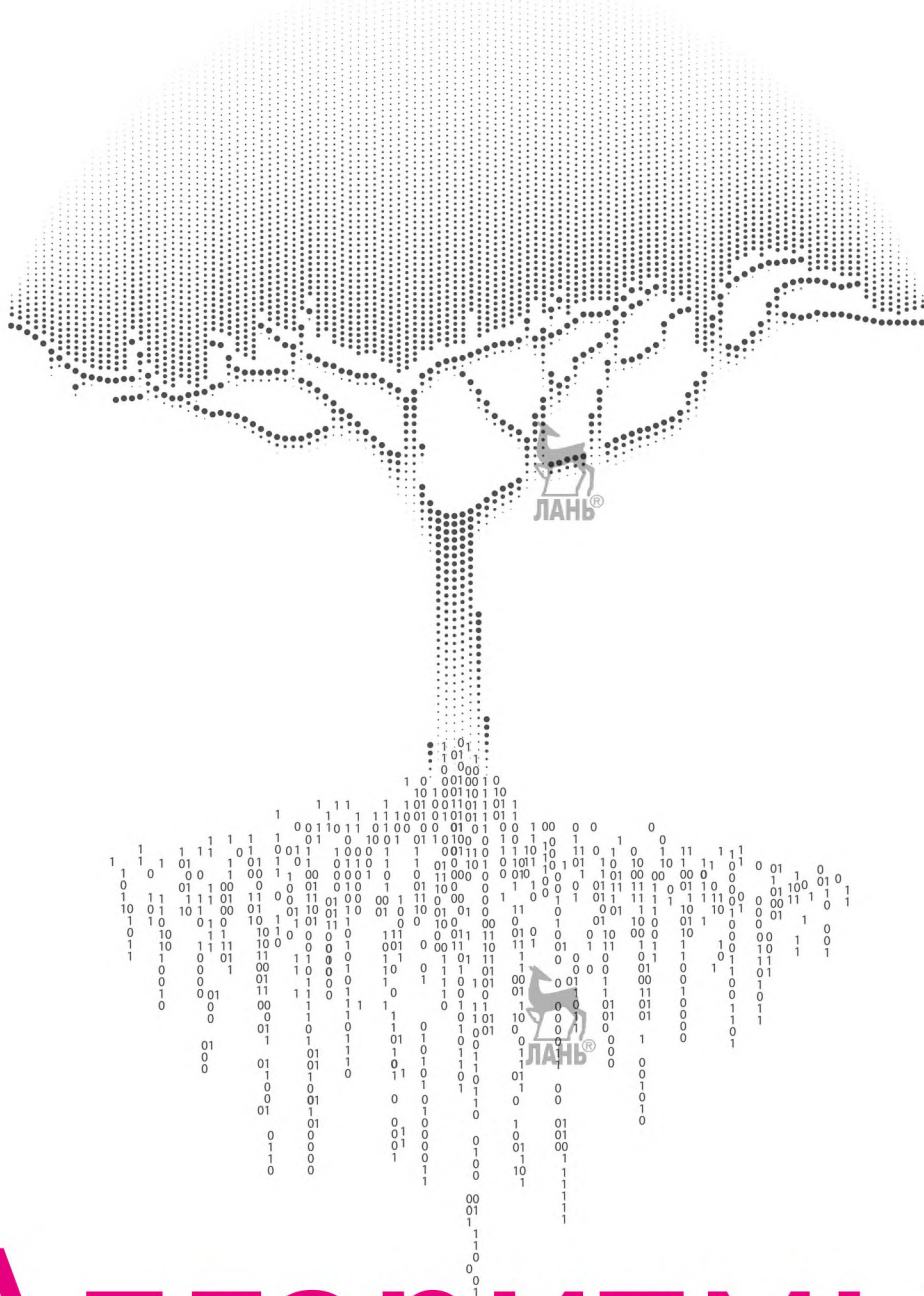

Джефф Эриксон



Алгоритмы

Алгоритмы — это источник жизненной силы информатики. Это механизмы, которые формируют доказательства, и музыка, которую исполняют программы. История алгоритмов стара, как сама математика.

Данная книга представляет собой обширный и оригинальный трактат по разработке и анализу алгоритмов, охватывающий несколько фундаментальных методов с особым акцентом на интуитивное понимание и процесс решения задач.

Издание содержит важные классические примеры, сотни проверенных на практике упражнений, а также много интересных исторических отступлений.



Джефф Эриксон — профессор информатики в Университете Иллинойса, Урбана-Шампейн; книга основана на лекциях по курсу «Алгоритмы», которые он читал в этом университете с 1998 г. Его исследовательские интересы лежат на стыке разработки алгоритмов и дискретной математики, особенно для задач, связанных с геометрией, топологией, графами и оптимизацией.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК "Галактика"
books@aliens-kniga.ru



ISBN 978-5-97060-981-1

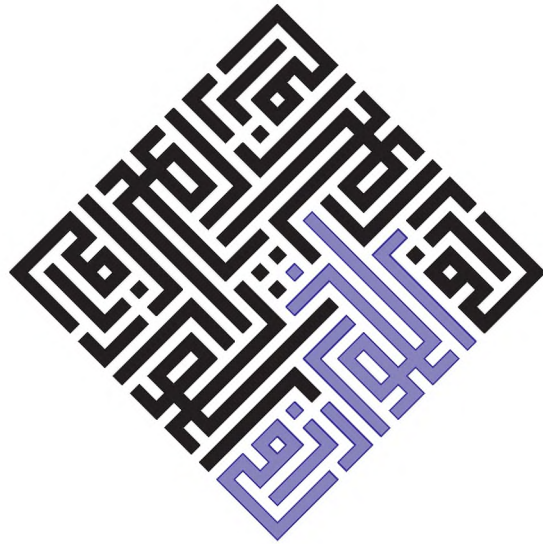


9 785970 609811 >

Джефф Эриксон

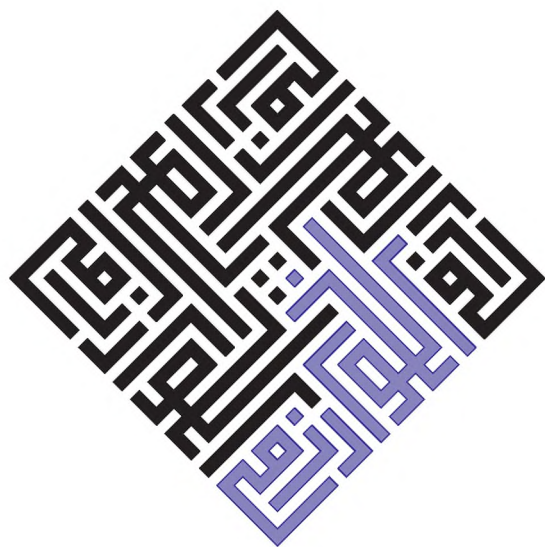
Алгоритмы

Algorithms



Jeff Erickson

Алгоритмы



Джефф Эриксон



Москва 2023

УДК 004.021
ББК 32.973.05
Э77

Э77 Джефф Эриксон

Алгоритмы / пер. с англ. А. В. Снастина, П. Б. Иванова. – М.: ДМК Пресс, 2023. – 526 с.: ил.

ISBN 978-5-97060-981-1

В этом руководстве содержатся основные сведения об алгоритмах: анализируются различные типы алгоритмов, рассматриваются методы их построения (рекурсия, динамическое программирование и др.), приводятся практические примеры. В конце каждой главы приводятся упражнения, направленные на закрепление пройденного.

Для изучения материала требуется знание основ дискретной математики и методов доказательств, а также представление об основных вычислительных задачах и алгоритмах. Желателен практический опыт работы с языком программирования, поддерживающим косвенную адресацию и рекурсию.

Издание адресовано студентам и преподавателям технических вузов, а также тем, кто хочет изучить основы алгоритмизации.

УДК 004.021
ББК 32.973.05

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-79264-483-2 (англ.)
ISBN 978-5-97060-981-1 (рус.)

© Copyright Jeff Erickson, 2019
© Оформление, перевод на русский язык,
издание, ДМК Пресс, 2023

*Ким, Кей и Ханне
с любовью и восхищением.*

*И Ирин
с благодарностью за нарушение ее обещания.*

Оглавление

https://t.me/it_boooks/2

Предисловие от издательства	12
Предисловие.....	13
Об этой книге.....	13
Обязательный минимум.....	13
Дополнительные ссылки	15
Упражнения в этой книге.....	17
Стащите эту книгу.....	18
Благодарности	19
Предостережение для преподавателя.....	20
Глава 0. Введение.....	22
0.1. Что такое алгоритм.....	22
0.2. Умножение	24
Умножение методом решетки	24
Удваивание и усреднение.....	27
Циркуль и линейка	29
0.3. Распределение мест в Конгрессе США.....	30
0.4. Отрицательный пример.....	32
0.5. Описание алгоритмов	33
Определение конкретной задачи.....	34
Описание алгоритма	35
0.6. Анализ алгоритмов	37
Корректность.....	37
Время выполнения.....	37
Упражнения	40
Глава 1. Рекурсия.....	45
1.1. Сведёние.....	45
1.2. Упрощение и делегирование	46
1.3. Ханойские башни.....	48
1.4. Сортировка слиянием.....	51
Корректность	52
Анализ	53
1.5. Быстрая сортировка	54
Корректность.....	55
Анализ	55
1.6. Шаблон.....	57
1.7. Рекурсивные деревья	57
♥Исключение нижних и верхних границ – это правильный подход, даю честное слово.....	60

♥1.8. Линейный алгоритм выбора.....	62
Алгоритм быстрого выбора	62
Правильные опорные элементы	63
Анализ	64
Проверка достоверности.....	66
1.9. Быстрое умножение	67
1.10. Возведение в степень	70
Упражнения	72
Ханойские башни	72
Рекурсивные деревья	77
Сортировка	78
Выбор.....	82
Арифметика.....	85
Массивы	89
Деревья.....	95
Глава 2. Поиск с возвратом.....	102
2.1. Задача об n ферзях	103
2.2. Деревья игры	105
2.3. Задача о сумме подмножеств	108
Корректность	109
Анализ	109
Варианты.....	110
2.4. Общий шаблон	111
2.5. Сегментация текста (Interpunctio Verborum).....	113
2.6. Максимальная возрастающая подпоследовательность.....	120
2.7. Максимальная возрастающая подпоследовательность, дубль 2	124
2.8. Оптимальные двоичные деревья поиска.....	126
Упражнения	129
Глава 3. Динамическое программирование	134
3.1. Mātrāvṛtta	134
Алгоритм поиска с возвратом может быть медленным.....	136
Мемоизация (запоминание): помнить все.....	137
Динамическое программирование: осмысленное заполнение.....	138
И все же не следует запоминать все подряд	140
♥3.2. Небольшое отступление: еще более быстрое определение чисел Фибоначчи	140
Стоп! Не так быстро	142
3.3. Interpunctio verborum redux (И снова о пунктуации)	143
3.4. Шаблон: интеллектуальная рекурсия	144
3.5. Внимание: жадность – это глупость.....	146
3.6. Максимальная возрастающая подпоследовательность.....	147
Первое рекуррентное выражение: кто следующий?	147

Второе рекуррентное выражение: что дальше?	149
3.7. Расстояние редактирования	150
Рекурсивная структура	151
Рекуррентное выражение	152
Динамическое программирование	153
3.8. Задача о сумме подмножеств	155
3.9. Оптимальные двоичные деревья поиска	157
3.10. Динамическое программирование для деревьев	161
Упражнения	163
Последовательности/Массивы	164
Разделение последовательностей/массивов	185
Деревья и поддеревья	197
Глава 4. Жадные алгоритмы	205
4.1. Сохранение файлов на магнитной ленте	205
4.2. Планирование учебных курсов	208
4.3. Общий шаблон	211
4.4. Коды Хаффмана	212
4.5. Задача о стабильных браках	218
Некоторые неудачные идеи	219
Алгоритмы Boston Pool и Гэйла–Шепли	221
Время выполнения	223
Корректность	223
Оптимальность	224
Упражнения	225
Глава 5. Основные графовые алгоритмы	238
5.1. Введение и историческая справка	238
5.2. Основные определения	242
5.3. Представления и примеры	244
5.4. Структуры данных	248
Списки смежных вершин	248
Матрицы смежности	249
Сравнение	250
5.5. Поиск в любом направлении	252
Анализ	255
5.6. Важные варианты	255
Стек: поиск в глубину	255
Очередь: поиск в ширину	255
Очередь с приоритетами: поиск по первому наилучшему совпадению	256
Несвязные графы	257
Направленные графы	259
5.7. Редукция графа: сплошная заливка	259

Упражнения	261
Графы	261
Алгоритмы обхода	263
Сведения	267
Глава 6. Поиск в глубину	281
6.1. Обход в прямом и обратном порядке	283
Классификация вершин и ребер	284
6.2. Обнаружение циклов	287
6.3. Топологическая сортировка	288
Неявная топологическая сортировка	289
6.4. Мемоизация и динамическое программирование	291
Динамическое программирование в НАГ	292
6.5. Сильная связность	294
6.6. Сильные компоненты за линейное время	295
Алгоритм Косараджу–Шарира	296
♥Алгоритм Тарьяна	298
Упражнения	301
Поиск в глубину, топологическая сортировка и сильные компоненты	301
Динамическое программирование	308
Глава 7. Минимальные остовные деревья	316
7.1. Различные веса ребер	317
7.2. Единственный алгоритм минимального остовного дерева	318
7.3. Алгоритм Борувки	320
Это тот самый алгоритм МОД, который вам нужен	322
7.4. Алгоритм Ярника (Прима)	323
♥Улучшенный алгоритм Ярника	324
7.5. Алгоритм Краскала	326
Упражнения	328
Глава 8. Кратчайшие пути	334
8.1. Деревья кратчайшего пути	335
♥8.2. Отрицательные ребра	336
8.3. Единственный алгоритм SSSP	337
8.4. Невзвешенные графы: поиск в ширину	340
8.5. Направленный ациклический граф: поиск в глубину	344
8.6. Поиск по первому наилучшему совпадению: алгоритм Дейкстры	347
Отсутствие отрицательных ребер	348
♥Отрицательные ребра	352
8.7. Ослабление напряжения всех ребер: алгоритм Беллмана–Форда	354
Улучшение Мура	356
Формулировка с использованием динамического программирования	358
Упражнения	361

Глава 9. Кратчайшие пути между всеми парами вершин в графе	374
9.1. Введение.....	374
9.2. Множество отдельных источников	375
9.3. Изменение весов.....	376
9.4. Алгоритм Джонсона.....	377
9.5. Динамическое программирование	378
9.6. Разделяй и властвуй.....	380
9.7. Странное умножение матриц	382
9.8. Алгоритм (Клини – Роя –) Флойда – Уоршелла (– Ингермана)	383
Упражнения	386
Глава 10. Максимальные потоки и наименьшие разрезы	393
10.1. Потоки	394
10.2. Разрезы.....	396
10.3. Теорема о максимальном потоке и наименьшем разрезе (Maxflow-Mincut)	397
10.4. Алгоритм увеличивающего пути Форда и Фалкерсона	400
♥ Иррациональные пропускные способности	401
10.5. Объединения и разбиения потоков	403
10.6. Алгоритмы Эдмондса и Карпа.....	407
Самые насыщенные увеличивающие пути	407
Кратчайшие увеличивающие пути	409
10.7. Дальнейшее развитие	411
Упражнения	413
Глава 11. Приложения потоков и разрезов	422
11.1. Реберно-непересекающиеся пути.....	422
11.2. Пропускные способности вершин и вершинно-непересекающиеся пути	423
11.3. Задача о паросочетании в двудольном графе	424
11.4. Выбор кортежа	427
Расписание экзаменов	428
11.5. Покрытия непересекающихся путей	431
Набор минимального преподавательского состава	432
11.6. Алгоритм исключения для бейсбола	434
11.7. Выбор проекта	437
Упражнения	439
Глава 12. NP-трудность.....	452
12.1. Игра, которую невозможно выиграть.....	452
12.2. P против NP.....	454
12.3. NP-трудная, NP-легкая и NP-полная задача	456
♥ 12.4. Формальные определения (HC SVNT DRACONES – Тут [обитают] драконы)	458
12.5. Редукции и задача Sat.....	460

12.6. 3Sat (от CircuitSat).....	462
12.7. Максимальное независимое множество (от 3Sat).....	465
12.8. Общий шаблон.....	467
12.9. Клика и вершинное покрытие (от независимого множества)	469
12.10. Раскраска графа (от 3Sat).....	470
12.11. Гамильтонов цикл.....	473
От вершинного покрытия	474
От 3Sat.....	476
Варианты и расширения.....	478
12.12. Задача о сумме подмножеств (от задачи вершинного покрытия).....	479
Да будет осмрителен выполняющий редукцию!.....	480
12.13. Другие полезные NP-трудные задачи	481
12.14. Выбор правильной задачи	484
12.15. Простой пример из реальной жизни.....	485
♥12.16. Что дальше	489
Полиномиальное пространство.....	489
Экспоненциальное время	491
Все выше и выше!	491
Упражнения	493
Предметный указатель	509
Список алгоритмов на псевдокоде	523

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Предисловие

Начнем с предисловия к книге алгоритмов практической арифметики.

— Иоанн Севильский (Ioannis Hispalensis),
Книга алгоритмов практической арифметики (около 1135 г.)

*Должен ли я объяснять тебе, друг мой, как ты сможешь это понять?
Напиши об этом книгу.*

— Генри Хоум, лорд Камес (Henry Home, Lord Kames) (1696–1782),
в письме сэру Гилберту Эллиотту (Gilbert Eliott)

*Человек всегда ошибается. Он строил множество планов и считал других людей своими помощниками, спорил с некоторыми или со всеми, много ошибался и кое-что сделал;
все это позволило продвинуться немного вперед, но человек всегда ошибается.
Оказалось, что получилось нечто новое, совсем не похожее на то, что он себе представлял.*

— Ральф Уолдо Эмерсон (Ralph Waldo Emerson), Experience, эссе, второй сборник (1844 г.)

То, что я кратко описал выше, является содержанием книги, в которой реализация основной темы с включением подробностей, вероятно, стала бы невозможной; то, что я написал, является вторым или третьим черновым вариантом предварительной версии этой книги.

— Майкл Спивак (Michael Spivak), предисловие к первому изданию
книги «Дифференциальная геометрия», том 1 (1970 г.)

Об этой книге

Основой этой книги стали конспекты лекций, написанных мною для разнообразных курсов по алгоритмам в университете Иллинойса в Урбана-Шампейне (University of Illinois at Urbana-Champaign), где я преподаю примерно раз в год с января 1999 г. Из-за изменений в теоретической программе бакалавриата я существенно переработал основную редакцию этих конспектов лекций в 2016 г. Эта книга состоит из некоторого подмножества переработанных конспектов лекций в основном по материалам основного базового курса и главным образом отображает алгоритмическое содержание новой университетской теоретической программы, обязательной для младших курсов.

Обязательный минимум

Для тех курсов по алгоритмам, которые я преподаю в университете Иллинойса, существуют два весьма важных предварительных требования к

обязательному образовательному минимуму: изучение курса дискретной математики и курса основных структур данных. Таким образом, эта книга, вероятнее всего, не подойдет для большинства студентов как начальный курс по структурам данных и алгоритмам. В частности, я предполагаю, что читатель как минимум хорошо знаком со специальными темами, такими как:

- **дискретная математика:** элементарная алгебра, логарифмические тождества, простейшая теория множеств, алгебра логики, логика первого порядка, множества, функции, эквивалентности (тождества), частичная упорядоченность, модульная арифметика (операции с остатками чисел по модулю), рекурсивные определения, деревья (как абстрактные объекты, а не структуры данных), графы (из вершин и ребер, а не графики функций);
- **методы доказательств:** прямой, косвенный, от противного (контрадикция), исчерпывающий (полный) анализ вариантов и индукция (особенно «строгая» и «структурная» индукция). В главе 0 используется индукция, и во всех случаях, когда в главе $n-1$ используется индукция, она используется и в главе n ;
- **итеративные концепции программирования:** переменные, условные выражения, циклы, записи, косвенная адресация (адреса, указатели, ссылки), подпрограммы, рекурсия. Я не требую свободного владения каким-либо конкретным языком программирования, но предполагаю наличие у читателя практического опыта работы хотя бы с одним языком, который поддерживает косвенную адресацию и рекурсию;
- **основные (базовые) абстрактные типы данных:** скалярные значения, последовательности, векторы, множества, стеки, очереди, отображения/словари, упорядоченные отображения/словари, очереди с приоритетами;
- **основные (базовые) структуры данных:** массивы, связанные списки (односвязные и двусвязные, линейные и кольцевые (циклические)), деревья двоичного поиска, как минимум одна форма сбалансированного дерева двоичного поиска (например, AVL-деревья, красно-черные деревья, декартовы деревья («дуча»)), списки с пропусками или косые деревья), хеш-таблицы, двоичные кучи, а также, что наиболее важно, понимание различий между этим списком и предыдущим;
- **основные вычислительные задачи:** элементарная арифметика, сортировка, поиск, перечисление, обход вершин дерева (прямой (упорядоченный), центрированный (с порядковой выборкой), в обратном порядке (с отложенной выборкой), поиск в ширину и т. д.);
- **основные алгоритмы:** элементарные (арифметические) алгоритмы, последовательный поиск, двоичный (бинарный) поиск, сортировка (выбором, вставкой, слиянием, пирамидальная, быстрая, по-

разрядная (цифровая) и т. д.), поиск в ширину и в глубину в деревьях (как минимум в двоичных), а также, что наиболее важно, понимание различий между этим списком и предыдущим;

- **элементарный анализ алгоритмов:** асимптотическая нотация (o , O , Θ , Ω , ω), преобразование циклов в суммы и рекурсивных вызовов в рекуррентные соотношения, определение (вычисление) оценки простых сумм и рекуррентных (рекурсивных) соотношений;
- **математическая зрелость:** способность к абстракции, к формальным (особенно рекурсивным) определениям и к выполнению доказательств (особенно индуктивных); запись и прослеживание математических доказательств; распознавание и устранение синтаксических, смысловых (семантических) и/или логических элементов, не имеющих никакого смысла.

