}

template<typename DATA> HierarchicalClustering(
    DATA const& data, DISTANCE const& d = DISTANCE()): root(0)
```

```

{
    int n = data.getSize(), sequenceNumber = n;
    Vector<Node*> nodeIndex(n);
    for(int i = 0; i < n; ++i) nodeIndex[i] = new(f.allocate())Node(i);
    IndexedArrayHeap<double> iHeap;
    for(int i = 1; i < n; ++i) for(int j = 0; j < i; ++j)
        iHeap.insert(d(data.getX(i), data.getX(j)), i * n + j);
    while(!iHeap.isEmpty())
    {
        IndexedArrayHeap<double>::ITEM_TYPE minP = iHeap.deleteMin();
        int index = minP.value, j = index % n,
            i = index/n; // согласно дизайну, j < i
        // обновить меньшее расстояние пары индексов,
        // поместить узел в маленькую индексную карту
        Node* winnerNode = new(f.allocate())Node(sequenceNumber++);
        int sizeI = nodeIndex[i]->size, sizeJ = nodeIndex[j]->size;
        winnerNode->left = nodeIndex[j];
        winnerNode->right = nodeIndex[i];
        winnerNode->size = sizeI + sizeJ;
        nodeIndex[j] = winnerNode;
        nodeIndex[i] = 0;
        for(int k = 0; k < n; ++k) // обновить все расстояния i и j
            if(i != k && j != k && nodeIndex[k])
            {
                int indexIK = max(i, k) * n + min(i, k),
                    indexJK = max(j, k) * n + min(j, k);
                double dijk = (sizeI * *iHeap.find(indexIK) +
                    sizeJ * *iHeap.find(indexJK))/(sizeI + sizeJ);
                iHeap.changeKey(dijk, indexJK);
                iHeap.deleteKey(indexIK);
            }
    }
    root = nodeIndex[0];
}

struct Functor
{
    template<typename DATA> ClusterResult operator()(DATA const& data,
        int k, HierarchicalClustering const& h) const
    {
        Vector<int> assignments = h.createAssignments(k, data.getSize());
        return ClusterResult(assignments, -clusterSilhouette(data,
            assignments, DISTANCE()));
    }
    template<typename DATA> ClusterResult operator()(DATA const& data, int k) const
    {return operator()(data, k, HierarchicalClustering(data));}
    template<typename DATA> ClusterResult operator()(DATA const& data) const
    { // Неправильно! -- требуется повторное вычисление -- исправить!
        return findClustersAndK(data, Functor(),
            HierarchicalClustering(data));
    }
};
};

```

## 30.15. Кластеризация на основе плотности

Прежде чем двигаться дальше, еще раз просмотрите раздел комментариев в *главе 28. Машинное обучение: кластеризация*. Для алгоритма DBSCAN при заданных параметрах  $\epsilon$  и  $nMin$ :

- ◆ центральная точка имеет  $\geq nMin$  в радиусе  $\epsilon$ ;
- ◆ граничная точка имеет центральную точку в пределах радиуса  $\epsilon$ , но сама таковой не является;
- ◆ шумовая точка не является ни тем, ни другим.

*Шумовые точки* считаются выбросами, не принадлежащими ни к одному кластеру. Соседние центральные точки образуют кластеры, а пограничные точки входят в кластеры своей центральной точки. Труднее всего определить параметры. Используется присвоение  $nMin = \lceil \log(n) \rceil$ , т. к. из-за аргументов  $k$ -NN в нем больше смысла, чем в методе, предложенном в оригинальной статье. В последнем случае предлагается задавать  $nMin = 4$  и  $min \epsilon$  таким образом, чтобы пропорция шумовых точек была приемлемой, но более точных указаний не дается.

В исходном DBSCAN центральные точки должны для соединения лежать в пределах радиуса  $\epsilon$ . Но граничные точки, соседние с двумя центральными точками, могут принадлежать любой из них и присваиваются в зависимости от порядка примеров. Простое решение состоит в том, чтобы связать основные точки с общей пограничной точкой. Это также позволяет выполнить простую реализацию с помощью `union-find`, а изменение незначительно (по сути, значение  $\epsilon$  удваивается):

1. Для любой точки:
2.     Если она центральная, присоединить к ней ее  $\epsilon$ -соседей.
3. Для любого корня `union-find`:
4.     Проверить, содержит ли он основные точки.
5.     Если да, точка образует кластер, иначе шум.
6. Назначить все точки кластерам, представленным их корнями.

Эвристика, реализованная здесь для настройки параметров:

1. Рассчитать среднее значение и стандартное отклонение расстояния до  $nMin$ -го ближайшего соседа по всем точкам.
2. Начать с  $z = -3$ :
3.     Использовать  $\epsilon = \text{среднее} + z \times \text{стандартное отклонение}$ .
4.     Остановить алгоритм, если доля шума  $< 0,05$  или  $z = 3$ , иначе  $z += 0,5$ .