В этой книге кратко рассматриваются некоторые из перечисленных выше предварительно требуемых тем, когда этого требует контекст, но это в большей степени напоминание, нежели подробное введение. Для более глубокого изучения настоятельно рекомендуются следующие свободно распространяемые материалы:

- Margaret M. Fleck. «Building Blocks for Theoretical Computer Science». Версия 1.3 (January 2013) или более поздняя доступна здесь: <http://mfleck.cs.illinois.edu/building-blocks/>;
- Eric Lehman, F. Thomson Leighton, and Albert R. Meyer. «Mathematics for Computer Science». Версия июня 2018 г. доступна здесь: <https://courses.csail.mit.edu/6.042/spring18/>. (Настоятельно рекомендуется найти самую последнюю версию.);
- Pat Morin. «Open Data Structures». Редакция 0.1G β (январь 2016 г.) или более поздняя доступна здесь: <http://opendatastructures.org/>;
- Don Sheehy. «A Course in Data Structures and Object-Oriented Design». Версия февраля 2019 г. или более поздняя доступна здесь: <https://don-sheehy.github.io/datastructures/>.

Дополнительные ссылки

Рекомендуется не ограничиваться одной из приведенных выше ссылок или каким-либо другим единственным источником. Авторы и читатели приносят собственные точки зрения и перспективные взгляды в любой учебно-просветительский материал. «Универсального преподавателя» для всех обучающихся подобрать невозможно, даже для очень умных студентов. Поиск «своего» автора, который наиболее эффективно вложит собственные знания и восприятие в ваш разум, требует определенных усилий, но эти усилия многократно окупаются в дальнейшем.

Приведенные ниже ссылки представляют собой особо ценные источники знаний и интуитивных представлений с примерами, упражнениями и развивающими заданиями, но этот список нельзя считать абсолютно полным.

- *Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman*. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974. (Я использовал эту книгу как студент в университете Райса – Rice University, потом как студент магистратуры в Калифорнийском университете в Ирвайне – UC Irvine.)
- *Boaz Barak*. Introduction to Theoretical Computer Science. Предварительная редакция книги, последняя редакция в июне 2019 г. (Это не книга по теории ИТ, доставшаяся вам от бабушки, это гораздо более качественный материал, а бесплатный доступ является дополнительным существенным преимуществом.)
- *Thomas Cormen, Charles Leiserson, Ron Rivest, and Cliff Stein*. Introduction to Algorithms, third edition. MIT Press/McGraw-Hill, 2009. (Я использовал первое издание этой книги, когда был ассистентом кафедры в университете Беркли – Berkeley.)
- *Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani*. Algorithms. McGraw-Hill, 2006. (Вероятно, эта книга по содержанию ближе всего к моей, но она значительно менее подробна.)
- *Jeff Edmonds*. How to Think about Algorithms. Cambridge University Press, 2008.
- *Michael R. Garey and David S. Johnson*. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, 1979.
- *Michael T. Goodrich and Roberto Tamassia*. Algorithm Design: Foundations, Analysis, and Internet Examples. John Wiley & Sons, 2002.
- *Jon Kleinberg and Éva Tardos*. Algorithm Design. Addison-Wesley, 2005. Найдите эту книгу в библиотеке, если сможете.
- *Donald Knuth*. The Art of Computer Programming, volumes 1–4A. Addison-Wesley, 1997 and 2011. (Мои родители подарили мне первые три тома на Рождество, когда мне было 14 лет. Увы, до сих пор так и не удалось прочитать эти тома полностью.)
- *Udi Manber*. Introduction to Algorithms: A Creative Approach. Addison-Wesley, 1989. (Я использовал эту книгу, когда был ассистентом кафедры в университете Беркли – Berkeley.)
- *Ian Parberry*. Problems on Algorithms. Prentice-Hall, 1995 (не выпускалась в печатном виде). Книгу можно загрузить (<https://larc.unt.edu/ian/books/free/license.html>) после внесения небольшой суммы пожертвования. Соблюдайте соглашение о нераспространении.
- *Robert Sedgwick and Kevin Wayne*. Algorithms. Addison-Wesley, 2011.
- *Robert Endre Tarjan*. Data Structures and Network Algorithms. SIAM, 1983.
- Записи лекций по моему курсу алгоритмов в университете Беркли, особенно прочитанные Диком Карпом (Dick Karp) и Раймондом Зайделем (Raimund Seidel).

- Записи лекций, слайды, подготовительные материалы, экзаменационные материалы, видеолекции, отчеты об исследованиях, посты в блогах, вопросы и ответы на сайте StackExchange, подкасты и полнофункциональные массовые открытые онлайн-курсы, свободно распространяемые в веб-среде многочисленными учебными заведениями по всему миру.

Упражнения в этой книге

В конце каждой главы вниманию читателя предлагается несколько упражнений. большинство из которых я использовал как минимум один раз для домашних заданий, на семинарах и в лабораторных работах или на экзаменах. Упражнения не упорядочены по возрастанию сложности, но (в общем случае) сгруппированы по общему применяемому методу или по определенной теме. Некоторые задачи помечены символами, описанными ниже:

- символы красного сердечка (карточная масть «черви») ♥ обозначают особенно сложные задачи. Многие из этих задач предлагались студентам на экзаменах на получение степени доктора философии (Ph.D.) в университете Иллинойса. Некоторые (немногие) действительно сложные задачи обозначены увеличенным символом ♥;
- синие ромбики (карточная масть «бубны») ♦ обозначают задачи, для которых требуется изучение материала из следующих глав, но относящиеся к теме текущей главы. Но задачи, для которых требуется знание материала предыдущих глав, никак не помечены. В этой книге, как и в жизни, знания накапливаются постепенно;
- зеленые символы карточной масти «трефы» ♣ обозначают задачи, для которых требуется знание материала, не относящегося к тематике данной книги, например конечные автоматы, линейная алгебра, теория вероятностей или плоские графы. Но такие задачи встречаются редко;
- черные символы карточной масти «пики» ♠ обозначают задачи, требующие значительного объема рутинной работы и/или написания программного кода. Такие задачи также редко встречаются;
- оранжевые звездочки ★ означают, что вы едите «Лаки чармс» (Lucky Charms – детские сухие завтраки в виде глазированных фигурок-талисманов), произведенные до 1998 г., т. е. имеете дело с очень старыми продуктами. Брр.

Эти упражнения предоставляют возможность применить полученные знания на практике, а не являются «задачами ради самих задач». Цель каждого упражнения не решение конкретно сформулированной задачи, а развитие определенного набора навыков и умений или получение практического решения определенного типа задач. Отчасти по этой причине я не привожу решения упражнений, здесь решения не самое важное. Заметьте,

это не «учебное руководство», и если вы сами не можете решить задачу, то, вероятнее всего, вы не должны предлагать ее своим подопечным. Но при этом, возможно, вы сможете найти решения некоторых домашних заданий, которые опубликованы в этом семестре на веб-странице курса, который я читаю. Кроме того, что может помешать вам написать собственное учебное руководство.

Стащите эту книгу

Эта книга опубликована под защитой лицензии Creative Common Licence, которая позволяет использовать, передавать (распространять), адаптировать (изменять) и пересказывать ее содержимое без разрешения автора при условии, что вы указываете ссылку на оригинальный источник. Полная электронная версия книги, доступная свободно (бесплатно), размещена на сайтах, таких как:

- сайт книги: <http://jeffe.cs.illinois.edu/teaching/algorithms/>;
- мнемоническое сокращенное имя: <http://algorithms.wtf>;
- сайт отчетов об ошибках: <https://github.com/jeffgerickson/algorithms>;
- глобальный архив Internet Archive: <https://archive.org/details/Algorithms-Jeff-Erickson>.

На сайте книги также размещено несколько сотен страниц дополнительных конспектов лекций, содержащих материалы, связанные с темой этой книги, а также расширенные и дополненные материалы. Кроме того, на сайте находится почти полный архив последних домашних заданий, задач для экзаменов, семинаров и лабораторных работ и прочие учебные ресурсы. В процессе чтения курса по алгоритмам я пересматриваю, обновляю и иногда исключаю некоторые учебные материалы, поэтому на веб-странице текущего курса можно обнаружить более свежие версии.

Вне зависимости от того, являетесь вы студентом или преподавателем, использование любой части этой книги или какого-либо другого конспекта лекций не только допускается, но и поощряется – без моего специального разрешения, – именно поэтому я разместил все эти материалы на открытых веб-сайтах. Но при цитировании этой книги убедительно прошу указывать ее название или хотя бы сокращенную ссылку на <http://algorithms.wtf>. Это особенно важно, если вы являетесь студентом и используете мои материалы учебного курса как помощь при выполнении домашнего задания. (Уточните подробности у своего преподавателя.)

Но если вы преподаватель, то я настоятельно рекомендую расширить курс с помощью дополнительных материалов, *написанных вами самостоятельно*. Написание собственных учебных материалов улучшит ваши преподавательские навыки, а также представление учебного материала в аудитории, что в свою очередь способствует более эффективному усвоению материала студентами. Кроме того, стимулом к самостоятельной работе послужит разочарование после ознакомления с теми частями книги, кото-

рые вам не понравились. Все книги плохи несовершенны, и эта не является исключением.

Последнее замечание: **рекомендую все, что вы пишете, открыто размещать в свободном доступе в глобальной веб-среде** – не скрывайте это «за семью замками» системы управления обучением или какой-либо другой системы ограничения доступа к информации, – чтобы студенты и преподаватели везде и всюду могли воспользоваться преимуществами ваших достижений и идей. В особенности если вы разрабатываете полезные ресурсы, которые непосредственно дополняют эту книгу, например слайды, видеоматериалы или руководства по решению задач, то убедительно прошу сообщить мне об этом, чтобы я мог добавить ссылки на ваши ресурсы на веб-сайте книги.

Благодарности

Создание этой книги в немалой степени зависело от вклада многочисленных студентов, изучающих курсы по алгоритмам, преподавателей таких курсов и исследователей в этой области. В особенности я безмерно благодарен более чем трем тысячам студентов Иллинойского университета, которые пользовались моими конспектами лекций как основным источником, высказывали полезные (иногда нелюбимые) критические замечания и терпеливо штудировали ранние версии этих конспектов, которые были действительно ужасными. Также благодарю многих коллег и студентов по всему миру, которые использовали мои конспекты в собственных учебных курсах и присылали полезные отзывы и отчеты об ошибках.

Особая благодарность за постоянное двустороннее взаимодействие и вклад (особенно в создание упражнений) моим великолепным ассистентам. Это:

Адितья Рамани (Aditya Ramani), Акаш Гаутам (Akash Gautam), Алекс Штайгер (Alex Steiger), Алина Эне (Alina Ene), Амир Найери (Amir Nayyeri), Аша Ситхарам (Asha Seetharam), Ашиш Вулимири (Ashish Vulimiri), Бен Мозли (Ben Moseley), Брэд Стёрт (Brad Sturt), Брайан Энсинк (Brian Ensink), Чао Сюй (Chao Xu), Чарли Карлсон (Charlie Carlson), Крис Нейнгенген (Chris Neihengen), Коннор Кларк (Connor Clark), Дэн Буллок (Dan Bullok), Дэн Крэнстон (Dan Cranston), Даниэль Хашаби (Daniel Khashabi), Дэвид Моррисон (David Morrison), Экта Манактала (Ekta Manaktala), Эрин Вольф Чемберс (Erin Wolf Chambers), Гейл Штайц (Gail Steitz), Чжо Као (Gio Kao), Грант Чайковский (Grant Czajkowski), Сянь-Чжи Чан (Hsien-Chih Chang), Игор Гаммер (Igor Gammmer), Джейкоб Лопел (Jacob Laurel), Джон Ли (John Lee), Джонатон Фишер (Johnathon Fischer), Цзюньцин Дэн (Junqing Deng), Кент Куанруд (Kent Quanrud), Кевин Миланс (Kevin Milans), Кевин Смолл (Kevin Small), Константинос Койлиарис (Konstantinos Koiliaris), Кайл Фокс (Kyle Fox), Кайл Цзяо (Kyle Jao), Лан Чень (Lan Chen),

Марк Идельман (Mark Idleman), Майкл Бонд (Michael Bond), Митч Харрис (Mitch Harris), Навин Ариважарен (Naveen Arivazhagen), Ник Бахмейер (Nick Bachmair), Ник Херлберт (Nick Hurlburt), Нирман Кумар (Nirman Kumar), Нитиш Корула (Nitish Korula), Патрик Лин (Patrick Lin), Филип Ши (Phillip Shih), Рачит Агравал (Rachit Agarwal), Реза Замани-Насаб (Reza Zamani-Nasab), Риши Талрейя (Rishi Talreja), Роб МакКенн (Rob McCann), Саханд Мозаффари (Sahand Mozaffari), Шалан Накви (Shalan Naqvi), Шрипад Тите (Shripad Thite), Спенсер Гордон (Spencer Gordon), Срихита Ватсавайя (Srihita Vatsavaya), Субро Рой (Subhro Roy), Тана Ваттанавароон (Tana Wattanawaroon), Уманг Матхур (Umang Mathur), Випул Гоял (Vipul Goyal), Ясу Фуракава (Yasu Furakawa) и Йипу Ван (Yipu Wang).

Огромную помощь оказали многочисленные обсуждения с коллегами по факультету Иллинойского университета, среди них Александра Колла (Alexandra Kolla), Синда Хеерен (Cinda Heeren), Эдгар Рамос (Edgar Ramos), Херберт Эдельсбруннер (Herbert Edelsbrunner), Джейсон Зих (Jason Zych), Ким Уиттлси (Kim Whittlesey), Ленни Питт (Lenny Pitt), Мадху Парасарати (Madhu Parasarathy), Махеш Висванатан (Mahesh Viswanathan), Маргарет Флек (Margaret Fleck), Шан-Хуа Тен (Shang-Hua Teng), Стив ЛаВалль (Steve LaValle) и особенно Чандра Чекури (Chandra Chekuri), Эд Рейнголд (Ed Reinhold) и Сариел Хар-Пелед (Sariel Har-Peled).

Разумеется, эта книга обязана своим появлением работе тех людей, которые научили меня использовать алгоритмы во времена студенчества. Это Боб Биксби (Bob Bixby) и Майкл Перлман (Michael Pearlman) в университете Райса; Дэвид Эппштейн (David Eppstein), Дэн Хершберг (Dan Hirschberg) и Джордж Лекер (George Lueker) в университете Ирвайна; Эбхирам Рэнэйд (Abhiram Ranade), Дик Карп (Dick Karp), Мануэл Блум (Manuel Blum), Майк Лаби (Mike Luby) и Раймунд Зайдель (Raimund Seidel) в университете Беркли.

Я позаимствовал общую структуру курса в начальном варианте и способ подробной записи своих конспектов лекций в первую очередь у Херберта Эдельсбруннера (Herbert Edelsbrunner). Идея превращения набора моих конспектов в книгу принадлежит Стиву ЛаВаллю (Steve LaValle), а дизайном некоторых компонентов книги занимался Роберт Грист (Robert Ghrist).

Предостережение для преподавателя

Разумеется, ни один из перечисленных выше людей не должен быть обвинен в каких-либо недостатках написанной мною книги. Несмотря на многочисленные этапы переработки и редактирования, в этой книге имеются некоторые ошибки, «баги», оплошности, упущения, запутанные моменты, нелепости, опечатки, математические, грамматические и логические ошибки, «глюки», плохо спроектированные решения, исторические неточности, анахронизмы, несоответствия, излишние обобщения, неясности,

лишняя болтовня, искажения, чрезмерные упрощения, избыточность, недержание речи, бессмыслица, всякий хлам, халтура, мусор и явная ложь – но все это полностью на совести Стива Скиены (Steve Skiena).

Я сопровождаю систему отслеживания ошибок (баг-трекер) на сайте GitHub: <https://github.com/jeffgerickson/algorithms>, где все читатели, в том числе и вы, можете представлять на рассмотрение отчеты об ошибках, запросы новых свойств и обеспечивать общую обратную связь по теме этой книги. Убедительно прошу сообщать мне обо всех найденных ошибках любого типа – математических, грамматических, исторических, типографических, культурологических и т. п. – как в основном тексте, так и в упражнениях или в других материалах учебного курса. (Вряд ли Стиву есть дело до этого.) Разумеется, приветствуются и любые другие варианты обратной связи.

Желаю успеха!

— Джефф

Обычно автор великодушно принимает на себя вину за все обнаруженные недостатки. Я не принимаю. Любые ошибки, недостатки или проблемы, найденные в этой книге, – это вина кого-то другого, но я был бы рад узнать, кто действительно виноват.

— Стивен С. Скиена (Steven S. Skiena), *The Algorithm Design Manual* (1997 г.)

Вне всякого сомнения, за этим утверждением следует аннотированный список всех книг с объяснением того, почему каждая из них – явная чушь.

— Адам Контини (Adam Contini), *MetaFilter*, 4 января 2010 г.

Глава 0

https://t.me/it_books/2

Введение

Отсюда начинается алгоритм. Эта техника называется алгоритмом, используя который индийцы дважды наслаждаются пятью разными фигурами: 0. 9. 8. 7. 6. 5. 4. 3. 2. 1.
(Hinc incipit algorismus. Haec algorismus ars praesens dicitur in qua talibus indorum fruimur bis quinque figuris 0. 9. 8. 7. 6. 5. 4. 3. 2. 1.)

— Брат Александр де Вилья Деу (Friar Alexander de Villa Dei),
Песнь об алгоритме (Carmen de Algorismo) (около 1220 г.)

Требую от художника сознательного отношения к работе, Вы правы, но Вы смешиваете два понятия: решение вопроса и правильную постановку вопроса.

— А. П. Чехов, в письме А. С. Суворину (27 октября 1888 г.)

Чем больше мы приучаем себя к механическим действиям в малозначимых случаях, тем больше сил мы освобождаем для использования в более важных случаях.

— Анна Бракетт (Anna C. Brackett), The Technique of Rest (1892 г.)

И вот я в 2:30 ночи пишу о методе, несмотря на твердое убеждение, что мгновение, когда мужчина начинает говорить о методе, является доказательством того, что он только что лишился идей.

— Реймонд Чандлер (Raymond Chandler), в письме Эрлу Стенли Гарднеру
(Erle Stanley Gardner) (5 мая 1939 г.)

Хорошему человеку не нужны правила.

Сегодня не тот день, чтобы выяснять, почему я имею так много.

— Доктор [Matt Smith], «Хороший человек идет на войну» (A Good Man Goes to War),
сериал «Доктор Кто» (Doctor Who, 2011 г.)

0.1. Что такое алгоритм

Алгоритм – это явно заданная, точная, однозначная, механически исполняемая последовательность элементарных инструкций, обычно предназначенная для достижения конкретной цели. Например, ниже приведен алгоритм исполнения известной навязчивой песенки «99 бутылок пива на стене» (99 Bottles of Beer on the Wall) для произвольных значений, отличающихся от 99.

BottlesOfBeer(n):For $i \leftarrow n$ down to 1

Петь "i бутылок пива на стене, i бутылок пива,"

Петь "Возьми одну, пусти по кругу, i - 1 бутылок пива на стене."

Петь "Нет бутылок пива на стене, нет бутылок пива,"

Петь "Сбегай в магазин, купи еще, n бутылок пива на стене."

Слово «алгоритм» происходит не от греческого корня *arithmos*, означающего «число», и *algos*, означающего «боль», как могли бы предположить алгоритмофобы-классики. Скорее, это искажение имени персидского ученого IX века Мухаммада ибн Мусы-аль-Хвāризми. Аль-Хвāризми, возможно, наиболее известен как автор трактата «Аль-Китāб аль-мухтасар ф āих āйсāб аль-габр ва'л-мукāбала», от которого происходит современное слово «алгебра». В другом трактате аль-Хвāризми описал современную десятичную систему записи и манипулирования числами (в частности, использование маленького круга или сифра для обозначения недостающего количества), которая была разработана в Индии несколькими столетиями ранее. Методы, описанные в этом последнем трактате, с использованием либо письменных цифр, либо счетных камней, стали известны в английском языке как алгоризм или аугрим, а его цифры стали известны в английском языке как *ciphers* (шифры).

Несмотря на то что и позиционная нотация, и работы аль-Хвāризми были уже известны некоторым европейским ученым, «индийско-арабская» система была популяризирована в Европе средневековым итальянским математиком и торговцем Леонардо Пизанским, более известным как Фибоначчи. Во многом благодаря его книге «*Liber Abaci*»¹, написанной в 1202 г., запись чисел цифровыми символами стала заменять счетные таблицы (известные под названием «абак(а)» (*abacus*)) и «арифметику на пальцах»² как основные предпочитаемые методики вычислений³ в Европе в XIII в. – но не потому, что запись десятичными цифрами была проще для обучения и использования, а потому, что такой способ обеспечивал ведение бухгалтерской книги. Цифры стали широко распространенными в Западной Европе только после появления наборного типографского шрифта,

¹ Несмотря на соблазн перевода названия *Liber Abaci* буквально как «Книга абака», более правильный перевод труда Фибоначчи – «Книга вычислений». И до и после Фибоначчи итальянское слово *abaco* использовалось для описания всего, что связано с вычислениями – устройств, методов, школ, книг и т. д. – во многом так же, как сегодня выражение *computer science* используется в английском языке (или «информационные технологии» в русском языке) или как китайское выражение «исследование операций» переводится буквально как «изучение использования счетных палочек».

² Счет с помощью десяти цифр-пальцев!

³ Слово *calculate* (вычислять) произошло от латинского *calculus*, означающего «небольшой камешек», т. е. камешки на счетной таблице (доске), которые Джеффри Чосер (Chaucer, Geoffrey) называл *augrym stones*. В 440 г. до н. э. Геродот в своей «Истории» писал, что «греки пишут и считают (λογίζεσθαι ψήφοις, дословно: «считать камешками») слева направо, а египтяне делают это наоборот. Но они говорят, что их способ записи направлен вправо, а греческий способ записи направлен влево». (Странно, что Геродот ничего не говорит о том, с какого конца начинают разбивать яйцо египтяне.)

а действительно вездесущими – только при массовом производстве дешевой бумаги в самом начале XIX в.

В итоге слово «алгоритм» (*algorism*) превратилось в современный термин «алгоритм» (*algorithm*) по ложной («вульгарной») этимологии от греческого слова *arithmos* (и, возможно, от ранее упомянутого *algos*)⁴. Таким образом, до недавнего времени термин «алгоритм» относился исключительно к механическим (автоматически выполняемые) методам арифметики с использованием позиционной системы записи арабских цифр. Люди, обученные быстрому и надежному выполнению этих процедур, назывались алгоритмистами (*algorists*) или вычислителями – *computators*, упрощенно *computers*.

0.2. Умножение

Несмотря на то что как предмет в академии алгоритмы существуют лишь несколько десятилетий, они были вместе с нами с самого зарождения цивилизации. Описания пошаговых арифметических вычислений встречаются среди самых ранних образцов письменности, задолго до работ, представленных Фибоначчи и аль-Хорезми и даже до появления позиционной системы записи чисел, которую они внедряли.

Умножение методом решетки

Самый известный метод умножения больших чисел, по крайней мере для американских учащихся, – умножение методом решетки (*lattice algorithm*). Этот алгоритм был популяризирован Фибоначчи и описан в *Liber Abaci*. Фибоначчи узнал об этом алгоритме из арабских источников, в том числе из работ аль-Хорезми, который в свою очередь изучил его по индийским источникам, включая трактат VII в. «*Brāhmasphuṭasiddhānta*» Брахмагупты (*Brahmagupta*), который, возможно, обучался по китайским источникам. Старейшие сохранившиеся описания этого алгоритма встречаются в трактате «Математическая классика» Сунь Цзы, написанном в Китае между III и V вв., а также в комментариях Евтокия Аскалонского к трактату Архимеда «Об измерении круга», написанных около 500 г. н. э., но существуют свидетельства того, что этот алгоритм был известен намного раньше. Евтокий считает, что этот метод появился в последнем трактате Аполлона Пергийского (около 300 г. до н. э.) под названием «*Okytokion*» (Ὀκυτόκιον)⁵. Шумеры

⁴ Некоторые средневековые источники утверждают, что греческая приставка *algo-* означает «искусство» или «введение». В других источниках сообщается, что алгоритмы были изобретены неким греческим философом, или королем Индии, или, возможно, королем Испании с именем *Algos*, или *Algor*, или *Argus*. Кое-кто, возможно, Данте Алигьери даже отождествлял изобретателя алгоритмов с мифологическим греческим кораблестроителем и одноименным аргонавтом (Арго). Совершенно неясно, претендовали ли все эти смехотворные утверждения на историческую точность или являлись лишь плодами богатой фантазии.

⁵ Дословно: «лекарство, способствующее быстрым и легким родам». В папирусе из Александрийской библиотеки воспроизводились некоторые извлечения из «*Okytokion*» приблизительно за 200 лет до Евтокия, но его описание алгоритма умножения методом решетки (если Евтокий действительно приводил его) также утерян.

записывали схему умножения на глиняных табличках еще в 2600 г. до н. э., и можно предположить, что они, вероятно, использовали алгоритм умножения методом решеток⁶.

Алгоритм умножения методом решетки предполагает, что исходные числа представлены как явным образом заданные строки цифр. Здесь я подразумеваю, что вычисления выполняются по основанию 10, но алгоритм легко обобщается для вычислений по любому другому основанию. Для упрощения нотации⁷ входные данные состоят из пар элементов массивов $X[0..m-1]$ и $Y[0..n-1]$, представляющих числа:

$$x = \sum_{i=0}^{m-1} X[i] \cdot 10^i \quad \text{и} \quad y = \sum_{j=0}^{n-1} Y[j] \cdot 10^j.$$

Аналогичным образом выходные данные состоят из одного массива $Z[0..m+n-1]$, представляющего произведение:

$$z = x \cdot y = \sum_{k=0}^{m+n-1} Z[k] \cdot 10^k.$$

Этот алгоритм использует сложение и умножение одnorазрядных чисел как простейшие операции. Сложение можно выполнить с использованием простого цикла `for`. На практике умножение одnorазрядных чисел выполняется с помощью таблицы поиска (преобразования), вырезанной на глиняных табличках, нарисованной на деревянных или бамбуковых дощечках, записанной на бумаге, хранящейся в памяти, доступной только для чтения (ПЗУ), или в памяти вычислителя. Полный алгоритм решеточного умножения можно записать в виде обобщенной формулы:

$$x \cdot y = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (X[i] \cdot Y[j] \cdot 10^{i+j}).$$

Различные варианты алгоритма решеточного умножения вычисляют частные произведения $X[i] \cdot Y[j] \cdot 10^{i+j}$ в разном порядке и используют разнообразные стратегии для вычисления их суммы. Например, в *Liber Abaci*

⁶ Существует достаточно достоверное доказательство того, что древние шумеры выполняли точные вычисления с весьма большими числами, используя собственную шестидесятиричную позиционную систему записи чисел, но мне неизвестны какие-либо сохранившиеся записи о конкретных методах, применявшихся шумерами. В дополнение к стандартным таблицам умножения и обратных величин были найдены таблицы со списками квадратов целых чисел от 1 до 59, что привело некоторых историков математики к выводу о том, что вавилоняне выполняли умножение с использованием равенства, подобного $xy = ((x+y)^2 - x^2 - y^2)/2$. Но этот прием работает только при $x+y < 60$, а история умалчивает о том, как вавилоняне могли бы вычислить x^2 при $x \geq 60$.

⁷ Но на грани разжигания исторического конфликта, как между Грецией и Египтом, или между Лилипутией и Блефуску, или между пользователями Мас и пользователями РС, или между людьми, считающими ноль натуральным числом, и тем, кто с этим не согласен.

Фибоначчи описывает вариант, в котором рассматриваются частные произведения mn в порядке возрастания величины, как показано в современном псевдокоде ниже.

```

FibonacciMultiply( $X[0..m - 1]$ ,  $Y[0..n - 1]$ ):
  hold  $\leftarrow 0$ 
  for  $k \leftarrow 0$  to  $n + m - 1$ 
    for all  $i$  and  $j$  such that  $i + j = k$ 
      hold  $\leftarrow$  hold +  $X[i] \cdot Y[j]$ 
     $Z[k] \leftarrow$  hold mod 10
    hold  $\leftarrow$   $\lfloor$ hold/10 $\rfloor$ 
  return  $Z[0..m + n - 1]$ 

```

Алгоритм Фибоначчи часто выполняется с сохранением всех частных произведений в двумерной таблице (ее также называют «табло» (tableau), или «решетка» (grate), или «сетка» (lattice)) и последующим суммированием вдоль диагоналей с соответствующими переносами, как показано на рис. 0.1 справа. Ученики американских начальных школ обучаются умножению одного сомножителя (множимого) на каждую цифру второго сомножителя (множителя), записывая все произведения множителя на цифры множимого перед их суммированием, как показано на рис. 0.1 слева. Этот же метод был описан Евтокием, хотя он, как полагается, рассматривал цифры множителя слева направо, как показано на рис. 0.2. Оба эти варианта (и несколько других) описаны и проиллюстрированы в пошаговом виде в книге неизвестного автора «L'Arte dell'Abbaco» (1458 г.), также известной под названием «Treviso Arithmetic», первой печатной книги по математике на Западе.

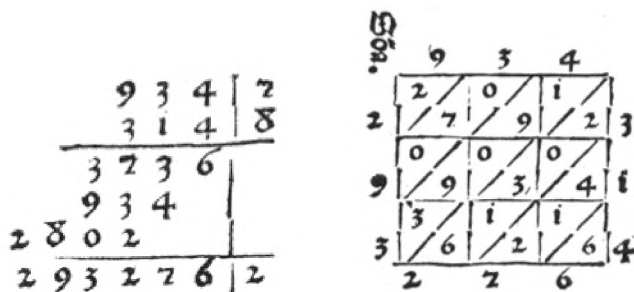


Рис. 0.1. Вычисление произведения $934 \times 314 = 293276$

с использованием умножения в столбик (с проверкой на ошибки методом, основанным на переходе к сравнениям, с «отбрасыванием» девяток) (слева) и решеточное умножение из книги «L'Arte dell'Abbaco» (1458 г.)

(Источник: Biblioteca nazionale Braidense (Milano);

<http://atena.beic.it/webclient/DeliveryManager?pid=2953344>)

The figure shows two versions of the multiplication of $1172 \frac{1}{8} \times 1172 \frac{1}{8}$. On the left is the original manuscript from the 6th century, written in Greek. It uses a grid-like structure to perform the multiplication, with the numbers written in Greek letters. On the right is a modern transcription of the same calculation, showing the numbers in Arabic numerals and the same grid-like structure. The final result is $1373877 \frac{1}{64}$.

Рис. 0.2. Вычисление Евтокия в VI в. $1172 \frac{1}{8} \times 1172 \frac{1}{8} = 1373877 \frac{1}{64}$ в комментариях к трактату Архимеда «Об измерении круга», воспроизведенное в оригинале (слева) и преобразованное в современную форму записи (справа) Йоханом Хайбергом (Johan Heiberg) (1891 г.) (Источник: Глобальный архив Internet Archive; <https://archive.org/details/archimedisopera05eutogoog/page/n377>)

Все эти варианты алгоритма умножения методом решетки и другие схожие варианты, описанные Сунь Цзы, аль-Хорезми, Фибоначчи, в книге «L'Arte dell'Abbaso» и во многих других источниках, вычисляют произведение любого m -значного и любого n -значного числа за время $O(mn)$ – время выполнения каждого варианта определяется в основном количеством операций умножения отдельных цифр.

Удваивание и усреднение

Алгоритм умножения методом решетки не является самым старым алгоритмом умножения, для которого имеется прямое письменное подтверждение. Существует еще более древний и, вероятно, более простой алгоритм, не применяющий позиционную запись чисел, – его иногда называют русским или эфиопским крестьянским умножением или просто крестьянским умножением (peasant multiplication). Вариант этого алгоритма был скопирован в папирус Ринда (Rhind papyrus; или математический папирус Ахмеса) египетским писцом по имени Ахмес (Ahmes) около 1650 г. до н. э. из документа, который, по утверждению Ахмеса, был старше приблизительно на 350 лет⁸. Этому алгоритму продолжали обучать в начальных школах Восточной Европы даже в конце XX века, а кроме того, алгоритм широко использовался в самых первых цифровых компьютерах, в аппаратуре которых не было непосредственно реализовано целочисленное умножение.

⁸ Версия этого алгоритма, действительно применявшаяся в Древнем Египте, не использовала усреднение или проверку на четность (сравнимость по модулю), но использовала сравнения. Чтобы избежать деления на два, алгоритм предварительно вычисляет две таблицы методом повторяющегося удваивания: одна таблица содержит все степени 2, не превышающие x , другая таблица содержит те же степени 2, умноженные на y . Затем выполняется поиск тех степеней 2, которые в сумме составляют число x , методом жадного вычитания, и соответствующие элементы в другой таблице суммируются для получения итогового произведения.

Алгоритм крестьянского умножения сводит сложную задачу умножения произвольных чисел к последовательности четырех более простых операций: (1) определения четности (четное или нечетное число), (2) сложения, (3) удваивания (числа) и (4) усреднения (взятие половины от числа с округлением в меньшую сторону или с недостатком).

<pre> PeasantMultiply(x, y): prod ← 0 while x > 0 if x is odd prod ← prod + y x ← ⌊x/2⌋ y ← y + y return prod </pre>	<i>x</i>	<i>y</i>	<i>prod</i>
			0
	123	+ 456 =	456
	61	+ 912 =	1368
	30	1824	
	15	+ 3648 =	5016
	7	+ 7296 =	12312
	3	+ 14592 =	26904
	1	+ 29184 =	56088

Рис. 0.3. Умножение методом удваивания и усреднения

Правильность этого алгоритма доказывается методом индукции из следующего рекурсивного тождественного равенства, справедливого для всех неотрицательных целых чисел x и y :

$$x \cdot y = \begin{cases} 0 & , \text{ если } x = 0; \\ \lfloor x/2 \rfloor \cdot (y + y) & , \text{ если } x - \text{ четное число}; \\ \lfloor x/2 \rfloor \cdot (y + y) + y & , \text{ если } x - \text{ нечетное число}. \end{cases}$$

Возможно, это рекуррентное тождество и есть алгоритм крестьянского умножения. Не позволяйте итеративному псевдокоду на рис. 0.3 обмануть вас – этот алгоритм рекурсивен по своей сущности.

Как было отмечено выше, алгоритм `PeasantMultiply` выполняет $O(\log x)$ операций проверки четности, сложения и усреднения, но можно улучшить его до $O(\log \min\{x, y\})$, если поменять местами аргументы при $x > y$. Предполагая, что числа представлены с использованием любой осмысленной позиционной нотации (двоичной, десятичной, вавилонской шестидесятеричной, египетской двенадцатеричной, римской, китайскими счетными палочками, позициями зернышек на абаке и т. д.), для каждой операции требуется как минимум $O(\log(xy)) = O(\log \max\{x, y\})$ операций умножения отдельных цифр, поэтому общее время выполнения алгоритма равно $O(\log \min\{x, y\} \cdot \log \max\{x, y\}) = O(\log x \cdot \log y)$.

Другими словами, этому алгоритму требуется время $O(mn)$ для умножения m -значного числа на n -значное число, с учетом постоянных сомножителей это то же самое время выполнения, что и для алгоритма умножения методом решетки. Этот алгоритм требует (при постоянном множителе) большего объема ручной работы, чем алгоритм решеточного умножения, но необходимые простейшие операции, вероятнее всего, гораздо проще выполняются человеком. В действительности эти два алгоритма равнозначны, если числа представлены в двоичном формате.

Циркуль и линейка

Греческие геометры-классики обозначали числа (или более точно – величины) отрезками прямых линий соответствующей длины, которыми они оперировали, используя два простых механических инструмента – циркуль и линейку, разновидности которых уже тогда широко применяли на практике землемеры, архитекторы и прочие ремесленники и мастеровые в течение многих столетий. Используя только эти два инструмента, ученые тех времен свели несколько сложных геометрических построений того времени к следующим простейшим операциям, начинающимся с одной или нескольких точно определенных исходных (начальных) точек:

- построению единственной прямой, проходящей через две различные точно определенные точки;
- построению единственной окружности с центром в одной точно определенной точке, проходящей через другую заданную точку;
- определению точки пересечения (если она существует) двух прямых;
- определению точек пересечения (если они существуют) прямой и окружности;
- определению точек пересечения (если они существуют) двух окружностей.

С определенной долей уверенности можно утверждать, что на практике греческие ученики, изучающие геометрию, выполняли построения на абаксе (абах (ἄβαξ)), поверхности, покрытой песком или плотным слоем пыли⁹. Несколькими веками ранее египетские землемеры выполняли многие из описанных выше построений, используя веревки для обозначения прямых и окружностей на земле¹⁰. Тем не менее Евклид и другие греческие геометры представляли построения с помощью циркуля и линейки как точные математические абстракции – точки как идеальные точки, прямые как идеальные прямые, окружности как идеальные окружности.

На рис. 0.4 показан алгоритм, описанный в «Началах» («Elements») Евклида около 2500 лет назад, для выполнения умножения или деления двух величин. Входные данные состоят из четырех различных точек A, B, C, D , а целью является построение точки Z , такой что $|AZ| = |AC||AD| / |AB|$. В частности, если определить $|AB|$ как единицу длины, то этот алгоритм вычисляет произведение $|AC|$ и $|AD|$.

Следует отметить, что Евклид сначала определяет новую элементарную операцию RightAngle с помощью написания «подпрограммы» (как называли

⁹ Изображаемые символы чисел от 1 до 9 были известны в Европе по меньшей мере за два века до написания Фибоначчи своего труда «Liber Abaci» как «гобар-числа» от арабского слова *ghubār*, означающего «пыль», что по существу представляло собой отсылку к индийскому практическому методу выполнения арифметических действий на поверхностях, покрытых песком. Греческое слово ἄβαξ является основой происхождения латинского слова *abacus*, которое изначально также означало поверхность, покрытую песком.

¹⁰ Помните, что означает слово «геометрия»? Впоследствии Демокрит упоминал этих египетских землемеров в слегка ироничной манере, называя их *arpedonaptai* (ἀρπεδονᾶπται), что означает «вязатели веревок».

бы ее современные программисты). Корректность алгоритма доказывается следующим наблюдением: треугольники ACE и AZF являются подобными. Вторая и третья строки основного алгоритма неоднозначны, потому что прямая α пересекает любую окружность с центром A в двух различных точках, но алгоритм действительно корректен вне зависимости от того, какие точки пересечения выбраны для E и F .

```

((Construct the line perpendicular to  $\ell$  passing through  $P$ .)
RightAngle( $\ell$ ,  $P$ ):
Choose a point  $A \in \ell$ 
 $A, B \leftarrow \text{Intersect}(\text{Circle}(P, A), \ell)$ 
 $C, D \leftarrow \text{Intersect}(\text{Circle}(A, B), \text{Circle}(B, A))$ 
return Line( $C, D$ )

((Construct a point  $Z$  such that  $|AZ| = |AC||AD|/|AB|$ .)
MultiplyOrDivide( $A, B, C, D$ ):
 $\alpha \leftarrow \text{RightAngle}(\text{Line}(A, C), A)$ 
 $E \leftarrow \text{Intersect}(\text{Circle}(A, B), \alpha)$ 
 $F \leftarrow \text{Intersect}(\text{Circle}(A, D), \alpha)$ 
 $\beta \leftarrow \text{RightAngle}(\text{Line}(E, C), F)$ 
 $\gamma \leftarrow \text{RightAngle}(\beta, F)$ 
return Intersect( $\gamma$ , Line( $A, C$ ))

```

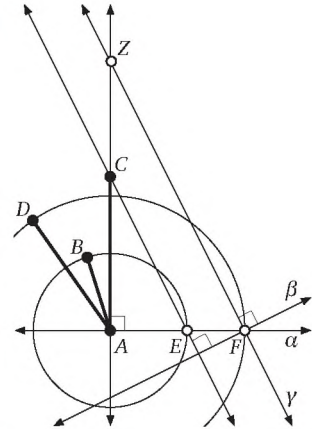


Рис. 0.4. Умножение с помощью циркуля и линейки

Алгоритм Евклида сводит задачу умножения двух величин (длин отрезков) к последовательности простейших операций, выполняемых с помощью циркуля и линейки. Эти операции сложно реализовать абсолютно точно в современных цифровых компьютерах, но алгоритм Евклида и не был предназначен для цифрового компьютера. Он был создан для платонова идеального геометра, вооруженного платоновым идеальным циркулем и платоновой идеальной линейкой, который способен точно выполнить каждую операцию за постоянное время согласно определению. В такой модели вычислений алгоритм `MultiplyOrDivide` выполняется за время $O(1)$.

0.3. Распределение мест в Конгрессе США

Ниже приведен еще один реальный пример практического алгоритма, весьма важного в сфере политики. Статья 1, раздел 2 Конституции США содержит следующее требование:

«Места представителей и прямые налоги распределяются между отдельными штатами, которые могут быть включены в настоящий Союз, согласно численности их населения... Число представителей устанавливается из расчета один представитель не более чем от каждых тридцати тысяч жителей при том условии, что каждый штат будет иметь по крайней мере одного представителя...»

Поскольку в Палате представителей количество мест ограничено, точное пропорциональное представление требует либо представительства

одновременно от нескольких штатов, либо дробного числа представителей, но такие варианты недопустимы (незаконны и невозможны). В итоге в течение нескольких следующих десятилетий предлагалось множество разнообразных алгоритмов распределения, в которых использовалось «беспристрастное» округление идеального дробного решения. Алгоритм, практически применяемый в настоящее время, называется методом Хантингтона–Хилла (Huntington-Hill method) или методом равных пропорций (method of equal proportions), впервые предложенным статистиком Бюро переписи населения США Джозефом Хиллом (Joseph Hill) в 1911 г. и улучшенным математиком Гарвардского университета Эдвардом Хантингтоном (Edward Huntington) в 1920 г., затем принятым в качестве федерального закона (2 U.S.C. § 2a) в 1941 г. и оставшимся в силе после проверки в Верховном суде в 1992 г.¹¹

Метод Хантингтона–Хилла распределяет представителей штатов поочередно по одному. Сначала на этапе предварительной обработки от каждого штата назначается один представитель. Затем на каждой итерации основного цикла следующий представитель назначается для штата с наивысшим приоритетом. Приоритет каждого штата определяется по формуле $P/\sqrt{r(r+1)}$, где P – численность населения штата, r – количество представителей, уже назначенных в этом штате.

Псевдокод этого алгоритма показан на рис 0.5. Входные данные состоят из массива $Pop[1..n]$, содержащего значения численности населения n штатов и целого числа R , равного общему количеству представителей, при этом предполагается, что $R \geq n$. (В настоящее время в США $n = 50$ и $R = 435$.) В выходном массиве $Rep[1..n]$ записано количество представителей от каждого штата.

Реализация алгоритма Хантингтона–Хилла использует очередь с приоритетами, поддерживающую операции `NewPriorityQueue`, `Insert` и `ExtractMax`. (Разумеется, в действующем законе ничего не сказано об очередях с приоритетами.) Вывод этого алгоритма, следовательно, его корректность никоим образом не зависит от того, как реализована эта очередь с приоритетами. Бюро переписи населения использует отсортированный массив, хранящийся в одном столбце электронной таблицы Excel, который вычисляется заново на каждой итерации. Вы изучали (по крайней мере, должны были изучать) более эффективную реализацию в начальном курсе по структурам данных.