Логика алгоритма опирается на неравенство Чебышева, согласно которому для любого распределения большинство данных находится в пределах нескольких стандартных отклонений от среднего (см. *главу 21. Вычислительная статистика*). Здесь это просто приближение, поскольку параметры оцениваются), поэтому поиск параметров кажется достаточно исчерпывающим и эффективным. Для повышения эффективности используйте индексную структуру данных, которая допускает запросы радиуса и  $k$ -NN. Дерево VP особенно подходит для работы с метрическими расстояниями:

```

template<typename DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
struct DBSCAN
{
    template<typename DATA, typename TREE> static pair<Vector<int>, double>
        findClusters(TREE const& t, DATA const& data, int nMin, double eps)
    { // определить центральные точки и граничные точки
        int n = data.getSize();
        UnionFind uf(n);
        Vector<bool> isCore(n);
        Vector<int> assignments(n, -1);
        for(int i = 0; i < n; ++i)
        {
            Vector<typename TREE::NodeType*> epsNeighbors =
                t.distanceQuery(data.getX(i), eps);
            if(epsNeighbors.getSize() - 1 >= nMin)
            {
                isCore[i] = true;
                for(int j = 0; j < epsNeighbors.getSize(); ++j)
                {
                    int v = epsNeighbors[j]->value;
                    uf.join(i, v);
                    assignments[v] = i; // повторное использование массива
                                     // для определения членства
                }
            }
        } // определить центральные/шумовые кластеры в зависимости
        // от того, являются ли их корни центральными
        for(int i = 0; i < n; ++i) if(isCore[i]) isCore[uf.isRoot(i)] = true;
        int k = 0;
        for(int i = 0; i < n; ++i)
            if(uf.isRoot(i) && isCore[i]) assignments[i] = k++;
        // найти классы граничных точек
        // значение -1 вместо k + 1 для совместимости с кодом анализа
        int noise = 0;
        for(int i = 0; i < n; ++i)
        {
            if(assignments[i] == -1)
            {
                assignments[i] = k;
                ++noise;
            }
            else if(!uf.isRoot(i))
                assignments[i] = assignments[uf.find(assignments[i])];
        }
        return make_pair(assignments, noise * 1.0/n);
    }
}

template<typename DATA> ClusterResult operator()(
    DATA const& data, double noisePercentage = 0.05) const
{ // оценка параметров
    int n = data.getSize(), nMin = log(n) + 1;
    assert(n > 0);
    typedef VpTree<typename DATA::X_TYPE, int, DISTANCE> TREE;

```

```

TREE t;
for(int i = 0; i < n; ++i) t.insert(data.getX(i), i);
IncrementalStatistics s;
DISTANCE d;
for(int i = 0; i < n; ++i)
{
    Vector<typename TREE::NodeType*> nMinNNs = t.kNN(data.getX(i),
        nMin + 1); // +1 для себя
    s.addValue(d(data.getX(i), nMinNNs.lastItem()->key));
}
double stdev = sqrt(s.getVariance()), z = -3;
pair<Vector<int>, double> result;
do
{
    result = findClusters(t, data, nMin, s.getMean() + z * stdev);
    z += 0.5;
}while(result.second >= noisePercentage && z <= 3);
return ClusterResult(result.first);
}
};

```

Требуется  $O(n)$  запросов  $k$ -NN для выбора параметра и  $O(n)$  запросов радиуса для определения состояния точек. Для низкоразмерных данных время выполнения обычно составляет  $O(\lg(n) + nMin)$  времени для каждого, но может быть близко к  $O(n)$  для многомерных данных без пригодной для эксплуатации структуры.

## 30.16. Не представленные реализации

Многие реализации были отброшены до того, как в их описании возникла необходимость. Я не хотел, чтобы эта глава стала свалкой для разного кода, поэтому здесь они лишь кратко упомянуты, а код приведен на веб-сайте книги.