¹¹ При аннулировании предыдущего решения Федерального окружного суда Верховный суд единогласно признал, что любой метод распределения, объективно принятый Конгрессом, является конституционным (Министерство торговли США против штата Монтана, юридический прецедент). Применяемый в настоящее время алгоритм распределения при избрании в Конгресс описан в чудовищных подробностях на веб-сайте Бюро переписи населения США <http://www.census.gov/topics/public-sector/congressional-apportionment.html>. Подробное описание истории решения этой задачи распределения можно найти здесь: <http://www.thirty-thousand.org/pages/Apportionment.htm>. Отчет Исследовательского управления Конгресса с описанием разнообразных методов распределения доступен здесь: <http://www.fas.org/sgp/crs/misc/R41382.pdf>.

```

ApportionCongress(Pop[1 .. n], R):
    PQ ← NewPriorityQueue
    «(Give every state its rst representative)»
    for s ← 1 to n
        Rep[s] ← 1
        Insert(PQ, s, Pop[s]/√2)
    «(Allocate the remaining n - R representatives)»
    for i ← 1 to n - R
        s ← ExtractMax(PQ)
        Rep[s] ← Rep[s] + 1
        priority ← Pop[s]/√Rep[s](Rep[s] + 1)
        Insert(PQ, s, priority)
    return Rep[1..n]

```

Рис. 0.5. Алгоритм распределения Хантингтона–Хилла

Подобные алгоритмы распределения используются при выборах в многопартийные парламенты по всему миру, где предполагается, что количество мест, выделяемых каждой партии, должно быть пропорциональным количеству голосов, полученных конкретной партией. Наиболее часто применяется метод Д’Ондта (D’Hondt)¹² и метод Сент-Лагю (Webster–Sainte-Laguë)¹³, в которых используются соответственно приоритеты $P/(r + 1)$ и $P/(2r + 1)$ вместо выражения с квадратным корнем в методе Хантингтона–Хилла. Метод Хантингтона–Хилла является существенно единственным решением для Палаты представителей США во многом благодаря конституционному требованию, согласно которому каждый штат непременно должен иметь как минимум одного представителя.

0.4. Отрицательный пример

В качестве типичного примера последовательности инструкций, которая в действительности не является алгоритмом, рассмотрим «алгоритм Мартина»¹⁴:

```

BeAMillionaireAndNeverPayTaxes(): «(БудьМиллионеромНоНикогдаНеПлатиНалоги())»
    Получить миллион долларов.
    Если налоговый инспектор подходит к двери вашего дома и говорит:
        "Вы никогда не платили налоги!", -
        Скажите: "Я забыл".

```

¹² Разработан Томасом Джефферсоном (Thomas Jefferson) в 1792 г., использовался для распределения мест в Конгрессе США с 1792 по 1832 г., повторно открыт (разработан) бельгийским математиком Виктором Д’Ондтом (Victor D’Hondt) в 1878 г. и усовершенствован швейцарским физиком Эдуардом Хагенбахом-Бишоффом (Eduard Hagenbach-Bischoff) в 1888 г.

¹³ Разработан Дэниэлом Уэбстером (Daniel Webster) в 1832 г., использовался для распределения мест в Конгрессе США с 1842 по 1911 г., повторно открыт (разработан) французским математиком Андре Сент-Лагю (André Sainte-Laguë) в 1910 г. и еще раз повторно разработан германским физиком Хансом Шеперсом (Hans Schepers) в 1980 г.

¹⁴ Стив Мартин (Steve Martin), «You Can Be A Millionaire», телевизионное шоу Saturday Night Live, 21 января 1978 г. Также звучит в аудио-альбоме «Comedy Is Not Pretty», Warner Bros. Records, 1979 г.

Все весьма просто, за исключением самого первого шага – он сногшибателен. Группа генеральных директоров-миллиардеров, венчурные инвесторы Кремниевой долины или дельцы, спекулирующие недвижимостью в Нью-Йорк-Сити, могут считать это алгоритмом, потому что для них самый первый шаг однозначен и банален¹⁵, но для остальных бедных недотеп вроде нас с вами процедура Мартина является весьма неопределенной, чтобы рассматривать ее как настоящий алгоритм. С другой стороны, это превосходный пример редукции (сведения) – в нем задача «перехода в миллионеры» и неуплаты налогов сводится к «более простой» задаче получения миллиона долларов. В этой книге операции редукции (сведения) будут встречаться многократно. Сотни бизнесменов и политиков уже наглядно доказали: если вы знаете, как решить более простую задачу, то редукция подскажет, как решить более сложную.

Алгоритм Мартина, как и некоторые из приведенных выше примеров, не является тем типом алгоритмов, над которыми размышляют ученые-исследователи в области информационных технологий, поскольку он сформулирован с использованием терминологии операций, которые трудно выполнимы для компьютеров. Эта книга (почти) полностью сконцентрирована на алгоритмах, которые могут быть разумно реализованы на стандартном цифровом компьютере. Каждый шаг в этих алгоритмах либо поддерживается непосредственно языками программирования общего назначения (через арифметические операции, присваивания, циклы или рекурсию), либо представляет собой некоторое действие, способ выполнения которого вам уже известен (например, сортировка, двоичный поиск, проход по дереву или исполнение песни «*n* бутылок пива на стене»).

0.5. Описание алгоритмов

Профессиональные навыки, требуемые для эффективного проектирования и анализа алгоритмов, тесно связаны с навыками, требуемыми для эффективного описания алгоритмов. По крайней мере, в моих курсах полное описание любого алгоритма содержит четыре компонента:

- **что:** точная спецификация задачи, которую решает алгоритм;
- **как:** точное описание самого алгоритма;
- **почему:** доказательство того, что именно этот алгоритм решает поставленную задачу;
- **скорость (как быстро):** анализ времени выполнения алгоритма.

Нет никакой необходимости в соблюдении указанного выше порядка разработки этих четырех компонентов (это даже нецелесообразно). Спецификации задач, описания алгоритмов, доказательства корректности и процедуры анализа времени выполнения обычно развиваются одновременно вместе с разработкой каждого компонента, предоставляющей информа-

¹⁵ Что-то вроде надежно защищенного квантового блокчейна с алгоритмом глубокого обучения чему-то.

цию для разработки прочих компонентов. Например, может потребоваться поправка в описании задачи для поддержки более быстрого алгоритма или изменение алгоритма для обработки нестандартного (сложного) варианта при доказательстве корректности. Как бы то ни было, отдельное представление этих компонентов обычно наиболее понятно для читателя.

При написании любого материала важно ориентировать описания на соответствующую аудиторию. Я рекомендую создавать описания для опытных, но скептически настроенных программистов, которые не так умны, как вы. Подумайте о том, каким вы будете через шесть месяцев. При разработке любого нового алгоритма вы естественным образом выстраиваете множество интуитивных предположений о поставленной задаче и о том, как ваш алгоритм решает ее, поэтому такое неформальное обоснование управляется вашей интуицией. Но любой человек или объект, который будет в дальнейшем читать ваш алгоритм или код, написанный на основе этого алгоритма, не сможет воспользоваться вашей интуицией или опытом. Ни компилятор. Ни сами вы шесть месяцев спустя. Все, что окажется в наличии, – созданное вами описание.

Даже если вы никогда не объясняли свои алгоритмы кому-либо другому, важно при разработке держать в уме предполагаемую аудиторию. Попытка обмена информацией в понятной форме заставит вас мыслить более точно и ясно. Особенно описание для группы новичков, которые будут воспринимать слова в точности так, как они написаны, заставит вас тщательно проработать подробности независимо от того, насколько «очевидными» или «интуитивно понятными» могут показаться в этот момент разработанные вами концепции. Аналогично описание для скептически настроенной аудитории заставит вас проработать надежные аргументы для доказательства корректности и эффективности, а не полагаться лишь на интуицию или умственные способности¹⁶.

Невозможно в достаточной степени выразить важность следующего положения: *ваша главная задача как создателя алгоритмов – объяснить другим людям, как и почему работают ваши алгоритмы*. Если вы не способны обмениваться своими идеями и концепциями с другим человеком, то, вероятнее всего, ваши идеи как будто не существуют вовсе. Создание корректного и эффективно выполняемого кода является важной, но второстепенной целью. Убедить себя, своих профессоров, своих (потенциальных) работодателей, коллег или студентов в том, что вы действительно умны, – это в лучшем случае лишь третья, далеко не самая важная цель.

Определение конкретной задачи

Прежде чем начать разработку нового алгоритма, необходимо установить, для решения какой задачи предназначается этот алгоритм. Точно так же, прежде чем мы сможем приступить к описанию алгоритма, потребуется описание задачи, для решения которой предназначается этот алгоритм.

¹⁶ В частности, я предполагаю, что читатель этой книги – скептически настроенный новичок.

Алгоритмические задачи часто формулируются с использованием обычного английского (русского или любого другого) языка с употреблением слов, обозначающих объекты реального мира. Но мы, как проектировщики алгоритмов, обязаны переформулировать эти задачи в терминах формальных, абстрактных, математических объектов, т. е. в форме чисел, массивов, списков, графов, деревьев и т. п., – таким образом, мы можем мыслить формально. Кроме того, мы обязательно должны определить, содержит ли формулировка задачи какие-либо скрытые неявные предположения (допущения), и зафиксировать эти предположения в явной форме. (Например, в песне « n бутылок пива на стене» подразумевается, что n всегда является неотрицательным целым числом¹⁷.)

Возможно, потребуется редактирование спецификации в процессе разработки алгоритма. Например, для алгоритма может возникнуть необходимость в подробном представлении входных или итоговых выходных данных (результатов), которые остались не сформулированными в исходном неформальном описании задачи. Или алгоритм, возможно, в действительности решает более обобщенную задачу, нежели изначально предложенная. (Это часто встречающееся свойство рекурсивных алгоритмов.)

Спецификация должна включать ровно столько подробностей, сколько нужно для того, чтобы любой другой человек мог использовать алгоритм как черный ящик, не задумываясь о том, как и/или почему он работает. По существу, необходимо обязательно объяснить тип и смысл каждого входного параметра, а также точно описать, как итоговый вывод зависит от входных параметров. С другой стороны, эта спецификация должна умышленно скрывать все подробности, которые не требуются для применения алгоритма как черного ящика. Пусть то, что не имеет значения, будет надежно скрыто от посторонних глаз.

Например, оба алгоритма – решеточного умножения и удваивания и усреднения – решают одну и ту же задачу: взять два неотрицательных целых числа x и y , представленных в виде массивов цифр, вычислить их произведение $x \cdot y$, также представленное в виде массива цифр. Для того, кто использует эти алгоритмы, выбор того или иного варианта абсолютно не важен. С другой стороны, греческий алгоритм построений с помощью циркуля и линейки решает другую задачу, потому что входные и выходные значения представлены отрезками прямых, а не массивами цифр.

Описание алгоритма

Компьютерные программы являются точными представлениями алгоритмов, но алгоритмы – это не программы. Алгоритмы – это абстрактные автоматически выполняемые процедуры, которые можно реализовать на любом языке программирования, поддерживающем элементарные низкоуровневые операции. Идиосинкразические особенности синтаксиса вашего любимого языка программирования не имеют никакого значения, они

¹⁷ Никогда не слышал, что кто-то поет « $\sqrt{2}$ бутылок пива на стене». Иногда приходилось слышать, как специалисты по теории множеств поют « $\aleph_0$ бутылок пива на стене», но по некоторой (вполне понятной) причине они никогда не могли допеть эту песню до конца.

лишь отвлекают ваше внимание (и внимание ваших читателей) от того, что в действительности происходит¹⁸. Качественное описание алгоритма ближе к тому, что необходимо писать в комментариях к реальной программе (как завершенному программному продукту), а не к ее исходному коду. Исходный код – плохое средство для повествования.

С другой стороны, описание в виде прозы на обычном английском (или любом другом языке) также не является удачным выбором. В алгоритмах содержится множество специфических структур – наиболее часто используются условные выражения, циклы, вызовы функций и рекурсия, – которые слишком легко «утопить» в неструктурированной прозе. Разговорный английский язык полон неоднозначностей и скрытых смыслов, но автор алгоритмов обязательно должен описывать их настолько однозначно и недвусмысленно, насколько это возможно. Обычная проза – плохой инструмент для соблюдения точности.

По моему мнению, самый точный способ представления алгоритма – использование сочетания псевдокода и структурированного английского языка. Псевдокод использует структуры формальных языков программирования и математические выражения для разделения алгоритма на элементарные шаги. Сами элементарные шаги могут быть записаны с применением математической нотации на правильном («чистом») английском языке или при объединении этих двух инструментов – выбирается самый понятный вариант. Правильно написанный псевдокод демонстрирует внутреннюю структуру алгоритма, но скрывает несущественные подробности реализации, делая алгоритм более простым для понимания, анализа, отладки и реализации.

Описание любого алгоритма должно включать все подробности, необходимые для полноценной спецификации алгоритма, для доказательства его корректности и для анализа времени его выполнения. В то же время следует исключить все подробности, которые не требуются для полноценной спецификации алгоритма, для доказательства его корректности и для анализа времени его выполнения. На уровне, более приближенном к практике, описание должно позволить опытному, но скептически настроенному программисту, который не читал эту книгу, быстро и правильно реализовать алгоритм на предпочитаемом языке программирования без необходимости понимания того, как работает этот алгоритм.

¹⁸ Разумеется, это вопрос «религиозных убеждений». Лингвисты-догматики постоянно спорят о гипотезе Сепира–Уорфа (*Sapir-Whorf hypothesis*; гипотеза лингвистической относительности), которая полагает, что (в большей или меньшей степени) люди мыслят в категориях, определяемых их родным языком. В соответствии со строгой формулировкой этого принципа некоторые концепции одного языка просто недоступны для понимания носителям других языков, и не только из-за различий в техническом развитии – как бы вы перевели выражение «jump the shark» или «Fortnite streamer» на арамейский язык? – но и из-за неотъемлемых структурных различий между языками и культурами. Более скептическая точка зрения представлена Стивеном Пинкером (Steven Pinker) в книге «Язык как инстинкт» («The Language Instinct»). По общему признанию, эта идея обладает определенной силой в применении к различным парадигмам программирования. (Что такое Y-комбинатор? Как работают шаблоны? Что такое абстрактная фабрика?) К счастью, слишком незначительной, чтобы оказать какое-либо воздействие на материал этой книги. В качестве весьма убедительного контрпримера см. монографию Криса Окасаки (Chris Okasaki) «Purely Functional Data Structures», а также более поздние его работы.

Я не хочу утомлять вас перечислением правил, которые я соблюдаю при написании псевдокода, но обязан предостеречь от особо вредной привычки. *Никогда* не описывайте повторяющиеся операции неформально, как в этом (анти)примере: «Сделать [это] один раз, затем сделать [то же] во второй раз и так далее» или «Повторять этот процесс до тех пор, пока [некоторое условие]». Каждый, кому приходилось искать ответ на один из обескураживающих вопросов типа «Что происходит дальше в этой последовательности?», уже знает, что описание нескольких первых шагов алгоритма почти ничего не сообщает о том, что происходит на более поздних шагах. Если в алгоритме есть цикл, то запишите его как цикл и дайте явное описание происходящего на любой произвольной итерации. Если алгоритм рекурсивный, то запишите его в рекурсивной форме и точно опишите границы возможных вариантов и все происходящее в каждом варианте.

0.6. Анализ алгоритмов

Недостаточно просто записать алгоритм и сказать: «Узрите!». Необходимо также убедить аудиторию (да и нас самих) в том, что этот алгоритм действительно делает то, для чего он предназначен, и делает это эффективно.

Корректность

При некоторых параметрах настройки приложения вполне приемлемым для программ является корректное поведение в течение большей части времени при всех «разумных» входных данных. Но только не в этой книге – мы требуем, чтобы алгоритмы были корректными всегда, при всех возможных входных данных. Более того, мы обязаны доказать, что предлагаемые нами алгоритмы являются корректными, при этом недостаточно доверять лишь собственным инстинктивным (интуитивным) предположениям или протестировать только несколько вариантов. Иногда корректность действительно очевидна, особенно для алгоритмов, которые вы изучали в начальных курсах. С другой стороны, «очевидность» слишком часто становится синонимом понятия «ошибочность». Для большинства алгоритмов, рассматриваемых в этом курсе, требуется определенный объем работы для доказательства их корректности. В частности, при доказательстве корректности обычно используется метод индукции. Нам нравится индукция. Индукция – наш друг¹⁹.

Разумеется, прежде чем мы сможем формально доказать, что предлагаемый алгоритм делает то, для чего он предназначен, сначала необходимо формально описать, что именно он должен делать.

Время выполнения

Наиболее часто используемым способом классификации различных алгоритмов для решения одной задачи является определение скорости их вы-

¹⁹ Если индукция не является вашим другом, то вам трудно будет читать эту книгу.

полнения. В идеальном случае нужен самый быстрый алгоритм для любой конкретной задачи. При некоторых параметрах настройки приложения вполне приемлемым для программ является эффективное выполнение в течение большей части времени при всех «разумных» входных данных. Но только не в этой книге – мы требуем, чтобы алгоритмы всегда выполнялись эффективно, даже в наихудшем случае.

Но как измерить время выполнения алгоритма? В качестве конкретного примера: сколько времени занимает исполнение песни `BottlesOfBeer(n)`? Очевидно, что время исполнения является функцией от входного значения n , но оно также зависит от того, насколько быстро вы можете петь. Некоторым певцам может потребоваться десять секунд на исполнение куплета, другим – двадцать. Определенные методики и технологии еще больше расширяют возможности. Передача песни по телеграфу с использованием кода Морзе может потратить целую минуту на куплет. Загрузка в формате *mp3* из веба, вероятно, займет около десятой доли секунды на куплет. Для передачи содержимого *mp3*-файла в оперативную память компьютера, скорее всего, потребуется всего лишь несколько микросекунд на куплет.

Здесь самое важное то, как изменяется время исполнения песни при возрастании n . Исполнение `BottlesOfBeer(2n)` требует приблизительно в два раза больше времени, чем исполнение `BottlesOfBeer(n)`, независимо от используемой технологии. Это отображается в асимптотическом выражении времени исполнения $\Theta(n)$.

Можно измерять время, подсчитывая, сколько раз алгоритм выполняет определенную инструкцию или достигает контрольной точки в своем «коде». Например, можно заметить, что слово «beer» повторяется трижды в каждом куплете `BottlesOfBeer`, поэтому количество спетых слов «beer» – подходящий показатель общего времени пения. На этот вопрос можно дать точный ответ: в песне `BottlesOfBeer(n)` слово «beer» упоминается в точности $3n + 3$ раза.

Между прочим, существует множество песен с квадратичным временем исполнения. Вот, вероятно, одна из самых известных большинству людей, говорящих на английском языке:

```

NDaysOfChristmas(gifts[2..n]):
  for i ← 1 to n
    Sing "On the ith day of Christmas, my true love gave to me"
    for j ← i down to 2
      Sing "j gifts[j],"
    if i > 1
      Sing "and"
    Sing "a partridge in a pear tree."

```

Входными данными для `NDaysOfChristmas` является список из $n - 1$ подарков (`gifts`), представленный здесь как массив. Достаточно легко показать,

что время исполнения равно $\Theta(n^2)$, по существу, исполнитель упоминает название подарка $\sum_{i=1}^n i = n(n+1)/2$ раз (считая «partridge in the pear tree» – дословно: «куропатка на грушевой ветке»; идиом. «яблоки на сосне», «в лесу родилась елочка»). Кроме того, легко заметить, что во время первых n дней Рождества «моя единственная любовь» дарит в точности $\sum_{i=1}^n \sum_{j=1}^n j = n(n+1)(n+2)/6 = \Theta(n^3)$ подарков.

К другим песням с квадратичным временем исполнения относятся: «Old MacDonald Had a Farm» («У старого МакДональда была ферма»), «There Was an Old Lady Who Swallowed a Fly» («Я знаю старушку, которая проглотила муху»), «Hole in the Bottom of the Sea» («На дне моря есть дыра»), «Green Grow the Rushes O» («Растет зеленый тростник, О»), «The Rattlin' Bog» («Шумное болото»), «The Court Of King Caractacus» («Двор короля Карактака»), «The Barley-Mow» («Сток ячменя»), «If I Were Not Upon the Stage» («Если бы я был не на сцене»), «Star Trekkin'» («Звездный путь»), «Ist das nicht ein Schnitzelbank?» («Это не резная скамейка?»)²⁰, «Il Pulcino Pio» («Цыпленок Пи»), «Minkurinn í hænsnakofanum» («Норка в курятнике»), «Echad Mi Yodea» («Кто знает Единого») и «То кокорáки» («Маленький петушок»). Если нужно больше примеров, спросите у хорошо знакомого вам ребенка дошкольного возраста.

Alouette(lapart[1..n]):

Chantez «Alouette, gentille alouette, alouette, je te plumerai.»
pour tout i de 1 à n

Chantez "Je te plumerai lapart[i]. Je te plumerai lapart[i]."
pour tout j de i à 1 **<<à rebours – вернуться назад>>**

Chantez "Et lapart[j]! Et lapart[j]!"

Chantez "Alouette! Alouette! Aaaaaa..."

Chantez "...alouette, gentille allouette, alouette, je te plumerai."

Для некоторых песен время исполнения выглядит еще более странным образом. Относительно недавним примером является песня Гая Стила (Guy Steele) «The TELNET Song», которая в действительности требует времени $\Theta(2^n)$ для исполнения первых n куплетов, Стил рекомендовал $n = 4$. Наконец, существует несколько песен, которые не заканчиваются никогда²¹.

За исключением «The TELNET Song» все перечисленные выше песни в основном вполне естественно записываются в виде небольшого набора вложенных циклов, поэтому время их исполнения (выполнения) можно вычислить с использованием вложенных операций суммирования. Время выполнения рекурсивного алгоритма проще выразить через рекуррентную формулу, как показано ниже:

²⁰ «Ja, das ist Otto von Schnitzelpusskrankengescheitmeyer!» – «Да, это Отто фон Шницельпускранкэнгешайтмейер!»

²¹ Они просто продолжают и продолжают бесконечно, мой друг.

$$x \cdot y = \begin{cases} 0 & , \text{ если } x = 0; \\ \lfloor x/2 \rfloor \cdot (y + y) & , \text{ если } x - \text{четное число}; \\ \lfloor x/2 \rfloor \cdot (y + y) + y & , \text{ если } x - \text{нечетное число}. \end{cases}$$

Пусть $T(x, y)$ обозначает количество операций определения четности, сложения и усреднения, требуемых для вычисления $x \cdot y$. Эта функция соответствует рекурсивному неравенству $T(x, y) \leq T(\lfloor x/2 \rfloor, 2y) + 2$ в основном варианте $T(0, y) = 0$. Методики, описанные в следующей главе, полагают, что верхняя граница $T(x, y) = O(\log x)$.

Иногда время выполнения алгоритма зависит от конкретной реализации некоторой внутренней структуры данных в подпрограмме. Например, алгоритм распределения представителей Хантингтона–Хилла `ApportionCongress` выполняется за время $O(N + RI + (R - n)E)$, где N обозначает время выполнения подпрограммы `NewPriorityQueue`, I – время выполнения операции `Insert`, а E – время выполнения операции `ExtractMax`. При разумном предположении $R \geq 2n$ (в среднем каждый штат получает как минимум двух представителей) можно упростить это предельное время выполнения до $O(N + R(I + E))$. Точное время выполнения зависит от реализации внутренней очереди с приоритетами. Бюро переписи населения реализует очередь с приоритетами как неотсортированный массив, для которого $N = I = \Theta(1)$ и $E = \Theta(n)$, поэтому предлагаемая Бюро переписи населения реализация алгоритма `ApportionCongress` выполняется за время $O(Rn)$. Но если реализовать очередь с приоритетами как двоичную кучу (пирамиду) или как динамический упорядоченный массив (`heap-ordered array`), то получим $N = \Theta(1)$ и $I = E = O(\log n)$, так что весь алгоритм в целом выполняется за время $O(R \log n)$.

Иногда требуется оценка не только времени выполнения, но и других вычислительных ресурсов, таких как пространство (объем) памяти, количество бросков монеты, количество промахов кеша или страницы памяти, количество сообщений, передаваемых между процессами, или количество «подарков от моей единственной возлюбленной». Эти ресурсы можно проанализировать с использованием тех же методик, которые применяются для анализа времени выполнения. Например, решеточное умножение двух n -значных чисел требует $O(n^2)$ пространства (памяти), если записываются все промежуточные (частичные) произведения перед их сложением, но только лишь $O(n)$ пространства (объема памяти), если произведения суммируются динамически (без промежуточного запоминания).

Упражнения

0. Описать и проанализировать эффективный алгоритм, который определяет с учетом разрешенного размещения стандартных фигур на стандартной шахматной доске, какой игрок выигрывает партию из заданной начальной позиции, если оба игрока играют безошибочно. [Подсказка: существует простейшее однострочное решение.]

- ♥1. а) Определить (или написать) песню, которая требует времени $\Theta(n^3)$ для исполнения первых n куплетов.
- б) Определить (или написать) песню, которая требует времени $\Theta(n \log n)$ для исполнения первых n куплетов.
- с) Определить (или написать) песню, которая требует некоторого другого невероятного времени для исполнения первых n куплетов.
2. Внимательные читатели, возможно, заметили и посетовали, что анализ таких песен, как « n бутылок пива на стене» или « n дней Рождества» слишком упрощен, потому что большие числа определяют более длительное время исполнения, чем малые числа. В более общем смысле: поскольку существует только определенное количество слов заданной длины, более крупные наборы слов не обязательно содержат более длинные слова²². Можно более точно оценить время исполнения песни, подсчитывая число спетых слогов, а не количество слов.
- а) Сколько времени потребуется, чтобы спеть целое число n ?
- б) Сколько времени потребуется, чтобы спеть « n бутылок пива на стене»?
- с) Сколько времени потребуется, чтобы спеть « n дней Рождества»?

Как обычно, необходимо представить ответы в форме $O(f(n))$ для некоторой функции f .

3. Застольная песня с многочисленными повторами «The Barley Mow» («Сток ячменя») веками исполнялась на Британских островах. Существует множество вариаций этой песни. На рис. 0.6 показан текст одной из версий, традиционных для графств Девон (Девоншир) и Корнуолл. Здесь `vessel[i]` – это название сосуда, содержащего 2^i унций пива²³.

²² Ja, das ist das Subatomarteilchenbeschleunigungs-natürmässigkeitsuntersuchungs-maschine! – Да, это машина для исследования природного (естественного) ускорения субатомных частиц!

²³ В действительности в этой песне используется некоторое подмножество следующих сосудов: nipperkin (рюмашка), quarter-gill (четверть джила), half-a-gill (полджила), gill (джил), quarter-pint (четверть пинты), half-a-pint (полпинты), pint (пинта), quart (кварта), pottle (4 пинты, или полгаллона), gallon (галлон), half-anker (полбочонка), anker (бочонок), firkin (бочонок побольше), half-barrel/kilderkin (полбочки/килдеркин), barrel (бочка), hogshead (большая бочка, хогсхед), pipe/butt (очень большая (вертикальная) бочка), tun (огромная бочка-резервуар), well (колодезь), river (река) и ocean (океан). С некоторыми исключениями (особенно в конце) каждый сосуд в этом списке вдвое больше по объему предыдущего. В ирландской и шотландской версиях этой песни немного другие тексты, и после галлона (gallon) они обычно переключаются на людей (barmaid (барменша), landlord (владелец (заведения), drayer и т. д.). Ранняя версия этой песни под названием «Give us once a drink» («Дай нам хоть раз выпить») впервые появилась в пьесе «Jack Drum's Entertainment, or the Comedie of Pasquill and Katherine» («Развлечения Джека Драма, или Комедия Пасквилья и Кэтерины»), написанной Джоном Марстоном (John Marston) около 1600 г. («Giue vs once a drinke for and the black bole. Sing gentle Butler bally moy!» (староангл.)). Существуют разногласия касательно того, написал ли Марстон эту «застольную голландскую песню» (high Dutch Song) специально для своей пьесы, является ли словосочетание «bally moy» мондегрином (дарвалдаем, т. е. неправильно услышанной фразой) от «barley mow» (собственно: «сток ячменя») и наоборот, или же это действительно та же самая песня. Эти споры лучше всего протекают за n бутылками пива.

BarleyMow(n):

```
"Here's a health to the barley-mow, my brave boys,"
"Here's a health to the barley-mow!"
"We'll drink it out of the jolly brown bowl,"
"Here's a health to the barley-mow!"
"Here's a health to the barley-mow, my brave boys,"
"Here's a health to the barley-mow!"
for i ← 1 to n
    "We'll drink it out of the vessel[i], boys,"
    "Here's a health to the barley-mow!"
    for j ← i downto 1
        "The vessel[ j],"
        "And the jolly brown bowl!"
        "Here's a health to the barley-mow!"
        "Here's a health to the barley-mow, my brave boys,"
        "Here's a health to the barley-mow!"
```

Рис. 0.6. Исполнение песни «The Barley Mow» («Сток ячменя»)

- а) Предположим, что каждое название сосуда $vessel[n]$ является одним словом, и можно петь со скоростью четыре слова в секунду. Сколько времени потребуется на исполнение песни $BarleyMow(n)$? (Дать ответ в форме строгого асимптотического граничного значения.)
 - б) Если необходимо исполнять эту песню с произвольными большими значениями n , то придется придумать собственные названия сосудов. Чтобы избежать повторений, эти названия должны становиться все длиннее при увеличении n . Предположим, что в названии $vessel[n]$ содержится $\Theta(\log n)$ слогов, и можно петь шесть слогов в секунду. Сколько времени в этом случае потребуется для исполнения песни $BarleyMow(n)$? (Дать ответ в форме строгого асимптотического граничного значения.)
 - в) Предположим, что при упоминании каждого названия сосуда действительно выпивается соответствующее количество пива: унция в каждой чаше «jolly brown bowl» и 2^i унций в каждом сосуде $vessel[i]$. Предполагая по условиям этой задачи, что вам не меньше 21 года, сколько унций пива вы выпьете при исполнении песни $BarleyMow(n)$? (Необходимо дать точный ответ, а не асимптотическое граничное значение.)
4. Напомню, что входными данными для алгоритма Хантингтона–Хилла $ApportionCongress$ является массив $Pop[1..n]$, где $Pop[i]$ – численность населения i -го штата, и целое число R – общее количество представителей, которые должны быть распределены. Выходные

данные: массив $Rep[1..n]$, где $Rep[i]$ – число представителей, назначенных от i -го штата по этому алгоритму.

Алгоритм Хантингтона–Хилла иногда описывается способом, позволяющим полностью избежать использования очередей с приоритетами. Алгоритм самого верхнего уровня «угадывает» положительное действительное число D , называемое делителем, затем выполняет приведенную ниже подпрограмму для вычисления распределения. Переменная q – идеальная квота для представителей, назначаемых от штата, при заданном делителе D . Действительное число назначенных представителей всегда равно либо $\lceil q \rceil$, либо $\lfloor q \rfloor$.

```

HNGuess(Pop[1..n], R, D):
  reps ← 0
  for i ← 1 to n
    q ← Pop[i]/D
    if q · q < [q] · [q]
      Rep[i] ← [q]
    else
      Rep[i] ← [q]
  reps ← reps + Rep[i]
  return reps

```

Существует три возможных варианта для конечного возвращаемого значения $reps$. Если $reps < R$, то требуемое количество представителей не назначено, что (по крайней мере, интуитивно) означает, что выбранный делитель D слишком мал. Если $reps > R$, то назначено слишком много представителей, что (по крайней мере, интуитивно) означает, что выбранный делитель D слишком велик. Наконец, если $reps = R$, то можно возвращать массив $Rep[1..n]$ как окончательное распределение мест в Конгрессе. На практике можно вычислить корректное распределение (с результатом $reps = R$), вызывая подпрограмму HNGuess с небольшим количеством целочисленных делителей, близких к стандартному делителю $D = P/R$.

В предлагаемых ниже задачах предполагается, что $P = \sum_{i=1}^n Pop[i]$ обозначает общую численность населения во всех n штатах, а также что $n \leq R \leq P$.

- Показать, что вызов HNGuess со стандартным делителем $D = P/R$ не всегда позволяет получить корректное распределение.
- Доказать, что если HNGuess возвращает одинаковое значение $reps$ для двух различных делителей D и D' , то вычисляется и одинаковое распределение $Rep[1..n]$ для обоих этих делителей.
- Доказать, что если HNGuess возвращает верное значение R , то вычислено то же распределение $Rep[1..n]$, что и при использовании ранее рассмотренного алгоритма ApportionCongress.

- d) Доказать, что «верный» делитель D , возможно, не существует. То есть описать входные данные $Pop[1..n]$ и R , где $n \leq R \leq P$, так, что для каждого действительного числа $D > 0$ число представителей, распределенных подпрограммой `HHGuess`, не равно R . [Подсказка: что произойдет, если заменить знак $<$ на знак $\leq$ в четвертой строке подпрограммы `HHGuess`?]

Глава 1

.....

Рекурсия

https://t.me/it_boooks/2

Контроль над большими силами осуществляется по тому же принципу, что и контроль над несколькими людьми: это просто вопрос разделения их численности.

— Сунь Цзю, «Искусство войны» (около 400 г. н. э.)
(перевел на англ. яз. Лайонел Джайлз (Lionel Giles), 1910 г.)

Наша жизнь испорчена подробностями... Упрощайте, упрощайте.

— Генри Дэвид Торо (Henry David Thoreau), «Уолден, или Жизнь в лесу» (1854 г.)

И не спрашивайте меня, что такое Voom. Я никогда об этом не узнаю.

Но парень! Должен сказать тебе, что он ДЕЙСТВИТЕЛЬНО чистит снег!

— Доктор Сьюз (Dr. Seuss), Теодор Сьюз Гайсел (Theodor Seuss Geisel),
«Кот в шляпе возвращается» (1958 г.)

В первую очередь делайте тяжелую работу. Легкая работа позаботится о себе сама.

— Цитата приписывается Дейлу Карнеги (Dale Carnegie)

1.1. Сведение

Сведение, или редукция (reduction), – одна из методик, наиболее часто используемых в проектировании алгоритмов. Сведение одной задачи X к другой задаче Y означает создание алгоритма для X , который использует алгоритм для Y как черный ящик или подпрограмму. Особенно важно то, что корректность итогового алгоритма для задачи X не может зависеть каким бы то ни было образом от того, как работает алгоритм для задачи Y . Единственное, что можно предполагать в этом случае, – черный ящик корректно решает задачу Y . Внутренние рабочие операции этого черного ящика просто нас не касаются, это задача решается кем-то другим. Часто наилучшим подходом является восприятие черного ящика как исключительно магического устройства.

Например, алгоритм «крестьянского умножения», описанный в предыдущей главе, сводит задачу умножения двух произвольных положительных целых чисел к трем более простым задачам: сложению, усреднению (взятию половины) и проверке четности. Алгоритм основан на абстрактном типе данных «положительное целое число», который поддерживает

три перечисленные операции, но корректность этого алгоритма умножения не зависит от точности представления данных (засечки на дощечке, глиняные таблички, вавилонская шестидесятеричная система, узелковое письмо кипу, счетные палочки, римские цифры, счет на пальцах, камешки на счетной таблице, числа гобар (хинди), двоичные числа, нега-позиционная двоичная система счисления, код Грея, равновесная троичная система счисления, фи-нарная система счисления (основанная на золотом сечении), четверично-мнимая система счисления...) или от точности реализации элементарных операций. Разумеется, время выполнения алгоритма умножения зависит от времени выполнения операций сложения, усреднения и определения четности, но это характеристика, не относящаяся к корректности. Наиболее важно то, что можно создать более эффективный алгоритм умножения с помощью простого перехода на более удобное представление данных (например, от засечек на дощечке к позиционной системе записи чисел).

Аналогичным образом алгоритм Хантингтона–Хилла сводит задачу распределения мест в Конгрессе к задаче сопровождения очереди с приоритетами, которая поддерживает операции Insert (вставка) и ExtractMax (извлечение максимального значения). Абстрактный тип данных «очередь с приоритетами» представляет собой черный ящик, а корректность алгоритма распределения не зависит от любой конкретной структуры данных очереди с приоритетами. Разумеется, время выполнения алгоритма распределения зависит от времени выполнения алгоритмов Insert и ExtractMax, но эти характеристики полностью отделены от характеристики корректности алгоритма. Достоинство сведения состоит в том, что можно создать более эффективный алгоритм распределения, просто применив новую структуру данных очереди с приоритетами. Более того, проектировщику такой структуры данных не обязательно знать или беспокоиться о том, что она будет использоваться для распределения мест в Конгрессе.

При проектировании алгоритмов автор может не знать точно, как именно реализованы используемые им базовые структурные блоки или как создаваемые алгоритмы могут применяться в качестве базовых структурных блоков для решения более крупномасштабных задач. Такое незнание создает дискомфорт для многих начинающих алгоритмистов, но это неизбежно и чрезвычайно полезно. Даже если вы точно знаете, как работают используемые компоненты, часто весьма полезно сделать вид, что вам это неизвестно.

1.2. Упрощение и делегирование

Рекурсия (recursion) – это особенно мощный тип сведения, который кратко можно описать следующим образом:

- если заданный частный случай задачи можно решить напрямую, то решить его напрямую;

- иначе свести его к одному или нескольким более простым частным случаям той же задачи.

Если ссылка на себя (рекурсивная ссылка) вас смущает, то, возможно, удобнее представить, что кто-то другой будет решать эти более простые задачи, как можно было бы представить это для других типов сведения. Я предпочитаю называть этого «кого-то другого» Феей Рекурсией (Recursion Fairy). Ваша единственная обязанность – упростить исходную задачу или решить ее напрямую, если упрощение не нужно или невозможно, а Фея Рекурсия решит за вас все более простые подзадачи, применяя методы, «о которых вам не нужно знать», так что «не лезьте не в свое дело» (butt out)¹. Читатели с хорошей математической подготовкой, вероятно, поняли, что под именем Феи Рекурсии скрывается более формальная сущность: индуктивное предположение (Induction hypothesis).

Существует одно умеренное техническое условие, которое необходимо соблюдать для обеспечения корректной работы любого рекурсивного метода: последовательность сведений для получения все более простых частных случаев задачи не должна быть бесконечной. В конце концов рекурсивные сведения непременно должны приводить к элементарному частному случаю задачи, который можно решить некоторым другим методом, иначе рекурсивный алгоритм будет бесконечно зациклен. Наиболее часто применяемым способом выполнения вышеописанного условия является сведение к одному или нескольким более простым (более «мелким») частным случаям той же задачи. Например, если исходными входными данными является некий элемент с n атрибутами, то входными данными для каждого рекурсивного вызова должен быть элемент, в котором обязательно содержится строго меньше, чем n атрибутов. Разумеется, невозможно, чтобы элемент вообще не содержал атрибутов (не может быть отрицательным число атрибутов – это глупо и бессмысленно!), так что в этом случае мы непременно должны «перезаточить» элемент, используя некоторый другой метод.

Нам уже знаком один частный случай этого шаблона в алгоритме «крестьянского умножения», основанного непосредственно на следующем рекурсивном тождестве.

$$x \cdot y = \begin{cases} 0 & , \text{ если } x = 0; \\ \lfloor x/2 \rfloor \cdot (y + y) & , \text{ если } x - \text{ четное число}; \\ \lfloor x/2 \rfloor \cdot (y + y) + y & , \text{ если } x - \text{ нечетное число}. \end{cases}$$

Это рекурсивное (рекуррентное) тождество можно записать в алгоритмической форме, как показано ниже:

¹ Когда я был студентом начальных курсов, то приписывал рекурсию «эльфам», а не Фее Рекурсии, в соответствии со сказкой братьев Гримм о старом сапожнике, который не закончил работу и лег спать, а когда проснулся, то обнаружил, что эльфы («Wichtelmänner») ночью сделали все за него. Кто-то более подкованный в энтеогенном плане, чем я, может распознать в этих «рекурсивных эльфах» (Rekursionswichtelmänner) «самотрансформирующихся механических (фрактальных) эльфов» Терренса Кемпа Маккенны (Terence Kemp McKenna).



```

PeasantMultiply(x,y):
  if x = 0
    return 0
  else
    x' ← ⌊x/2⌋
    y' ← y + y
    prod ← PeasantMultiply(x', y')    <<Рекурсия!>>
    if x is odd
      prod ← prod + y
    return prod

```

Ленивый египетский писец мог выполнить этот алгоритм, вычисляя x' и y' и попросив более юного коллегу перемножить x' и y' , а затем, вероятно, сложив y с результатом младшего писца. Задача юного писца проще, потому что $x' < x$, и постоянное уменьшение положительного целого числа в итоге приводит к 0. Старшего писца совершенно не интересует, как младший коллега в действительности вычисляет произведение $x' \cdot y'$ (и нам тоже нет до этого никакого дела).

1.3. Ханойские башни

Головоломка «Ханойская башня» (Tower of Hanoi) впервые была опубликована – как действительно существующая в реальном мире задача – французским преподавателем и специалистом по занимательной математике Эдуардом Люка (Édouard Lucas) в 1883 г.² под псевдонимом N. Claus (de Siam) (Н. Клаус (из Сиам)) (это анаграмма фразы «Lucas d'Amiens» (Люка из Амьена)). В следующем году Анри де Парвиль (Henry de Parville) описал эту головоломку в следующем удивительном рассказе³:

«В большом храме города Бенареса⁴ под куполом, накрывающим “центр всего мира” покоится медная плита, в которую вставлены три алмазные иглы длиной в локоть и толщиной в осиную талию. На одну из этих игл Бог при сотворении мира нанизал шестьдесят четыре диска из чистого золота – самый большой диск лежит в самом низу на медной плите, и каждый диск, лежащий выше, меньше предыдущего. Это “Башня Брамь”. День и ночь священнослужители неустанно переносят диски с одной алмазной иглы на другую, руководствуясь навеки установленными и непреложными законами Брамь, по которым священнослужитель не должен двигать зараз более одного диска и всегда должен так переносить этот диск на иглу, чтобы под ним

² Позже Люка признался, что придумал эту головоломку в 1876 г.

³ Для этой книги был использован перевод на английский язык, взятый из книги Уолтера Уильяма Роуз Болла (W. W. Rouse Ball) «Mathematical Recreations and Essays» («Математические эссе и развлечения»; есть перевод на русский язык) (1892 г.)

⁴ Под «большим храмом города Бенареса» почти определенно подразумевается Храм Каши Вишванатх (Каши Вишванатх мандир) в священном городе Варанаси (штат Уттар-Прадеш, Индия), расположенный приблизительно на расстоянии 2400 км к северо-западу от Ханоя (Вьетнам), где предположительно проживает воображаемый Н. Клаус. По воле случая французская армия захватила Ханой в 1883 г., в том же году Люка опубликовал свою головоломку, в итоге Ханой был провозглашен столицей французского Индокитая.

не оказался диск, меньший его по размеру. А когда все шестьдесят четыре диска будут перенесены с той иглы, на которую Бог поместил их при сотворении мира, на одну из двух других игл, то башня, храм и сами священнослужители-брамины обратятся в прах, и грянет гром, и мир исчезнет».

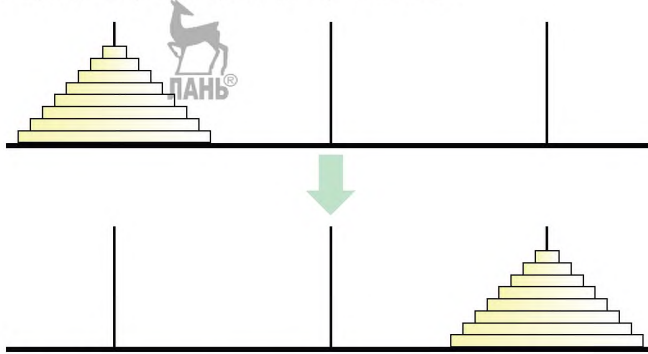


Рис. 1.1. Головоломка «Ханойская башня» (с 8 дисками)

Разумеется, как опытные специалисты в области информационных технологий, при чтении этого рассказа мы сразу же приходим к мысли о замене «жестко закодированной» константы 64 на переменную n . А поскольку большинство материальных экземпляров этой головоломки сделано из дерева, а не из алмазов и золота, я буду называть три возможных места для дисков стержнями, а не иглами. Как можно переместить башню из n дисков с одного стержня на другой, используя третий дополнительный стержень как временное место расположения дисков, при условии, что нельзя положить диск большего размера на диск меньшего размера?

Как подсказывает Н. Клаус (из Сиам) в своем наставлении, включенном в эту головоломку, секрет ее решения заключается в рекурсивном способе мышления. Вместо попыток решения всей головоломки сразу попробуйте сосредоточиться на перемещении только одного самого большого диска. Мы не можем переместить его первым, потому что этому мешают другие диски. Поэтому сначала необходимо переместить $n - 1$ этих дисков меньшего размера на вспомогательный стержень. После этого появляется возможность положить самый большой диск прямо на целевой стержень. Далее для завершения решения головоломки необходимо переместить $n - 1$ дисков меньшего размера со вспомогательного на целевой стержень.