- ◆ левое красно-черное дерево — базовая версия без итераторов и аугментаций;
- ◆ двусторонняя куча — на самом деле неплоха с точки зрения эффективности в своей нише. У меня просто не было времени, чтобы описать должным образом эту структуру и подходящие для нее задачи;
- ◆ хеширование во внешней памяти — сегодня используются криптографические хеш-функции, потому что они занимают меньше времени, чем функции ввода/вывода. Часто задействуется линейное зондирование с применением ЕМ-вектора, но ручной способ может работать лучше за счет более удобного изменения размера:
  - *расширяемое хеширование* — это известное решение проблемы изменения размера. По сути, оно представляет собой дерево внутренней памяти, хвосты которого группируются в корзины, а сегменты хранятся во внешней памяти. К сожалению, эта структура занимает слишком много внутренней памяти для дерева, которое имеет по крайней мере  $n/B$  указателей на страницы корзины. Структуру дерева можно хранить во внешней памяти, но одна вставка может привести к большому количеству повторных вычислений;
  - *линейное хеширование* также позволяет изменять размер и не требует большого количества внутренней памяти. К сожалению, в ходе работы над кодом я что-то

сломал, и ни линейное хеширование, ни линейное зондирование ЕМ больше не работают, так что дальнейшая информация приведена в *разд. Советы по дополнительной подготовке*;

- ◆ адаптивное арифметическое кодирование — к сожалению, код работает лишь в некоторых случаях. Метод интересен сам по себе из-за *древовидной структуры данных Фенвика* для адаптивных обновлений;
- ◆ *интервальное дерево* — найти все интервалы, содержащие точку. Если  $n$  интервалов хранятся в статической структуре, их можно вычислить за время  $O(\lg(n))$ . (Подробнее см. в работе [30.3].) Алгоритм не был реализован из-за отсутствия вариантов использования и динамического обновления. То же самое справедливо для многих других подобных структур данных, таких как *дерево сегментов*, *дерево приоритетного поиска* и т. д.;
- ◆ хеширование с учетом местоположения — для евклидовых данных (дополнительные ссылки см. в работах [30.12, 30.2 и 30.15]). Чтобы использовать его для классификации с ближайшими соседями, примените поиск по сетке, чтобы найти хороший диапазон параметров, и вернитесь к ближайшему среднему классификатору, если ничего не найдено;
- ◆ версия BFGS, которая начинается с конечно-разностной оценки Гессе и корректирует или пересчитывает ее, если она не является положительно определенной;
- ◆ оптимизация с сопряженным градиентом — я использовал *формулу PRP+* (см. [30.13]) и тот же начальный шаг, поиск строки, политику перезапуска и условия завершения, что и L-BFGS (хотя масштабирование шага для L-BFGS позже было улучшено, но на производительность теста это не влияет). Этот начальный шаг кажется более надежным, чем сопоставление изменений первого порядка (см. [30.13]). Вблизи минимума, где квадратичная функция является хорошим приближением, в точной арифметике и при точном линейном поиске достаточно  $D$  итераций. Исходя из этого, сделан вывод, что лучше всего перезапускать CG через каждые  $D$  итераций, но результаты сходимости для формулы PRP+ в этом не нуждаются (см. [30.13]). По сути, у алгоритма есть возможность автоматического перезапуска. Тем не менее его необходимо перезапустить, потому что направления могут испортиться из-за меняющегося числового состояния, как и у L-BFGS. Формула *Хагера* — *Чжана* показала себя хуже, чем PRP+, несмотря на его поддержку в работе [30.19]). Чтобы сопряженный градиент в PRP+ сходил, от задачи требуется меньше (например, нет ограничений на  $\kappa(H_0)$  — подробности см. в работе [30.13], но на практике это, похоже, не имеет значения;
- ◆ алгоритм выбора признаков MRMR для обучения с учителем и оценка взаимной информации в нем.

## 30.17. Советы по дополнительной подготовке

- ◆ Поэкспериментируйте с одним из описанных здесь алгоритмов. Подготовьте хорошие тестовые файлы. Согласны ли вы, что эти алгоритмы не столь хороши? Если да, стали бы вы когда-нибудь использовать и поддерживать такой алгоритм или структуру данных в производственном коде?
- ◆ Исправьте код хеширования ЕМ. Добавьте постоянство, чтобы разрешить инициализацию из файла хранилища.



- ◆ Исправьте код адаптивного арифметического кодирования.
- ◆ Улучшите точность завершения и масштабирование DSRS (это уже было реализовано еще до того, как я об этом задумался).
- ◆ Вместо BIC попробуйте ЕМ-кластеризацию с упрощенным силуэтом. Сравните этот алгоритм с использованием повторяющихся  $k$ -средних для выбора  $k$  с последующим ЕМ.
- ◆ Исправьте код иерархической кластеризации.

## 30.18. Список рекомендуемой литературы

- 30.1. Aggarwal C. C., & Reddy C. K. (Eds.). (2014). *Data Clustering: Algorithms and Applications*. CRC.
- 30.2. Andoni A., & Indyk P. (2008). Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. *Communication of the ACM*, 51(1), 117.
- 30.3. de Berg M., Cheong O., & Van Kreveld M., Overmars M. (2008). *Computational Geometry: Algorithms and Applications*. Springer.
- 30.4. Dahlquist G., & Björck Å. (2008). *Numerical Methods in Scientific Computing*. SIAM.
- 30.5. Demšar J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research*, 7(Jan), 1–30.
- 30.6. Dietzfelbinger M., & Schellbach U. (2009). On risks of using cuckoo hashing with simple universal hash classes. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms* (pp. 795–804). Society for Industrial and Applied Mathematics.
- 30.7. Gibbons J. D., & Chakraborti S. (2011). *Nonparametric Statistical Inference*. Springer.
- 30.8. Gupta M. R., & Chen Y. (2010). *Theory and Use of the EM Algorithm*. Now Publishers.
- 30.9. Fraley C., Raftery A. E., Murphy T. B., & Scrucca L. (2012). *mclust version 4 for R: Normal mixture modeling for model-based clustering, classification, and density estimation*. Department of Statistics, University of Washington, 23, 2012.
- 30.10. Hennig C., Meila M., Murtagh F., & Rocci R. (Eds.). (2016). *Handbook of Cluster Analysis*. CRC.