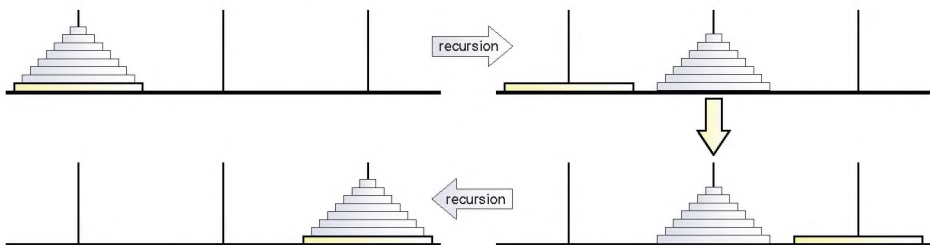


Рис. 1.2. Алгоритм решения головоломки «Ханойская башня» – сосредоточимся на самом нижнем диске, на остальные не обращаем внимания

Теперь, все, что нам нужно, – понять, как...

НЕТ! ПОСТОЙТЕ!

Вот и все. Решение найдено. Мы успешно свели задачу «Ханойская башня» с n дисками к двум частным случаям задачи «Ханойская башня» с $(n - 1)$ дисками, которую можно с ликованием передать Фее Рекурсии или в соответствии с легендой Люка поручить младшим монахам в храме. Наша часть работы завершена. Если бы мы не доверяли младшим монахам, то зачем было привлекать их к этой работе, так что позвольте им спокойно выполнять их обязанности.

Описанное здесь сведение использует одно не слишком заметное, но чрезвычайно важное предположение: самый большой диск действительно существует. Показанный алгоритм работает для любого положительного числа дисков, но он отказывается работать при $n = 0$. Этот случай обязательно необходимо обработать с применением другого метода. К счастью, монахи в Бенаресе, будучи истинными буддистами, весьма компетентны в плане перемещения нуля дисков со стержня на стержень практически без затрат времени – они просто ничего не делают в этом случае.



Рис. 1.3. Вырожденный частный случай алгоритма «Ханойская башня». «Ложки нет» (© к/ф «Матрица»)

Может возникнуть искушение задуматься о том, как перемещать все эти диски по стержням, или более обобщенно – что происходит, когда рекурсия разворачивается, – но в действительности не следует думать об этом. Для большинства рекурсивных алгоритмов (мысленное) развертывание процесса рекурсии не является необходимостью и не приносит никакой пользы. Единственная наша обязанность – свести частный случай поставленной задачи к одному или нескольким еще более простым частным случаям этой задачи или решить задачу напрямую, если такое сведение невозможно. Предложенный здесь алгоритм решения «Ханойской башни» тривиально корректен при $n = 0$. При любом $n \geq 1$ Фея Рекурсия корректно перемещает $n - 1$ верхних дисков (более формально: по индуктивному предположению принимается, что наш рекурсивный алгоритм корректно перемещает $n - 1$ верхних дисков), таким образом, этот алгоритм корректен.

Рекурсивный алгоритм `hanoi`, записанный на псевдокоде, показан на рис. 1.4. Алгоритм перемещает стопку из n дисков с исходного стержня-источника (`src`) на целевой стержень (`dst`), используя третий временный стержень (`tmp`) как место временного размещения дисков. Следует отметить, что алгоритм корректно ничего не делает при $n = 0$.



```
Hanoi(n, src, dst, tmp):
if n > 0
    Hanoi(n - 1, src, tmp, dst)    <<Рекурсия!>>
    move disk n from src to dst
    Hanoi(n - 1, tmp, dst, src)    <<Рекурсия!>>
```

Рис. 1.4. Рекурсивный алгоритм для решения задачи «Ханойская башня»

Пусть $T(n)$ обозначает количество ходов, требуемых для передвижения n дисков, – время работы нашего алгоритма. В вырожденном частном случае подразумевается, что $T(0) = 0$, а в более общем рекурсивном алгоритме предполагается, что $T(n) = 2T(n - 1) + 1$ при любом $n \geq 1$. Выписав первые несколько значений $T(n)$, мы можем легко догадаться, что $T(n) = 2^n - 1$; прямое индуктивное доказательство подтверждает правильность этого предположения. В частности, перемещение башни из 64 дисков требует $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ отдельных ходов. Таким образом, даже при невероятной скорости выполнения одного хода в секунду монахам в храме Бенареса придется работать приблизительно 585 млрд. лет («plus de cinq milliards de siècles» – «более пяти миллиардов веков»), прежде чем башня, храм и сами монахи-брамины обратятся в прах и с раскатом грома мир исчезнет.

1.4. Сортировка слиянием

Сортировка слиянием (Mergesort) – один из самых первых алгоритмов, предназначенных для компьютеров общего назначения с возможностью хранения программ. Этот алгоритм был разработан Джоном фон Нейманом (John von Neumann) в 1945 г. и подробно описан в опубликованной совместно с Германом Голдстейном (Herman Goldstine) работе в 1947 г. как одна из первых нечисловых программ для компьютера EDVAC⁵.

1. Разделить входной массив данных на два подмассива приблизительно равного размера.
2. Выполнить рекурсивно сортировку слиянием для каждого подмассива.
3. Выполнить слияние новых отсортированных подмассивов в один отсортированный массив.

⁵ Голдстейн и фон Нейман в действительности описали нерекурсивный вариант, который теперь обычно называют восходящей сортировкой слиянием (bottom-up mergesort). В те времена большие наборы данных сортировались машинами специального назначения, которые в основном производились компанией IBM. Для этих машин применялись перфокарты и использовались варианты двоичной поразрядной (цифровой) сортировки. Фон Нейман (успешно) доказал, что, поскольку компьютер EDVAC был способен выполнять сортировку быстрее, чем специализированные сортировочные машины IBM «без вмешательства человека и без потребности в дополнительном оборудовании», EDVAC представлял собой «универсальную (многоцелевую)» машину, поэтому необходимость в специализированных сортировочных машинах отпала.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L	
Divide:	S	O	R	T	I	N		G	E	X	A	M	P	L
Recurse Left:	I	N	O	R	S	T		G	E	X	A	M	P	L
Recurse Right:	I	N	O	R	S	T		A	E	G	L	M	P	X
Merge:	A	E	G	I	L	M	N	O	P	R	S	T	X	

Рис. 1.5. Пример сортировки слиянием

Первый шаг абсолютно очевиден – простое разделение массива на два по размеру, – а второй шаг можно передать (делегировать – *delegate*) Фее Рекурсии. Вся настоящая работа выполняется на завершающем шаге слияния. Полное описание этого алгоритма приведено на рис. 1.6. Для того чтобы рекурсивная структура была четко видна, я выделил шаг слияния в отдельную независимую подпрограмму. Алгоритм слияния также является рекурсивным – определение первого элемента выходного массива, затем рекурсивное слияние остальной части входных массивов.

```
MergeSort(A[1..n]):
```

```
if n > 1
```

```
  m ← [n/2]
```

```
  MergeSort(A[1..m])      <<Рекурсия!>>
```

```
  MergeSort(A[m + 1..n])  <<Рекурсия!>>
```

```
  Merge(A[1..n], m)
```

```
Merge(A[1..n], m):
```

```
  i ← 1; j ← m + 1
```

```
  for k ← 1 to n
```

```
    if j > n
```

```
      B[k] ← A[i]; i ← i + 1
```

```
    else if i > m
```

```
      B[k] ← A[j]; j ← j + 1
```

```
    else if A[i] < A[j]
```

```
      B[k] ← A[i]; i ← i + 1
```

```
    else
```

```
      B[k] ← A[j]; j ← j + 1
```

```
  for k ← 1 to n
```

```
    A[k] ← B[k]
```

Рис. 1.6. Алгоритм сортировки слиянием

Корректность

Для доказательства корректности этого алгоритма дважды воспользуемся помощью нашего старого доброго друга – индукции: сначала для подпрограммы Merge, затем для алгоритма более высокого уровня MergeSort.

Лемма 1.1. Алгоритм Merge корректно выполняет слияние подмассивов $A[1..m]$ и $A[m + 1..n]$, предполагая, что эти подмассивы отсортированы при вводе.

Доказательство. Пусть $A[1..n]$ – любой произвольный массив, а m – любое произвольное целое число, такое что подмассивы $A[1..m]$ и $A[m + 1..n]$ отсортированы. Докажем, что для всех k от 0 до $n - k - 1$ последних ите-

раций основного цикла корректно выполняют слияние $A[i..m]$ и $A[j..n]$ в массив $B[k..n]$. Это доказательство проводится методом (математической) индукции по $n - k + 1$, т. е. по числу оставшихся элементов, для которых требуется слияние.

Если $k > n$, то алгоритм корректно выполняет слияние двух пустых подмассивов, при этом абсолютно ничего не делая. (Это базовый (исходный) случай для индуктивного доказательства.) Иначе существуют четыре варианта, которые необходимо рассмотреть для k -й итерации основного цикла:

- если $j > n$, то подмассив $A[j..n]$ пуст, следовательно, $\min(A[i..m] \cup A[j..n]) = A[i]$;
- если $i > m$, то подмассив $A[i..m]$ пуст, следовательно, $\min(A[i..m] \cup A[j..n]) = A[j]$;
- иначе если $A[i] < A[j]$, то $\min(A[i..m] \cup A[j..n]) = A[i]$;
- иначе мы обязательно должны получить $A[i] \geq A[j]$ и $\min(A[i..m] \cup A[j..n]) = A[i]$.

Во всех четырех случаях $B[k]$ правильно присваивается наименьший элемент $A[i..m] \cup A[j..n]$. В двух случаях во время присваивания $B[k] \leftarrow A[i]$ Фея Рекурсия корректно выполняет слияние... Извините, я имел в виду, что индуктивное предположение подразумевает, что $n - k$ последних итераций основного цикла правильно выполняет слияние $A[i + 1..m]$ и $A[j..n]$ в массив $B[k + 1..n]$. Аналогично в двух других случаях Фея Рекурсия также корректно выполняет слияние остальных элементов подмассивов. □

Теорема 1.2. Алгоритм MergeSort корректно сортирует любой входной массив $A[1..n]$.

Доказательство. Докажем эту теорему методом (математической) индукции по n . Если $n \leq 1$, то алгоритм ничего не делает, и это правильно. Иначе Фея Рекурсия корректно сортирует... Еще раз извините, я имел в виду, что индуктивное предположение подразумевает, что данный алгоритм корректно сортирует два меньших подмассива $A[1..m]$ и $A[m + 1..n]$, после чего эти два подмассива корректно объединяются подпрограммой Merge (слияние) в один отсортированный массив (по лемме 1.1). □

Анализ

Поскольку алгоритм MergeSort является рекурсивным, его время выполнения, естественно, записывается в виде рекуррентного выражения. Для выполнения подпрограммы Merge, очевидно, требуется время $O(n)$, потому что это простой цикл for с постоянным объемом работы на каждой итерации. Сразу же получаем следующее рекуррентное выражение для алгоритма MergeSort:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) + O(n).$$

Как и в большинстве рекуррентных тождеств по принципу «разделяй и властвуй», можно безопасно отбросить минимальное и максимальное предельные значения (используя методику преобразований области (domain transformations), описанную ниже в этой главе), что дает упрощенное рекуррентное тождество $T(n) = 2T(n/2) + O(n)$. Случай «все уровни равнозначны» метода рекурсивного дерева (который также будет описан ниже в этой главе) позволяет немедленно получить аналитическое (в замкнутой форме) решение $T(n) = O(n \log n)$. Даже если вы (пока еще) не знакомы с рекурсивными деревьями, можете проверить правильность решения $T(n) = O(n \log n)$ методом индукции.

1.5. Быстрая сортировка

Быстрая сортировка (quicksort) – еще один рекурсивный алгоритм сортировки, разработанный Тони Хоаром (Tony Hoare) в 1959 г. и впервые опубликованный в 1961 г. В этом алгоритме основная часть работы заключается в разделении массива на подмассивы меньшего размера перед рекурсией, поэтому слияние отсортированных подмассивов является тривиальной задачей.

1. Выбрать опорный элемент (pivot) в исходном массиве.
2. Разделить исходный массив на три подмассива, содержащих элементы, меньшие опорного, сам опорный элемент и элементы, большие опорного.
3. Рекурсивно выполнить алгоритм быстрой сортировки для первого и последнего подмассива.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	N	G	E	X	A	M	P	L
Partition:	A	G	O	E	I	N	L	M	P	T	X	S	R
Recurse Left:	A	E	G	I	L	M	N	O	P	T	X	S	R
Recurse Right:	A	E	G	I	L	M	N	O	P	R	S	T	X

Рис. 1.7. Пример выполнения алгоритма быстрой сортировки

Более подробный псевдокод показан на рис. 1.8. В подпрограмме Partition входной параметр p является индексом опорного элемента в неотсортированном массиве. Эта подпрограмма разделяет массив на части и возвращает новый индекс опорного элемента. Существует много разнообразных эффективных алгоритмов разделения, авторство представленного здесь алгоритма приписывается Нико Ломуто (Nico Lomuto)⁶. Переменная ℓ является счетчиком элементов массива, меньших (ℓ ess), чем опорный элемент.

⁶ Хоар предлагал более сложный «двунаправленный» алгоритм разделения, обладающий некоторыми практическими преимуществами, по сравнению с алгоритмом Ломуто. С другой стороны, алгоритм разделения Хоара – один из тех, в которых ошибки на единицу (ошибки неучтенной единицы – off-by-one errors) приводят к полному краху.


```
QuickSort(A[1..n]):
```

```
if n > 1
```

```
    Choose a pivot element A[p]
```

```
    r ← Partition(A, p)
```

```
    QuickSort(A[1..r - 1])  <<Рекурсия!>>
```

```
    QuickSort(A[r + 1..n])  <<Рекурсия!>>
```

```
Partition(A[1..n], p):
```

```
    swap A[p] ↔ A[n]
```

```
    ℓ ← 0      <<#items < pivot>>
```

```
    for i ← 1 to n - 1
```

```
        if A[i] < A[n]
```

```
            ℓ ← ℓ + 1
```

```
            swap A[ℓ] ↔ A[i]
```

```
    swap A[n] ↔ A[ℓ + 1]
```

```
    return ℓ + 1
```

Рис. 1.8. Алгоритм быстрой сортировки QuickSort

Корректность

Как и для алгоритма сортировки слиянием, доказательство корректности алгоритма QuickSort требует двух отдельных доказательств методом индукции: одно для доказательства того, что подпрограмма Partition правильно разделяет массив, второе для доказательства того, что алгоритм QuickSort корректно выполняет сортировку при том, что подпрограмма Partition корректна. Для доказательства корректности подпрограммы Partition необходимо доказать инвариантность следующего цикла: в конце каждой итерации основного цикла каждый элемент в подмассиве $A[1..ℓ]$ меньше $A[n]$ и ни один элемент в подмассиве $A[ℓ + 1..n]$ не меньше $A[n]$. Остальные очевидные, но утомительные подробности этого доказательства предлагаются читателю в качестве упражнения для самостоятельного выполнения.

Анализ

Анализ алгоритма быстрой сортировки также похож на анализ алгоритма сортировки слиянием. Ясно, что подпрограмма Partition выполняется за время $O(n)$, потому что это простой цикл for с константным временем работы на каждой итерации. Для алгоритма QuickSort получаем рекуррентное выражение, зависящее от r – позиции (rank) выбранного опорного элемента:

$$T(n) = T(r - 1) + T(n - r) + O(n).$$

Если бы можно было каким-либо способом всегда выбирать опорный элемент так, чтобы он являлся элементом со средним значением (медианой) массива A , то получили бы позицию $r = \lfloor n/2 \rfloor$, и две подзадачи были бы близки по размеру друг к другу, насколько это возможно. При этом рекуррентное выражение приобрело бы следующий вид:

$$T(n) = T(\lfloor n/2 \rfloor - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n),$$

и в результате $T(n) = O(n \log n)$ при использовании либо метода рекурсивного дерева, либо еще более простого метода «О да, мы уже решили это рекуррентное выражение для алгоритма сортировки слиянием».

В сущности, как мы увидим немного позже в этой главе, в действительности можно разместить средний (по значению) элемент в неотсортированном массиве за линейное время, но этот алгоритм достаточно сложен, и скрытая постоянная в нотации $O(\cdot)$ достаточно велика для того, чтобы сделать итоговый алгоритм сортировки почти неприменимым. В практической деятельности большинство программистов применяет упрощенные способы, например выбор первого или последнего элемента массива. В этом случае позиция r может принимать любое значение между 1 и n , поэтому получаем:

$$T(n) = \max_{1 \leq r \leq n} (T(r-1) + T(n-r) + O(n)).$$

В наихудшем случае эти две подзадачи являются абсолютно несбалансированными как при $r = 1$, так и при $r = n$, и рекуррентное выражение приобретает следующий вид: $T(n) \leq T(n-1) + O(n)$.

Решением является $T(n) = O(n^2)$.

Другой общеизвестный эвристический алгоритм называется «среднее из трех (значений)» (median of three) – выбираются три элемента (обычно в начале, в середине и в конце массива), и среднее значение этих трех элементов принимается как опорный элемент. Хотя этот эвристический алгоритм на практике несколько более эффективен, чем простой выбор одного элемента, особенно если массив уже (почти) отсортирован, тем не менее мы можем получить не самое удачное значение $r = 2$ или $r = n - 1$ в наихудшем случае. При использовании эвристического алгоритма «среднее из трех» рекуррентное выражение принимает вид $T(n) \leq T(1) + T(n-2) + O(n)$, решением которого остается $T(n) = O(n^2)$.

На интуитивном уровне понятно, что опорный элемент «обычно» должен находиться где-то в середине массива, например в диапазоне между $n/10$ и $9n/10$. Это наблюдение позволяет предположить, что «в среднем» время выполнения алгоритма должно составлять $O(n \log n)$. Несмотря на то что это интуитивное предположение можно формализовать, большинство наиболее распространенных методов формализации принимает абсолютно нереалистичное предположение о том, что все перестановки в исходном массиве почти равнозначны. Данные в реальном мире могут быть случайными (случайно распределенными), но это совсем не та случайность, которую можно предсказать заранее, а кроме того, данные определенно не являются равномерно распределенными⁷.

Кроме того, иногда по каким-то причинам принимается время выполнения «в наилучшем случае». Мы так делать не будем.

⁷ С другой стороны, если индекс опорного элемента r единообразно выбирается случайным образом, то алгоритм QuickSort выполняется за время $O(n \log n)$ с высокой вероятностью для каждого возможного входного массива. Главное различие состоит в том, что эта случайность управляется самим алгоритмом, а не каким-то всемогущим злоумышленником, который передает входные данные после чтения нашего кода. К сожалению, анализ алгоритма быстрой сортировки со случайным выбором опорного элемента не относится к теме этой книги, но вы можете найти конспекты лекций по этой теме здесь: <http://algorithms.wtf/>.

1.6. Шаблон

Оба алгоритма сортировки – сортировки слиянием и быстрой сортировки – соответствуют общему трехшаговому шаблону, который называется «разделяй и властвуй» (divide and conquer).

1. **Разделить** предлагаемый вариант задачи на несколько более мелких частных случаев в точности той же задачи.
2. **Делегировать** (т. е. передать) каждый более мелкий частный случай для выполнения Фее Рекурсии.
3. **Объединить** решения для всех более мелких частных случаев в итоговое решение предложенного варианта задачи.

Если размер какого-либо частного случая становится меньше некоторого постоянного порогового значения, то рекурсия отменяется, и этот частный случай задачи решается напрямую «в лоб» (методом грубой силы) за постоянное время.

Доказательство корректности алгоритма «разделяй и властвуй» почти всегда требует применения индукции. Анализ времени выполнения требует определения и решения рекуррентного выражения, которое обычно (но, к сожалению, не всегда) можно решить с использованием рекурсивных деревьев.

1.7. Рекурсивные деревья

Так что же такое эти «рекурсивные деревья», которые я так часто упоминаю? Рекурсивные деревья (recursion trees) – это простой и наглядный инструмент общего назначения для решения рекуррентных выражений типа «разделяй и властвуй». Рекурсивное дерево – это дерево с корнем (корневое дерево) с одним узлом (node) для каждой рекурсивной подзадачи. Значением (value) каждого узла является количество времени, затрачиваемого на решение соответствующей подзадачи, исключая рекурсивные вызовы. Таким образом, общее время выполнения алгоритма представляет собой сумму значений всех узлов этого дерева.

Чтобы придать этой концепции более определенную форму, представьте некоторый алгоритм типа «разделяй и властвуй», который затрачивает время $O(f(n))$ на нерекурсивную работу, а затем выполняет r рекурсивных вызовов, каждый для задачи размером n/c . С точностью до постоянных множителей (которые можно скрыть в формате записи $O(\cdot)$) время выполнения этого алгоритма управляется следующим рекуррентным выражением:

$$T(n) = r T(n/c) + f(n).$$

Корень рекурсивного дерева для $T(n)$ имеет значение $f(n)$ и r потомков, каждый из которых является корнем (рекурсивно определяемого) рекурсивного дерева для $T(n/c)$. В итоге рекурсивное дерево представляет со-

бой r -е дерево, в котором каждый узел на глубине d содержит значение $f(n/c^d)$. (Можно смело предположить, что n является целочисленной степенью c , поэтому n/c^d всегда представлено целым числом, хотя в действительности это не имеет значения.)

На практике я рекомендую изображать графически первые два или три уровня такого рекурсивного дерева, как показано на рис. 1.9.