- 30.11. Kutzelnigg R. (2009). *Random Graphs and Cuckoo Hashing: A Precise Average Case Analysis of Cuckoo Hashing and Some Parameters of Sparse Random Graphs*. Suedwestdeutscher Verlag fuer Hochschulschriften.
- 30.12. Leskovec J., Rajaraman A., & Ullman J. D. (2020). *Mining of Massive Data Sets*. Cambridge University Press.
- 30.13. Nocedal J., Wright S. (2006). *Numerical Optimization*. Springer.
- 30.14. Paredes R., & Navarro G. (2006). Optimal incremental sorting. In *Proc. 8th Workshop on Algorithm Engineering and Experiments and 3rd Workshop on Analytic Algorithmics and Combinatorics (ALENEXANALCO'06)* (pp. 171–182). SIAM.
- 30.15. Slaney M., & Casey M. (2008). Locality-sensitive hashing for finding nearest neighbors. *IEEE Signal Processing Magazine*, 25(2), 128–131.
- 30.16. Wikipedia (2015). Friedman test. [https://en.wikipedia.org/wiki/Friedman\\_test](https://en.wikipedia.org/wiki/Friedman_test). Accessed November 18, 2015.
- 30.17. Schapire Robert E. *Boosting: foundations and algorithms* / Robert E. Schapire and Yoav Freund. The MIT Press, Cambridge, Massachusetts, London, England, 1012.
- 30.18. Kauffman L., & Rousseeuw P. J. (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley.
- 30.19. Hager W. W., & Zhang H. (2005). A new conjugate gradient method with guaranteed descent and an efficient line search. *SIAM Journal on optimization*, 16(1), 170–192.

## 31. Приложение: примечания о языке C++

### 31.1. Введение

Меня часто спрашивают, как улучшить свои навыки в C++. Для прочтения этой книги знания C++ обязательны. Поэтому я решил порекомендовать ресурсы, которые счел полезными.

### 31.2. Путеводитель по литературе, посвященной C++

Мои любимые сайты по этому языку: [en.cppreference.com](http://en.cppreference.com) и [cplusplus.com](http://cplusplus.com). Они актуальны, просты в навигации и содержат краткие примеры использования функций. Поиск в Google также часто ведет на [stackoverflow.com](http://stackoverflow.com), где можно найти ответы на многие вопросы и разборы ошибок компилятора. Но этот сайт подойдет только для тех, кто уже знаком с языком.

Новичку следует начать с простой книги. У меня нет конкретных предложений на этот счет, но в хорошей книге должны рекомендоваться основы компиляции и работа с IDE.

После этого следует пройтись по основным источникам информации:

- ◆ [31.12] — основы языка;
- ◆ [31.5] — библиотеки;
- ◆ [31.15] — шаблоны.

Затем вам нужно овладеть концептуальным языком:

- ◆ начните с книг [31.10, 31.8, 31.9, 31.11] — именно в таком порядке;
- ◆ работы [31.3, 31.7, 31.2, 31.13 (возможны и ее предыдущие издания)];
- ◆ [31.14] — здесь описаны некоторые особенности языка C;
- ◆ [31.1] — начала шаблонного метапрограммирования, из которого произошли многие стандарты C++

### 31.3. Советы по дополнительной подготовке

- ◆ Поищите интересные книги, которые я еще не прочитал, — например, [31.5 и 31.6, а также все ее последующие тома). Стоит ли добавить какие-то из них к приведенному в предыдущем разделе списку?

## 31.4. Список рекомендуемой литературы

- 31.1. Alexandrescu A. (2001). Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley.
- 31.2. Coplien J. O. (1992). Advanced C++ Programming Styles and Idioms. Addison-Wesley.
- 31.3. Dewhurst S. C. (2005). C++ Common Knowledge: Essential Intermediate Programming. Pearson.
- 31.4. Josuttis N. M. (2012). The C++ Standard library: A Tutorial and Reference. Addison-Wesley.
- 31.5. Josuttis N. M. (2019). C++17: Complete Guide. Independently Published.
- 31.6. Lakos J. (2019). Large-Scale C++ Volume I: Process and Architecture. Addison-Wesley.
- 31.7. Lippman S. B. (1996). Inside the C++ Object Model. Addison-Wesley.
- 31.8. Meyers S. (1996). Effective C++: 35 New Ways to Improve Your Programs and Designs. Addison-Wesley.
- 31.9. Meyers S. (2001). Effective STL: 50 Specific Ways to Improve Your Standard Template Library. Addison-Wesley.
- 31.10. Meyers S. (2005). Effective C++: 55 Specific Ways to Improve Your Programs and Designs. Addison-Wesley.
- 31.11. Meyers S. (2014). Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. Addison-Wesley.
- 31.12. Stroustrup B. (2013). The C++ programming language. Addison-Wesley.
- 31.13. Sutter H. (2005). Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions. Boston, MA: Addison-Wesley.
- 31.14. van der Linden P. (1994). Expert C programming: Deep C secrets. Pearson.
- 31.15. Vandevoorde D., Josuttis N. M. & Gregor D. (2017). C++ Templates: The Complete Guide. Addison-Wesley.

# Предметный указатель

## A

Agile-разработка 46  
Asymptotic Relative Efficiency, ARE 472  
Autoregression, AR 553

## C

Central Limit Theorem, CLT 473  
Chaos Monkey 54  
Conjugate Gradient, CG 633

## D

Design of Experiments, DOE 541

## E

Empirical Distribution Function, EDF 462  
ETF, Exchange traded fund 567  
Exploratory Data Analysis, EDA 575

## F

False Discovery Rate, FDR 518  
Family-Wise Error Rate, FWER 517

## L

Latin Hypercube Sampling, LHS 543

## M

Markov Chain Monte Carlo, MCMC 544  
Mean-Squared Error, MSE 464  
Minimum Description Length, MDL 463

## N

Normalized Median Absolute Deviation,  
MADN 486

## P

Partial Autocorrelation Function, PACF 558  
Probability Density Function, PDF 118, 463

## R

Random Walk Metropolis, RWM 545  
Randomized Quasi Monte Carlo, RQMC 539

## S

Scrum 46  
SVD, Singular Value Decomposition 619  
Symmetric and Positive Definite, SPSP 633

## U

UMA Unbiased, UMAU 477  
Uniformly Most Accurate, UMA 476

---

## A

Автоматическая регрессия 54  
Авторегрессия (Autoregression, AR) 553  
Авторское право 74  
Адаптивное интегрирование 676  
Аксиомы  
◊ Клейнберга 944  
◊ Колмогорова 118

## Алгоритм 21

◊ Алгоритм A\* 355  
◊ Алгоритм AES 454  
◊ Алгоритм BFGS 743  
◊ Алгоритм BWT 335  
◊ Алгоритм BZIP2 335  
◊ Алгоритм DFS 298  
◊ Алгоритм GZIP 336

Алгоритм (*прод.*)

- ◊ Алгоритм HashQ 294, 314
- ◊ Алгоритм Horspool 314
- ◊ Алгоритм KSort 307
- ◊ Алгоритм L-BFGS 744
- ◊ Алгоритм MTFT 332, 336
- ◊ Алгоритм RC4 125
- ◊ Алгоритм realtime A\* 352
- ◊ Алгоритм RLE 336
- ◊ Алгоритм RSA 456
- ◊ Алгоритм WMMM 302
- ◊ Алгоритм Априори 974
- ◊ Алгоритм Беллмана — Форда 242
- ◊ Алгоритм Берлекэмпа — Мессе 450
- ◊ Алгоритм Блюстейна 646
- ◊ Алгоритм ветвления и границ (Branch and Bound, B&B) 346
- ◊ Алгоритм Бу — Манбера 296
- ◊ Алгоритм Гаусса — Томаса 611
- ◊ Алгоритм Гейла — Шепли 248
- ◊ Алгоритм Глушкова 297
- ◊ Алгоритм Дейкстры 241, 251
- ◊ Алгоритм Евклида 406, 441
- ◊ Алгоритм Кленшоу 657
- ◊ Алгоритм кратчайших путей Дейкстры 224
- ◊ Алгоритм Крускала 251
- ◊ Алгоритм Лас-Вегаса 29
- ◊ Алгоритм Миллера — Рабина 407
- ◊ Алгоритм Монте-Карло 29
- ◊ Алгоритм Нелдера — Мида 748
- ◊ Алгоритм повторного локального поиска (Iterated Local Search, ILS) 375
- ◊ Алгоритм поиска по компасу 752
- ◊ Алгоритм Прима 37, 239
- ◊ Алгоритм рекурсивного поиска по первому наилучшему (Recursive best-first search, RBFS) 357
- ◊ Алгоритм Роббинса — Манро 773
- ◊ Алгоритм Салтелли 536
- ◊ Алгоритм случайного блуждания по Метрополису (Random Walk Metropolis, RWM) 545
- ◊ Алгоритм сопряженных градиентов (Conjugate Gradient, CG) 633
- ◊ Алгоритм стохастической аппроксимации возмущения 774
- ◊ Алгоритм Хамерли 966
- ◊ Алгоритм Хорспула 294, 315
- Альфа-элемент 434

## Аппроксимация

- ◊ выборочного среднего 772
- ◊ пути выборки 772
- Арифметика с бинарными полиномами 429
- Арифметические коды 336
- Архитектура системы 47
- Асимптотическая относительная эффективность (Asymptotic Relative Efficiency, ARE) 472
- Атака «человек посередине» 456

**Б**

- Байтовый код 322
- Бинарное дерево 105
- Битовая маска 107
- Битовый поток 318
- Брекетинг 702
- Бритва Оккама 802
- Быстрая сортировка с двумя опорными точками 162
- Быстрое преобразование Фурье (БПФ) 645
- Быстрый выбор 159
- ◊ с несколькими ключами 159

**В**

- Ведерная очередь 994
- Вектор
- ◊ векторов 232
- ◊ признаков 791
- Вера 25
- Вероятность 118
- Виды
- ◊ алфавита 293
- ◊ графов 231
- Вихрь Мерсенна 123
- Временная логика 25
- Выборка латинского гиперкуба (Latin Hypercube Sampling, LHS) 543
- Выпадающее значение 976
- Выпадающие данные 485
- Выпуклая оболочка 425

**Г**

- Гамма-код 320
- Гауссова квадратура 723