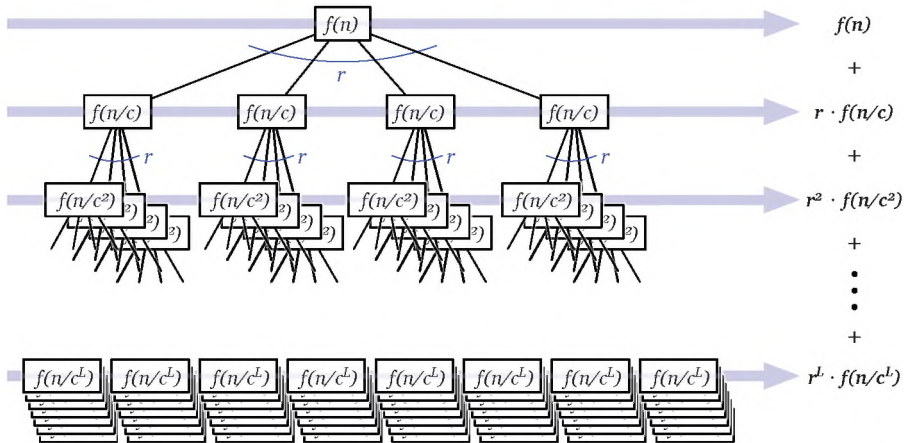


Рис. 1.9. Рекурсивное дерево для рекуррентного выражения $T(n) = rT(n/c) + f(n)$

Листья этого рекурсивного дерева соответствуют простейшему (базовому) случаю (нескольким простейшим случаям) рекуррентного выражения. Поскольку мы определяем только асимптотические границы, точный простейший случай в действительности не важен, и можно без сомнений предположить, что $T(n) = 1$ для всех $n \leq n_0$, где n_0 – произвольно выбранная постоянная величина. В сущности, можно выбрать любое значение n_0 , которое наиболее удобно для анализа. Для рассматриваемого здесь примера я выбираю $n_0 = 1$.

Далее $T(n)$ является суммой всех значений в рекурсивном дереве. Эту сумму можно вычислить, рассматривая поочередно каждый уровень дерева. Для любого целого i каждый i -й уровень дерева содержит в точности r^i узлов, и каждый узел содержит значение $f(n/c^i)$. Следовательно,

$$T(n) = \sum_{i=0}^L r^i \cdot f(n/c^i), \quad (\Sigma)$$

где L – глубина рекурсивного дерева. В рассматриваемом здесь простейшем (базовом) случае $n_0 = 1$ сразу же предполагается $L = \log_c n$, так как $n/c^L = n_0 = 1$. Отсюда следует, что число листьев в этом рекурсивном дереве в точности равно $r^L = r^{\log_c n} = n^{\log_c r}$. Таким образом, самый последний член в сумме по уровням (Σ) равен $n^{\log_c r} \cdot f(1) = O(n^{\log_c r})$, поскольку $f(1) = O(1)$.

Существует три типовых случая, в которых последовательность суммирования по уровням (Σ) особенно легко вычисляется:

- *убывание* – если последовательность является экспоненциально убывающей – каждый член представлен постоянным множителем, меньшим предыдущего члена, – то $T(n) = O(f(n))$. В этом случае сумма главным образом определяется значением в корне рекурсивного дерева;
- *равенство* – если все члены последовательности равны, то сразу же получаем $T(n) = O(f(n)) \cdot L = O(f(n) \log n)$ (постоянная величина с исключается из формы записи $O(\cdot)$);
- *возрастание* – если последовательность является экспоненциально возрастающей – каждый член представлен постоянным множителем, большим предыдущего члена, – то $T(n) = O(n^{\log_c r})$. В этом случае сумма главным образом определяется числом листьев в рекурсивном дереве.

В первом и третьем случаях только максимальное значение члена геометрической последовательности (прогрессии) имеет значение, а все прочие члены «поглощаются» формой записи $O(\cdot)$. В случае убывания даже нет необходимости вычислять L , асимптотическая верхняя граница должна оставаться постоянной, даже если бы рекурсивное дерево было бесконечным.

Простейший пример: если изобразить графически несколько первых уровней рекурсивного дерева для (упрощенного) рекуррентного выражения сортировки слиянием $T(n) = 2T(n/2) + O(n)$, то обнаружим, что все уровни равны, и сразу же получаем $T(n) = O(n \log n)$.

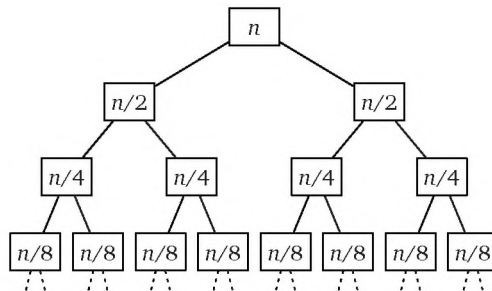


Рис. 1.10. Рекурсивное дерево для алгоритма сортировки слиянием

Методику рекурсивного дерева можно также использовать для алгоритмов, в которых рекурсивные подзадачи имеют различные размеры. Например, если бы можно было каким-то способом так реализовать алгоритм быстрой сортировки, чтобы опорный элемент всегда располагался в средней трети сортируемого массива, то время выполнения в наихудшем случае соответствовало бы рекуррентному выражению:

$$T(n) \leq T(n/3) + T(2n/3) + O(n).$$

Возможно, это рекуррентное выражение выглядит не слишком привлекательно, но в действительности его достаточно легко приручить. Если

графически изобразить несколько уровней итогового рекурсивного дерева, то можно быстро заметить, что сумма значений на любом уровне не превышает n , – на более глубоких уровнях могут быть пропущены некоторые узлы, – а все дерево в целом имеет глубину $\log_{3/2} n = O(\log n)$. Из этого сразу же следует $T(n) = O(n \log n)$. (Кроме того, число полных уровней в этом рекурсивном дереве равно $\log_3 n = \Omega(\log n)$, поэтому проводимый здесь традиционный анализ с запасом прочности можно улучшить с постоянным коэффициентом, но не более того, что для достижения нашей цели вообще не имеет никакого значения.) Тот факт, что рекурсивное дерево не сбалансировано, просто ничего не значит.

Более сложный пример: наихудший случай рекуррентного выражения для алгоритма быстрой сортировки $T(n) = T(n-1) + T(1) + O(n)$ дает абсолютно несбалансированное рекурсивное дерево, в котором единственным потомком каждого внутреннего узла является лист (т. е. узел без потомков). Сумма по уровням не попадает ни в одну из определенных выше трех категорий по умолчанию, но остается возможность вывода решения $T(n) = O(n^2)$, если учесть, что значение каждого уровня не превышает n и в дереве имеется не более n уровней. (И в этом случае такой традиционный анализ с запасом прочности ограничен, потому что каждый из $n/2$ уровней содержит значение, не меньшее $n/2$.)

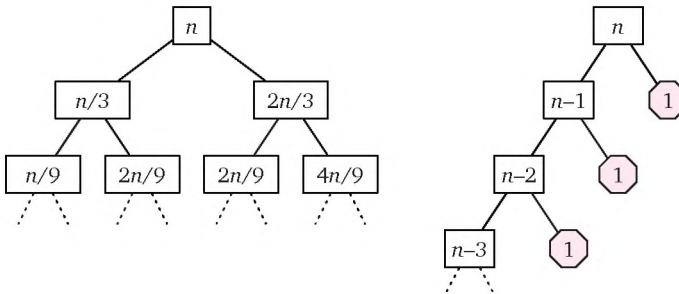


Рис. 1.11. Рекурсивные деревья для алгоритма быстрой сортировки с правильно выбранными опорными элементами (слева) и для наихудшего выбора опорного элемента (справа)

♥ Исключение нижних и верхних границ – это правильный подход, даю честное слово

Возможно, внимательные читатели обратили внимание на то, что в приведенном выше анализе важные подробности скрыты от их глаз. Время выполнения алгоритма сортировки сливанием в действительности не должно непременно подчиняться рекуррентному выражению $T(n) = 2T(n/2) + O(n)$, ведь размер исходного массива n может быть нечетным, а что могла бы означать сортировка массива размером $42\frac{1}{2}$ или $17\frac{7}{8}$? Истинное рекуррентное выражение для алгоритма сортировки сливанием выглядит немного сложнее:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n).$$

Разумеется, можно было бы проверить, что $T(n) = O(n \log n)$, используя индукцию, но для этого необходимо выполнить весьма большой объем вычислений. К счастью, существует простая методика исключения нижних и верхних границ из рекуррентных выражений, которая называется преобразованием области (domain transformation).

- Во-первых, поскольку мы выделяем верхнюю границу, то можем смело зависить значение $T(n)$ в первый раз, притворившись, что размеры двух решаемых подзадач равны, и во второй раз при удалении верхней границы (ceiling)⁸:

$$T(n) \leq 2T(\lceil n/2 \rceil) + n \leq 2T(n/2 + 1) + n.$$

- Во-вторых, мы определяем новую функцию $S(n) = T(n + \alpha)$, выбирая постоянную α так, чтобы функция $S(n)$ соответствовала более простому рекуррентному выражению $S(n) \leq 2S(n/2) + O(n)$. Чтобы найти подходящую константу α , выводим рекуррентную формулу для S из известного нам рекуррентного выражения для T :

$$\begin{aligned} S(n) &= T(n + \alpha) && \text{[определение } S\text{]} \\ &\leq 2T(n/2 + \alpha/2 + 1) + n + \alpha && \text{[рекуррентное выражение для } T\text{]} \\ &= 2S(n/2 - \alpha/2 + 1) + n + \alpha && \text{[определение } S\text{]}. \end{aligned}$$

Выбор постоянного значения $\alpha = 2$ упрощает эту рекуррентную формулу до $S(n) \leq 2S(n/2) + n + 2$ – это в точности то, что нам нужно.

- Наконец, по методу рекурсивного дерева подразумевается $S(n) = O(n \log n)$, следовательно,

$$T(n) = S(n - 2) = O((n - 2) \log(n - 2)) = O(n \log n),$$

что в точности совпадает с предположением.

Аналогичные преобразования области можно использовать для исключения нижних и верхних границ и даже членов более низких степеней из любого рекуррентного выражения типа «разделяй и властвуй». Но сейчас, когда это стало понятным, нет необходимости снова углубляться в подробности. Всюду в дальнейшем при встрече с любым рекуррентным выражением типа «разделяй и властвуй» я буду без лишних объяснений исключать нижние и верхние границы и члены более низких степеней, и вам рекомендую делать то же самое.

⁸ Формально мы интерпретируем T как функцию от действительных, а не только целых чисел, что соответствует приведенному выше рекуррентному выражению в базовом частном случае $T(n) = C$ для всех $n \leq n_0$, для некоторых действительных чисел $C \geq 0$ и $n_0 > 0$, значения которых неважны. Если приходится рассматривать n как целое число, то $T(n)$ совпадает с временем выполнения алгоритма при исходных данных размером n , но это также не имеет значения.

♥ 1.8. Линейный алгоритм выбора

Во время обсуждения алгоритма быстрой сортировки я мимоходом отметил, что можно найти среднее значение в неотсортированном массиве за линейное время. Первый подобный алгоритм разработали Мануэл Блум (Manuel Blum), Боб Флойд (Bob Floyd), Вон Пратт (Vaughan Pratt), Рон Ривест (Ron Rivest) и Боб Тарьян (Bob Tarjan) в начале 1970 гг. Этот алгоритм в действительности решает более общую задачу выбора k -го наименьшего элемента из n -элементного массива при условии, что массив и целое число k являются входными данными, с использованием варианта алгоритма под названием quickselect (быстрый выбор) или one-armed quicksort (однорукая быстрая сортировка). Алгоритм быстрого выбора впервые был описан Тони Хоаром (Tony Hoare) в 1961 г. буквально на той же странице, на которой он впервые опубликовал алгоритм быстрой сортировки.

Алгоритм быстрого выбора

Обобщенный алгоритм быстрого выбора определяет опорный элемент, разделяет массив, применяя ту же подпрограмму Partition, что и алгоритм QuickSort, затем выполняет рекурсивную процедуру поиска только в одном из двух подмассивов, а именно в том, который содержит k -й наименьший элемент исходного массива с входными данными. Псевдокод для алгоритма быстрого выбора показан на рис. 1.12.

```

QuickSelect(A[1..n], k):
  if n = 1
    return A[1]
  else
    Choose a pivot element A[p]
    r ← Partition(A[1..n], p)
    if k < r
      return QuickSelect(A[1..r - 1], k)
    else if k > r
      return QuickSelect(A[r + 1..n], k - r)
    else
      return A[r]

```

Рис. 1.12. Алгоритм быстрого выбора
или однорукой быстрой сортировки

Этот алгоритм обладает двумя важными свойствами. Первое, как и для алгоритма быстрой сортировки: корректность алгоритма быстрого выбора не зависит от способа выбора опорного элемента. Второе: даже если нас действительно интересует только выбор средних значений (medians) (особый случай $k = n/2$), стратегия рекурсии Хоара требует рассмотрения более общей задачи выбора: среднее значение входного массива $A[1..n]$ почти

никогда не является средним значением любого из двух меньших подмассивов $A[1..r-1]$ или $A[r+1..n]$.

Время выполнения алгоритма QuickSelect в наихудшем случае соответствует рекуррентному выражению, подобному выражению для алгоритма QuickSort. Нам неизвестно значение r , также неизвестно, в каком из двух подмассивов будет выполняться рекурсивный поиск, поэтому необходимо предположить наихудший случай.

$$T(n) \leq \max_{1 \leq r \leq n} \max\{T(r-1), T(n-r)\} + O(n).$$

Это рекуррентное выражение можно немного упростить, если ввести переменную ℓ , обозначающую длину рекурсивной подзадачи:

$$T(n) \leq \max_{0 \leq \ell \leq n-1} T(\ell) + O(n).$$

Если выбранный опорный элемент всегда является либо наименьшим, либо наибольшим элементом в массиве, то рекуррентное выражение упрощается до $T(n) = T(n-1) + O(n)$, из которого следует $T(n) = O(n^2)$. (Рекурсивное дерево для этого рекуррентного выражения – простой путь.)

Правильные опорные элементы

Можно было бы избежать этого квадратичного поведения в наихудшем случае, если бы существовала возможность каким-то магическим способом выбрать правильный опорный элемент, что означает $\ell \leq \alpha n$ для некоторой постоянной $\alpha < 1$. В этом случае рекуррентное выражение можно было бы упростить до

$$T(n) \leq T(\alpha n) + O(n).$$

Это рекуррентное выражение разворачивается в убывающую геометрическую прогрессию, в которой доминирующим является ее наибольший член, поэтому $T(n) = O(n)$. (И в этом случае рекурсивным деревом является простой путь. Постоянная в нотации времени выполнения $O(n)$ зависит от постоянной α .)

Другими словами, если бы можно было каким-либо способом быстро найти элемент, самый близкий к среднему значению, за линейное время, то можно было бы определить точное среднее значение за линейное время. Так что теперь все, что нам необходимо, – это Приблизительная Медианная Фея (Approximate Median Fairy). Алгоритм Блума–Флойда–Пратта–Ривест–Тарьяна выбирает правильный опорный элемент для быстрого выбора, рекурсивно вычисляя среднее значение тщательно подобранного подмножества входного массива. Поэтому Приблизительная Медианная Фея – это просто переодетая Фея Рекурсия.

Если говорить более определенно, мы разделяем входной массив на $\lceil n/5 \rceil$ блоков, каждый из которых содержит в точности пять элементов, возможно, кроме последнего. (Если самый последний блок неполон, просто появля-

ется несколько бесконечностей.) Вычисляется среднее значение в каждом блоке прямым перебором (brute force), эти средние значения объединяются в новый массив $M[1..[n/5]]$, затем рекурсивно вычисляется среднее значение этого нового массива. Далее среднее значение блока средних значений (в приведенном ниже псевдокоде оно обозначено *mom* – median of medians) используется как опорный элемент в алгоритме быстрого выбора.

```

MomSelect(A[1..n], k):
  if n ≤ 25                                <<Или любое другое число>>
    use brute force                        <<использовать прямой перебор>>
  else
    m ← [n/5]
    for i ← 1 to m
      M[i] ← MedianOfFive(A[5i - 4..5i])  <<Прямой перебор>>
    mom ← MomSelect(M[1..m], [m/2])       <<Рекурсия!>>
    r ← Partition(A[1..n], mom)
    if k < r
      return MomSelect(A[1..r - 1], k)    <<Рекурсия!>>
    else if k > r
      return MomSelect(A[r + 1..n], k - r) <<Рекурсия!>>
    else
      return mom

```

Алгоритм MomSelect использует рекурсию для достижения двух различных целей: в первый раз для выбора опорного элемента (*mom*), во второй раз для поиска требуемых элементов по одну сторону от выбранного опорного элемента.

Анализ

Но почему этот алгоритм работает так быстро? Первое главное (и верное) соображение: *среднее значение блока средних значений является правильным выбором опорного элемента*. Опорный элемент *mom* больше $\lceil [n/5]/2 \rceil - 1 \approx n/10$ блока средних значений, а каждое среднее значение в блоке больше, чем два других элемента в этом блоке. Таким образом, *mom* больше чем как минимум $3n/10$ элементов в исходном массиве, и при учете симметрии *mom* меньше чем как минимум $3n/10$ элементов. Следовательно, в наихудшем случае второй рекурсивный вызов выполняет поиск в массиве размером не более $7n/10$.

Поведение этого алгоритма можно наглядно представить, изобразив графически входной массив как решетку (сетку) размером $5 \times n/5$, в которой каждый столбец представляет пять смежных элементов. Для большей наглядности предположим, что каждый столбец сортируется сверху вниз, а затем столбцы сортируются по средним элементам. (Обратите особое

внимание: в действительности алгоритм не делает этого!) В таком графическом представлении среднее значение блока средних значений (median-of-medians) – это элемент, расположенный ближе всего к центру решетки, как показано на рис. 1.13.

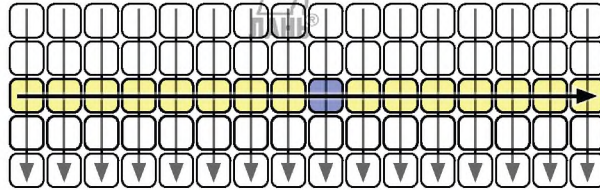


Рис. 1.13

Левая половина первых трех рядов этой решетки содержит $3n/10$ элементов, каждый из которых меньше *tot*. Если элемент, который мы ищем, больше *tot*, то алгоритм отбрасывает все элементы, которые меньше *tot*, включая эти $3n/10$ элементов, перед началом рекурсии. Таким образом, входные данные для рекурсивной подзадачи содержат не более $7n/10$ элементов. С учетом фактора симметрии предполагается, что если искомым элемент меньше *tot*, то отбрасываются как минимум $3n/10$ элементов, которые больше *tot*, поэтому входные данные для рекурсивной подзадачи содержат не более $7n/10$ элементов (см. рис. 1.14).

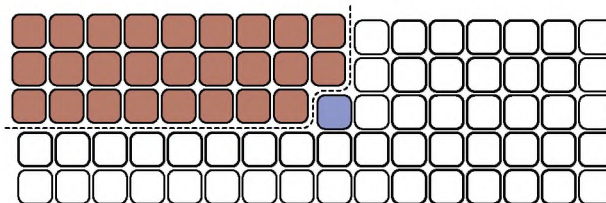


Рис. 1.14

Теперь понятно, что *tot* является правильным опорным элементом, но рассматриваемый здесь алгоритм продолжает выполнять два рекурсивных вызова вместо одного – как при этом доказать, что время выполнения линейное? Второе главное соображение состоит в том, что общий размер двух рекурсивных подзадач представляет собой постоянную величину, меньшую, чем размер начального входного массива. Время выполнения этого алгоритма в наихудшем случае определяется рекуррентным выражением:

$$T(n) \leq T(n/5) + T(7n/10) + O(n).$$

Если графически изобразить рекурсивное дерево для этого рекуррентного выражения, то можно увидеть, что общий объем работы на каждом уровне рекурсивного дерева не превышает $9/10$ от общего объема работы на предыдущем уровне. Следовательно, суммы уровней экспоненциально убывают, что дает нам решение $T(n) = O(n)$. (И в этом случае тот факт, что

рекурсивное дерево является несбалансированным, не имеет никакого значения.) Ура! Спасибо, *mom*!

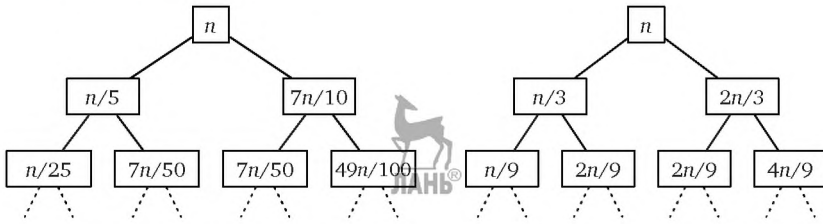


Рис. 1.15. Рекурсивные деревья для алгоритма MomSelect и аналогичного алгоритма выбора с блоками размером 3 элемента

Проверка достоверности

Здесь многие студенты задают вопрос о загадочном постоянном числе 5. Почему был выбран именно этот конкретный размер блока? Ответ: 5 – наименьший нечетный размер блока, который обеспечивает экспоненциальное убывание при анализе методом рекурсивного дерева. (Четные размеры блока вносят дополнительные сложности.) Если бы использовались блоки размером 3, то рекуррентное выражение времени выполнения выглядело бы так:

$$T(n) \leq T(n/3) + T(2n/3) + O(n).$$

Но мы уже видели это рекуррентное выражение ранее. Каждый уровень рекурсивного дерева содержит суммарное значение, не превышающее n , а глубина рекурсивного дерева равна $\log_{3/2} n = O(\log n)$, следовательно, решением этого рекуррентного выражения является $T(n) \leq O(n \log n)$. (При этом анализ является строгим, потому что рекурсивное дерево содержит $\log_3 n$ полных уровней.) Выбор среднего значения из блока средних значений с использованием 3-элементных блоков выполняется не быстрее, чем сортировка.