- Генераторы псевдослучайных чисел 120
- Генетический локальный поиск 766

Гипотеза  
◇ коллектора 802  
◇ Римана 408  
Гистограмма 976  
Градиентный спуск 742  
Гребневая регрессия 938

## Д

Двоичный поиск 161  
Двоичный симметричный канал (Binary Symmetric Channel, BSC) 432  
Двойное хеширование 218  
Двусвязный список 94  
Двусторонняя куча 229  
Двусторонняя очередь 103  
Дедек 39  
Дерево 105  
◇ LOUDS 313  
◇ VP 315  
Динамическая отсортированная последовательность 164  
Дисбаланс классов 844  
Дискретное косинусное преобразование (ДКП) типа I 648  
Дискретное преобразование Фурье (ДПФ) 645  
Дисперсия 120  
Диспетчер 49  
Дифференциальная эволюция 764  
Добавление элементов в конец 90  
Доктрина  
◇ добросовестного использования 75  
◇ первой продажи 75

## Е

Евклидово расстояние 412

## Ж

Жесткий диск 265

## З

Задача  
◇ выполнимости (Satisfiability Problem, SAT) 342  
◇ коммивояжера (Traveling Salesman Problem, TSP) 340  
◇ максимальной выполнимости 342  
◇ многоугольного бандита 971

◇ о максимальном потоке 243  
◇ о потоке с минимальной стоимостью 244  
◇ о черном лебеде 531  
◇ об укладке рюкзака (knapsack problem) 341  
◇ остановки 338  
◇ упаковки контейнера (bin packing problem) 343  
◇ целочисленного программирования (integer programming problem) 343  
Задачи принятия решений 339  
Закон Хайрама 58  
Запрос частичного соответствия 420  
Знак сертификации 75

## И

Иерархическая кластеризация 968  
Инвариант 26  
Инверсии 150  
Индекс  
◇ Рэнда 946  
◇ чувствительности Соболя 535  
Интеграционное тестирование 54  
Интегрирование Монте-Карло 683  
Интеллектуальный указатель 52  
Интервал Вальда 475  
Интервальная арифметика 636  
Интервальный запрос 420  
Исключительная лицензия 78  
Исследовательский анализ данных (Exploratory Data Analysis, EDA) 575

## К

Квадратичное зондирование 218  
Квадратура  
◇ Гаусса — Радау 716  
◇ Кленшоу — Кертиса 658, 676  
Класс моделей 810  
Классификация 842  
Классы сложности задач 338  
Клевета/диффамация 79  
Код  
◇ Bigna 336  
◇ Golomb 336  
◇ Tunstall 336  
◇ Рида — Соломона (Reed-Solomon, RS) 438  
◇ Фибоначчи 321, 336  
Кодирование 320  
Коды Хаффмана 324

Коллективный знак 75  
 Коллизии 196, 206, 211, 215, 218, 219  
 Коммерческая тайна 73  
 Компараторы 84  
 Конвейер преобразования 49  
 Контракт 76  
 ◇ алгоритма 25  
 Конфиденциальные отношения 73  
 Корзина 818  
 Корреляция  
 ◇ Кендалла 490  
 ◇ Пирсона 488  
 ◇ Спирмена 489  
 Коэффициент загрузки 206  
 Критерий сбалансированной частоты  
   ошибок 844  
 Кубические сплайны 668  
 Кукушкино хеширование 218  
 Куча  
 ◇ min-max 229  
 ◇ спаривания 991  
 ◇ Фибоначчи 229

## Л

Лавина 203  
 Лассо 926  
 Лексикографическое сравнение 85  
 Лексический анализ 310  
 Лемма удвоения 306  
 Линейная интерполяция 662  
 Линейное зондирование 218  
 Линейный дискриминантный анализ, LDA 913  
 Линейный конгруэнтный генератор (Linear Congruential Generator, LCG) 121  
 Лицензирование 78  
 Лицензия 22, 71  
 Логика  
 ◇ второго порядка 24  
 ◇ высказываний 24  
 ◇ знания 24  
 ◇ первого порядка 24  
 Локально-чувствительное хеширование (Locality-Sensitive Hashing, LSH) 427

## М

Марковская цепь Монте-Карло (Markov Chain Monte Carlo, MCMC) 544  
 Массив  
 ◇ LCP 314  
 ◇ динамического размера 88

◇ смежности 232  
 ◇ суффиксов 306  
 Массивы 82  
 Математическое ожидание 120  
 Машина Тьюринга 82  
 Машинная модель 27  
 Медоиды 957  
 Метамета модель 941  
 Мета модель 940  
 Метаэвристика 365  
 Метод  
 ◇ Беллмана — Форда 250  
 ◇ ближайшего соседа 858  
 ◇ Бройдена 694  
 ◇ возврата 738  
 ◇ Ньютона 405, 693, 742  
 ◇ Ньютона — Крылова 724  
 ◇ один против всех, OVA 848  
 ◇ один против одного, OVO 849  
 ◇ толстых узлов 254  
 ◇ удержания 797  
 Методика «ранжирования/  
   деранжирования» 262  
 Методы  
 ◇ push-relabel 251  
 ◇ социальной инженерии 455  
 Методы-обертки 824  
 Механизм опорных векторов, SVM 866  
 Минимаксная оценка 470  
 Минимальная длина описания (Minimum Description Length, MDL) 463  
 Минимальное остовное дерево (Minimum Spanning Tree, MST) 239  
 Минимизация  
 ◇ риска Оккама, ORM 801  
 ◇ структурного риска, SRM 800  
 ◇ эмпирического риска, ERM 800  
 Многомерная куча 230  
 Множественный рекурсивный генератор (Multiple Recursive Generator, MRG) 123  
 Моделирование 320  
 Модель 810  
 ◇ мешка слов 830  
 Модульное тестирование 54

## Н

Набор битов 109  
 Наименьшая медиана квадратов, LMS 938  
 Накопление округлений 585  
 Направление спуска 738

Наследование 48  
Независимые потоки 121  
Неправомерное действие 73  
Непрерывные распределения 133  
Неприводимый многочлен 430  
Неравенства  
◊ Крамера — Рао 466  
◊ Бонферрони 517  
◊ братьев Марковых 655  
◊ Дворецкого — Кифера — Вольфовица (DKW) 491  
◊ Коксмы — Хлавки 537  
◊ Крафта 320  
◊ Хёффдинга 480  
Норма Фробениуса 603  
Нормализованное среднее абсолютное отклонение (Normalized Median Absolute Deviation, MADN) 486

## О

Обобщенная линейная модель, GLM 940  
Общая аддитивная модель, GAM 940  
Ограничение AllDifferent 387  
Ограниченная машина Больцмана, RBM 916  
Одноразовый код 48, 453  
Одношаговые методы 710  
Окрестность решения 365  
Операция ввода/вывода 273  
Операция изменения ключа 224  
Оптимальный механизм обучения Байеса 799  
Ортогонализация Грама — Шмидта 591  
Основная теорема 37  
Открытая адресация 218  
Относительный совокупный коэффициент риска 472  
Оценщик плотности ядра, KDE 976  
Оценщики 461  
Очередь 102  
◊ Бродала 229  
◊ с корзиной 229  
◊ с приоритетом 221  
Ошибка удвоения 675

## П

Парадокс Симпсона 535, 926  
Параллелизм 38  
Параметр 810  
Патент 72

Перестроение структуры 253  
Пирамидальная сортировка 162  
Площадь под ROC-кривой (AUC) 910  
Поведенческие собеседования 62  
Погрешность точности 591  
Подзадача оптимизации Нелдера — Мида 65  
Подмодели 810  
Подпространство Крылова 633  
Поиск  
◊ в глубину (depth-first search, DFS) 235  
◊ в ширину (breadth-first search, BFS) 236  
◊ золотого сечения 732  
◊ по сетке 770  
Поле Галуа 434  
Полиномы Чебышева 655  
Политика «последнее недавно использованное» (Last Recently Used, LRU) 255  
Полный сумматор 399  
Полупараметрические модели регрессии 940  
Последовательный поиск 161  
Постоянный код 48  
Потери по Чебышеву 479  
Потоковый шифр RC4 453  
Поточный шифр 453  
Правило  
◊ Гаусса — Лобатто 679  
◊ Гаусса — Лобатто — Кронрода 677  
◊ единственного определения 83  
◊ Лайнесса 723  
◊ Лопиталья 27  
◊ Симпсона 676  
◊ трапеций 675, 714  
Предиктор 810  
Предложение 76  
Предобуславливатели 633  
Преобразование Берроуза — Уилера (Burrows — Wheeler Transform, BWT) 307, 333  
Прерывистый поиск 763  
Префиксные коды 320  
Примитивный многочлен 430  
Принцип минимальной длины описания, MDL 812  
Проблема  
◊ национального флага Нидерландов 162  
◊ непересекающихся множеств 115  
Прогнозы 791  
Произведение непересекающихся циклов 158  
Проклятие размерности 802



Пространство признаков 791  
 Протоколы связи 452  
 Процедура Холма 520  
 Прямой поиск 825  
 Псевдоразмерность 924  
 Пузырьковая сортировка 162

## Р

Равновесие Нэша 29  
 Разделение 150  
 Разложение по сингулярным значениям  
 (Singular Value Decomposition, SVD) 619  
 Рандомизация 118  
 Рандомизированный квази-Монте-Карло  
 (Randomized Quasi Monte Carlo, RQMC)  
 539  
 Распределение Пуассона 211  
 Расстояние  
 ◇ Ван Донгена 965  
 ◇ по иерархии 413  
 ◇ Хэмминга 431  
 Расширение  
 ◇ Кронрода 677  
 ◇ обратного указателя 192  
 Расширенный шаблон 299  
 Регрессия 922  
 ◇ гребня ядра 939  
 ◇ опорного вектора ядра, SVR 938  
 Регулярное выражение 297  
 Решето Эратосфена 259, 408

## С

Сбалансированная точность 844  
 Свидетели-эксперты 80  
 Свободный список 97  
 Свойства Xorshift 122  
 Сервисный знак 75  
 Сериализация 268  
 Сжатие с потерями 337  
 Симметричные и положительно  
 определенные матрицы (Symmetric and  
 Positive Definite, SPSD) 633  
 Синтаксическое дерево 310  
 Система C++ STL 30  
 Скорректированный индекс Рэнда 946  
 Сложность по Колмогорову 317  
 Собственность 71  
 Совокупная функция распределения (СФР)  
 118

Соглашение  
 ◇ о неконкуренции 79  
 ◇ о неприглашении 79  
 ◇ о неразглашении 78  
 Сокращение Хаусхолдера 612  
 Сортировка  
 ◇ выбором 162  
 ◇ подсчетом 154  
 ◇ с подсчетом по ключу (KSort) 155  
 Спаренная куча 229  
 Сплайны многомерной адаптивной  
 регрессии, MARS 939  
 Среднеквадратичная ошибка (Mean-Squared  
 Error, MSE) 464  
 Среднеквадратичная ошибка, СКО 922  
 Стандарт 76  
 Стори-пойнты 58  
 Стохастический градиентный спуск 773  
 Стражи 35  
 Страницы 33  
 Строка 293  
 Структура  
 ◇ данных 21  
 ◇ с выбором ранга 310  
 ◇ краткая 310  
 Сумма квадратов ошибок, SSE 931  
 Сценарии использования 46

## Т

Таблица истинности 24  
 Табличная хеш-функция 205  
 Табу-поиск 392  
 Теорема  
 ◇ Берри — Эссена 473  
 ◇ Вейерштрасса 650  
 ◇ Джексона 651  
 ◇ Зутендейка 738  
 ◇ кодирования зашумленного канала 432  
 ◇ Лагранжа 200  
 ◇ Лагранжа об ошибке интерполяции (LIE)  
 650  
 ◇ Лемана — Шеффе 469  
 ◇ народной стоимости 899  
 ◇ об отсутствии бесплатных обедов (No  
 Free Lunch theorem, NFL) 390, 804, 853  
 ◇ Осумы о разложении 874  
 ◇ присяжных 889  
 ◇ Тейлора 588  
 ◇ Ферма 407  
 ◇ Фубини 681  
 ◇ Эрдоса — Вертези 650

Техника STAR 62  
Типы подстрок 293  
Товарный знак 75  
Топологическая сортировка 238  
Трансферное обучение 978

## У

Удвоение  
◇ массива 90  
◇ по Тьюки 1001  
Указатель  
◇ на идиому реализации 50  
◇ получения 265  
◇ размещения 265  
Умножение Карацубы 410  
Унарный код 320  
Уникальный токен завершения 51  
Упорядоченное дерево 105  
Упрощенный силуэт 949  
Условия Каруша — Куна — Таккера (ККТ) 780  
Утилита UNIX diff 315

## Ф

Фаззинг 55  
Файл 265  
Факторизация Холецкого 139  
Формула  
◇ барицентрической интерполяции 652  
◇ центральной разности второго порядка 690  
◇ Ланса — Уильямса 1007  
Функция  
◇ Боба Дженкинса 218  
◇ плотности вероятности (ФПВ) 118  
◇ потерь 791  
◇ Рунге 652  
◇ частичной автокорреляции (Partial Autocorrelation Function, PACF) 558

## Х

Хвостовая рекурсия 36  
Хеширование  
◇ Zobrista 205  
◇ с учетом местоположения, LSH 911

Хеш-функция 196  
◇ FNV 203  
◇ Xorshift 204  
Хранилище 49

## Ц

Центральная предельная теорема (Central Limit Theorem, CLT) 473  
Цикл Ланцоша 634  
Циклический избыточный код (CRC) 453  
Циклический список 94

## Ч

Частота  
◇ ложных открытий (False Discovery Rate, FDR) 518  
◇ ошибок в семействе (Family-Wise Error Rate, FWER) 517

## Ш

Шаблон сдвиг-и (shift-and) 299  
Шаблоны 48  
Шрифт 81  
Шум подсказок 802

## Э

Эвристика  
◇ ОСБА (Optimal Computing Budget Allocation) 550  
◇ имитации отжига (Simulated Annealing, SA) 370  
◇ максимальной нарушающей пары 876  
Экономическая полезность 338  
Экспоненциальный поиск 161, 702  
Эмпирическая функция распределения (Empirical Distribution Function, EDF) 462  
Эмпирический риск 797  
Энтропия первого порядка 317  
Эпсилон-переходы 297

## Я

Ядерная регрессия Надарая — Ватсона 939