Более тщательный анализ позволяет узнать, что постоянная, скрытая в форме записи $O(\)$, достаточно велика, даже если считать только операции сравнения. Выбор среднего значения из 5 элементов требует не более 6 операций сравнения, поэтому необходимо максимум $6n/5$ операций сравнения для подготовки решения рекурсивной подзадачи. Простейший вариант разделения массива после рекурсивного вызова должен потребовать $n - 1$ сравнений, но нам уже известно, что $3n/10$ элементов больше, чем опорный элемент, и $3n/10$ элементов меньше, чем опорный элемент, поэтому процедура разделения в действительности потребует только $2n/5$ дополнительных операций сравнения. Следовательно, более точное рекуррентное выражение для числа сравнений в наихудшем случае будет таким:

$$T(n) \leq T(n/5) + T(7n/10) + 8n/5.$$

Метод рекурсивного дерева позволяет вывести выражение для верхней границы:

$$T(n) = \left(\frac{8n}{5}\right) \sum_{i \geq 0} \left(\frac{9}{10}\right)^i = \frac{8n}{5} \cdot 10 = 16n.$$

На практике процедура выбора среднего значения из блока средних значений выполняется не так медленно, как прогнозирует анализ наихудшего случая – наихудший случай определения опорного элемента на каждом уровне рекурсии весьма маловероятен, – тем не менее эта процедура остается более медленной, чем сортировка даже относительно больших массивов⁹.

1.9. Быстрое умножение

В предыдущей главе мы рассматривали два древних алгоритма для умножения двух n -разрядных чисел за время $O(n^2)$: решеточное умножение в начальной школе и алгоритм египетских крестьян.

Можно получить более эффективный алгоритм, разделяя массивы цифр на половины и применяя следующее тождество:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd.$$

Из этого рекуррентного выражения сразу же выводится следующий алгоритм «разделяй и властвуй» для умножения двух n -разрядных чисел x и y . Каждое из четырех вспомогательных произведений ac , bc , ad и bd вычисляется рекурсивно, но умножения в последней строке не являются рекурсивными, потому что на степень числа 10 можно умножать, сдвигая цифры влево и заполняя освобождающиеся разряды нулями для сохранения корректности числа. Все это выполняется за время $O(n)$.

SplitMultiply(x, y, n):

```

if n = 1
    return x · y
else
    m ← ⌊n/2⌋
    a ← ⌊x/10m⌋; b ← x mod 10m
    c ← ⌊y/10m⌋; d ← y mod 10m
    e ← SplitMultiply(a, c, m)
    f ← SplitMultiply(b, d, m)
    g ← SplitMultiply(b, c, m)
    h ← SplitMultiply(a, d, m)
    return 102me + 10m(g + h) + f

```

$\langle\langle x = 10^m a + b \rangle\rangle$
 $\langle\langle y = 10^m c + d \rangle\rangle$

⁹ В действительности правильным практическим способом выбора опорного элемента является его выбор равномерным и случайным образом. Тогда ожидаемое число операций сравнения, требуемых для нахождения среднего значения, не превышает $4n$. Более подробно см. мои конспекты лекций по рандомизированным алгоритмам здесь: <http://algorithms.wtf>.

Корректность этого алгоритма легко доказывается методом индукции. Время его выполнения определяется следующим рекуррентным выражением:

$$T(n) = 4T([n/2]) + O(n).$$

Метод рекурсивного дерева позволяет преобразовать это рекуррентное выражение в возрастающую геометрическую прогрессию, для которой $T(n) = O(n^{\log_2 4}) = O(n^2)$. По существу, этот алгоритм умножает каждую цифру числа x на каждую цифру числа y точно так же, как алгоритм решеточного умножения. Кажется, этот вариант не сработал. Очень жаль. Это была неплохая идея.

В середине 1950-х гг. Андрей Николаевич Колмогоров, один из крупнейших математиков XX в., публично предположил, что не существует алгоритма умножения двух n -значных чисел за субквадратичное время (гипотеза n^2). В 1960 г. Колмогоров организовал в Московском университете семинар, на котором еще раз повторил свою «гипотезу n^2 » и сформулировал несколько связанных с этим предположением задач, которые должны были обсуждаться на следующих семинарах. Почти ровно через неделю 23-летний студент Анатолий Карацуба представил Колмогорову блестящий контрпример. Сам Карацуба вспоминает об этом так:

«После семинара я рассказал Колмогорову о новом алгоритме и об опровержении гипотезы n^2 . Колмогоров был очень взволнован, так как это противоречило его весьма правдоподобной гипотезе. На следующем заседании семинара Колмогоров сам сообщил участникам о моем методе, и на этом семинар завершил работу».

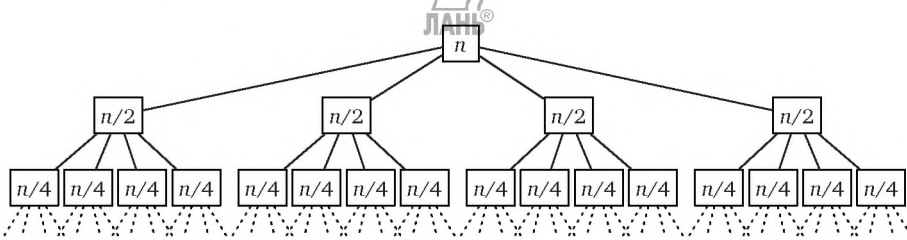


Рис. 1.16. Рекурсивное дерево для простейшего умножения методом «разделяй и властвуй»

Карацуба заметил, что средний коэффициент $bc + ad$ можно вычислить по двум другим коэффициентам ac и bd , используя только одну дополнительную операцию умножения с применением следующего алгебраического тождества:

$$ac + bd - (a - b)(c - d) = bc + ad.$$

Этот прием позволяет заменить четыре рекурсивных вызова в приведенном выше алгоритме на три рекурсивных вызова, как показано ниже:


```
FastMultiply(x, y, n):
```

```
  if n = 1
```

```
    return x · y
```

```
  else
```

```
    m ← ⌊n/2⌋
```

```
    a ← ⌊x/10m⌋; b ← x mod 10m
```

$\langle\langle x = 10^m a + b \rangle\rangle$

```
    c ← ⌊y/10m⌋; d ← y mod 10m
```

$\langle\langle y = 10^m c + d \rangle\rangle$

```
    e ← FastMultiply(a, c, m)
```

```
    f ← FastMultiply(b, d, m)
```

```
    g ← FastMultiply(a - b, c - d, m)
```

```
    return 102me + 10m(e + f - g) + f
```



Время выполнения алгоритма быстрого умножения Карацубы FastMultiply определяется рекуррентным выражением:

$$T(n) \leq 3T(\lfloor n/2 \rfloor) + O(n).$$

И в этом случае метод рекурсивного дерева позволяет преобразовать приведенное рекуррентное выражение в возрастающую геометрическую прогрессию, но новое решение определяет только лишь $T(n) = O(n^{\log_2 3}) = O(n^{1.58496})$, а это существенное улучшение, по сравнению с ранее определенной верхней границей квадратичного времени¹⁰. Можно считать, что алгоритм Карацубы положил начало проектированию и анализу алгоритмов как отдельной формальной области научных исследований.

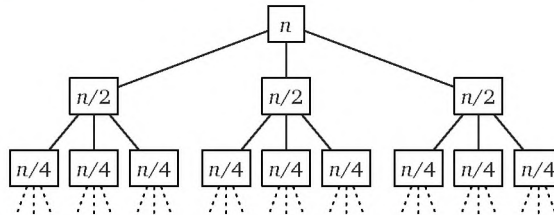


Рис. 1.17. Рекурсивное дерево для алгоритма умножения Карацубы «разделяй и властвуй»

Можно продолжить развитие идеи Карацубы, разделяя числа на большее количество фрагментов и комбинируя эти фрагменты более сложными способами, чтобы получить еще более быстрые алгоритмы умножения. Андрей Тоом открыл бесконечное семейство алгоритмов, которые разделяют любое целое число на k частей, каждая часть содержит n/k цифр, затем вычисляется произведение с использованием всего лишь $2k - 1$ ре-

¹⁰ Мое описание немного упрощает реальный исторический факт. В действительности Карацуба предложил алгоритм, основанный на формуле $(a + b)(c + d) - ac - bd = bc + ad$. Этот алгоритм также выполняется за время $O(n \lg 3)$, но настоящее рекуррентное выражение несколько более сложное: $a - b$ и $c - d$ остаются m -разрядными числами, но $a + b$ и $c + d$ могут содержать $m + 1$ цифр (каждое). Допущенное здесь упрощение стало возможным благодаря Дональду Кнуту (Donald Knuth).

курсивных операций умножения. В дальнейшем алгоритм Тоома был упрощен Стивеном Куком (Stephen Cook) в его кандидатской диссертации. Для любого фиксированного k алгоритм Тоома–Кука выполняется за время $O(n^{1+1/(\lg k)})$, где скрытая постоянная в нотации $O(\cdot)$ зависит от k .

И самое главное: эта стратегия «разделяй и властвуй» привела Гаусса (Gauss) (это действительно так) к открытию быстрых преобразований Фурье (fast Fourier transform)¹¹. Основной (базовый) алгоритм БПФ выполняется за время $O(n \log n)$, но применение БПФ для умножения целых чисел влечет за собой некоторые небольшие дополнительные издержки (накладные расходы). Первый алгоритм умножения целых чисел на основе БПФ, опубликованный Арнольдом Шёнхаге (Arnold Schönhage) и Фолькером Штрассеном (Volker Strassen) в 1971 г., выполняется за время $O(n \log n \log \log n)$. Алгоритм Шёнхаге–Штрассена оставался теоретически самым быстрым алгоритмом умножения целых чисел в течение нескольких десятилетий, до того как Мартин Фюрер (Martin Fürer) обнаружил первое из длинной последовательности технических усовершенствований. Наконец, в 2019 г. Дэвид Харви (David Harvey) и Йорис ван дер Хувен (Joris van der Hoeven) опубликовали алгоритм, выполняемый за время $O(n \log n)$ ¹².

1.10. Возведение в степень

Пусть существует число a и положительное целое число n , и предположим, что необходимо вычислить a^n . Простейший стандартный метод – обычный цикл `for`, который выполняет $n - 1$ операций умножения a самого на себя:

```
SlowPower(a, n):
    x ← a
    for i ← 2 to n
        x ← x · a
    return x
```

Этот итеративный алгоритм требует n операций умножения.

Входной параметр a может быть целым, или рациональным числом, или числом с плавающей точкой. В действительности это значение вообще не обязательно должно быть числом при условии, что мы знаем, как его нужно умножать. Например, этот алгоритм можно использовать для вычисления показателей степени некоторого конечного числа (операция, весьма часто используемая в криптографических алгоритмах) или для вычисления степеней матриц (операция, применяемая для вычисления рекуррентных выражений и для определения наикратчайших путей в графах). Поскольку нам неизвестно, какой тип объекта умножается, невозможно узнать, сколь-

¹¹ О быстрых преобразованиях Фурье см. конспекты лекций здесь: <http://algorithms.wtf>.

¹² Алгоритм Шёнхаге–Штрассена действительно является самым быстрым практически применяемым алгоритмом для умножения целых чисел с более чем 75 000 разрядов (цифр). Более поздние алгоритмы Фюрера, Харви и ван дер Хувена должны работать быстрее «на практике» только для целых чисел с количеством разрядов, превышающим количество частиц во Вселенной.

ко потребуются элементарных операций умножения, поэтому мы вынуждены анализировать время выполнения, оценивая количество операций умножения.

Существует гораздо более быстрый метод «разделяй и властвуй», впервые предложенный индийским мастером стихосложения (просодии) Пингала (Piṅgala) во II в. до н. э., с применением следующей простой рекурсивной формулы:

$$a^n = \begin{cases} 1 & , \text{если } n = 0; \\ (a^{n/2})^2 & , \text{если } n > 0 \text{ и } n \text{ четное;} \\ (a^{\lfloor n/2 \rfloor})^2 \cdot a & \text{во всех прочих случаях.} \end{cases}$$

```
PiṅgalaPower(a, n):
  if n = 1
    return a
  else
    x ← PiṅgalaPower(a, ⌊n/2⌋)
    if n is even
      return x · x
    else
      return x · x · a
```

Общее количество операций умножения, выполняемых этим алгоритмом, определяется рекуррентным выражением $T(n) \leq T(n/2) + 2$. Метод рекурсивного дерева сразу же дает решение $T(n) = O(\log n)$.

Почти такой же алгоритм возведения в степень можно вывести и непосредственно из алгоритма египетского крестьянского умножения, описанного в предыдущей главе, если заменить операцию сложения операцией умножения (и при этом, разумеется, операция удвоения заменяется операцией возведения в квадрат).

$$a^n = \begin{cases} 1 & , \text{если } n = 0; \\ (a^2)^{n/2} & , \text{если } n > 0 \text{ и } n \text{ четное;} \\ (a^2)^{\lfloor n/2 \rfloor} \cdot a & \text{во всех прочих случаях.} \end{cases}$$

```
PeasantPower(a, n):
  if n = 1
    return a
  else if n is even
    return PeasantPower(a², n/2)
  else
    return PeasantPower(a², ⌊n/2⌋) · a
```

Этот алгоритм, который можно вполне обоснованно назвать «возведение в квадрат и усреднение», также выполняет всего лишь $O(\log n)$ операций умножения.

Оба приведенных выше алгоритма асимптотически оптимальны. Любой алгоритм, который вычисляет a^n , непременно должен выполнить не менее $\Omega(\log n)$ операций умножения, потому что каждая операция умножения как минимум удваивает наибольшую степень, вычисленную до сих пор. В действительности если n является степенью двойки, то оба этих алгоритма требуют в точности $\log_2 n$ операций умножения и являются строго оптимальными. Тем не менее существуют немного более быстрые методы для других значений n . Например, оба алгоритма `PiñgalaPower` и `PeasantPower` вычисляют a^{15} , используя шесть операций умножения, хотя на самом деле необходимо выполнить только пять:

- `Piñgala`: $a \rightarrow a^2 \rightarrow a^3 \rightarrow a^6 \rightarrow a^7 \rightarrow a^{14} \rightarrow a^{15}$;
- `Peasant`: $a \rightarrow a^2 \rightarrow a^4 \rightarrow a^8 \rightarrow a^{12} \rightarrow a^{14} \rightarrow a^{15}$;
- Оптимальный алгоритм: $a \rightarrow a^2 \rightarrow a^3 \rightarrow a^5 \rightarrow a^{10} \rightarrow a^{15}$.

Существует ли абсолютно минимальное число операций умножения для эффективного вычисления заданной степени n – этот вопрос остается открытым в течение весьма долгого времени.

Упражнения

Ханойские башни

1. Доказать, что исходный рекурсивный алгоритм «Ханойская башня» выполняет в точности ту же самую последовательность перемещений – те же диски, на тех же стержнях, в том же порядке, – что и каждый из описанных ниже нерекурсивных алгоритмов. Стержни помечены цифрами 0, 1 и 2, а задача состоит в перемещении стопки из n дисков со стержня 0 на стержень 2 (как показано на рис. 1.1 на стр. 49).

- (a) Если n – четное число, то поменять местами стержни 1 и 2. На i -м шаге выполнить единственное допустимое перемещение, которое исключает стержень $i \bmod 3$. Если допустимых перемещений не существует, то все диски находятся на стержне $i \bmod 3$, и задача решена.
- (b) Первым ходом переместить диск 1 на стержень 1, если n – четное число, или на стержень 2, если n – нечетное число. Затем повторно выполнять единственное допустимое перемещение диска, отличного от участвующего в предыдущем перемещении. Если допустимых перемещений не существует, то задача решена.

(с) Предположим, что диски $n + 1$, $n + 2$ и $n + 3$ являются самыми нижними на стержнях 0, 1 и 2 соответственно. Повторно выполнять единственное допустимое перемещение с соблюдением перечисленных ниже ограничений до тех пор, пока не останется возможных перемещений.

- Не помещать нечетный диск непосредственно поверх другого нечетного диска.
- Не помещать четный диск непосредственно поверх другого четного диска.
- Не отменять предыдущее перемещение.

(d) Пусть $\rho(n)$ обозначает наименьшее целое число k , такое что $n/2^k$ не является целым числом. Например, $\rho(42) = 2$, так как $42/2^1$ целое число, а $42/2^2$ не является целым числом. (То есть $\rho(n)$ на единицу больше, чем позиция наименьшей значимой единицы в двоичном представлении числа n .) Поскольку такое поведение напоминает метки на измерительной линейке, $\rho(n)$ иногда называют функцией линейки (ruler function; в русскоязычных источниках ее обычно называют функцией Римана (ТФДП)).

RulerHanoi(n):

```

i ← 1
while ρ(i) ≤ n
  if n - i is even
    move disk ρ(i) forward      ((0 → 1 → 2 → 0))
  else
    move disk ρ(i) backward    ((0 → 2 → 1 → 0))
  i ← i + 1

```

2. Головоломка «Ханойская башня» является относительно недавним потомком гораздо более древней механической головоломки, известной как «Китайские кольца» (Chinese linked rings), Baguenaudier (фр. название), Кольца Кардано (Cardan's Rings), Меледа (Meleda), Терпение (Patience), Оковы Тиринга (Tiring Irons), Замок узника (Prisoner's Lock), Spin-Out и под многими другими именами. Эта головоломка уже в XVI в. была хорошо известна и в Китае, и в Европе. Итальянский математик Лука Пачоли (Luca Pacioli) описал головоломку с 7 кольцами и ее решение в своем неопубликованном трактате «De Viribus Quantitatis», написанном между 1498 и 1506 гг.¹⁵ Всего лишь несколько лет спустя китайский поэт и писатель минской эпохи Ян Шэнь (Yang Shen) описал головоломку с 9 кольцами как «забаву

¹⁵ «De Viribus Quantitatis» («О мощи чисел») – одна из первых заметных работ по занимательной математике и, возможно, самый старый сохранившийся трактат по магии. Лука Пачоли более известен как автор «Summa de Arithmetica», почти полной энциклопедии по математике конца XV в., в которую включено первое описание бухгалтерского учета методом двойной записи (итальянская бухгалтерия).

для женщин и детей». По легенде, эту головоломку изобрел некий китайский полководец во II в. для своей жены, чтобы занять ее в то время, когда он участвовал в военных действиях.

Головоломка Baguenaudier имеет множество физических форм, но одна из наиболее часто встречающихся состоит из длинной металлической петли и нескольких колец, соединенных на твердом основании перемещаемыми стержнями. Изначально петля продета сквозь кольца, как показано на рис. 1.18. Цель головоломки – снять все кольца с петли.

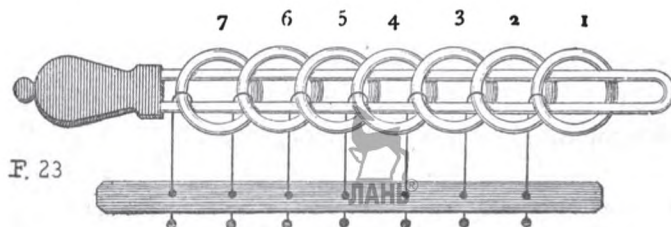


Рис. 1.18. Головоломка Baguenaudier с 7 кольцами, иллюстрация из книги Эдуарда Люка «*Récréations Mathématiques*» («Математические развлечения») (1891 г.). (Источник – интернет-архив <https://archive.org/details/p1rcrationsm00lucaoft/page/162>)

В более абстрактной форме можно смоделировать эту головоломку как последовательность битов, по биту для каждого кольца, где i -й бит равен 1, если петля проходит сквозь i -е кольцо, иначе бит равен 0. (Здесь мы отсчитываем индекс колец справа налево, как показано на рис. 1.18.) При решении головоломки разрешены два возможных (допустимых) хода:

- всегда можно изменить значение на противоположное для 1-го (= крайнего правого) бита;
- если строка битов заканчивается в точности z нулями, то можно изменить на противоположное значение $(z + 2)$ -го бита.

Цель этой формы головоломки – преобразовать строку из n единиц (битов) в строку из n нулей (битов). Например, показанная ниже последовательность из 21 хода решает головоломку с 5 кольцами:

$$\begin{aligned}
 &11111 \xrightarrow{1} 11110 \xrightarrow{3} 11010 \xrightarrow{1} 11011 \xrightarrow{2} 11001 \xrightarrow{1} 11000 \\
 &\xrightarrow{5} 01000 \xrightarrow{1} 01001 \xrightarrow{2} 01011 \xrightarrow{1} 01010 \xrightarrow{3} 01110 \\
 &\xrightarrow{1} 01111 \xrightarrow{2} 01101 \xrightarrow{1} 01100 \xrightarrow{4} 00100 \xrightarrow{1} 00101 \\
 &\xrightarrow{2} 00111 \xrightarrow{1} 00110 \xrightarrow{3} 00010 \xrightarrow{1} 00011 \xrightarrow{2} 00001 \xrightarrow{1} 00000
 \end{aligned}$$

- ♦(а) Назовем последовательность ходов (перемещений) сокращенной (reduced), если нет хода, противоположного предыдущему ходу. Доказать, что для любого неотрицательного целого числа n

существует ровно одна сокращенная последовательность ходов, решающая головоломку Baguenaudier с n кольцами. [Подсказка: эта задача решается намного проще, если вы уже знакомы с графами.]

- (b) Описать алгоритм решения головоломки Baguenaudier. Входные данные: число колец n . Алгоритм должен выводить сведенную последовательность ходов, решающую эту головоломку. Например, в качестве входных данных задано число 5, тогда алгоритм должен вывести последовательность ходов: 1, 3, 1, 2, 1, 5, 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1.
 - (c) Сколько в точности ходов выполняет предложенный вами алгоритм? Представить число ходов как функцию от n . Доказать, что ваш ответ верен.
3. В менее известной главе из истории головоломки «Ханойская башня» действие переносится из храма Бенареса в Пизу начала XIII в.¹⁴ Это перемещение совершил состоятельный купец-математик Леонардо Фибоначчи (Leonardo Fibonacci) по приказу императора Священной Римской империи Фридриха II, который слышал рассказы о храме Бенареса от воинов, возвращавшихся из Крестовых походов. Башни в Пизе и служащие в них монахи стали знаменитыми, что помогло Пизе утвердиться в роли господствующего центра торговли на Апеннинском полуострове.

К сожалению, почти сразу же после «перемещения» храма одна из алмазных игл стала наклоняться. Чтобы избежать опасности падения наклоняющейся башни из-за слишком большого веса дисков, Фибоначчи убедил монахов следовать более свободному правилу: *за один ход с наклоняющейся иглы можно перемещать вместе любое количество дисков на другую иглу*. По-прежнему запрещалось помещать диск большего размера поверх диска меньшего размера, и диски должны были перемещаться только по одному на «падающую» иглу или между двумя вертикальными иглами.

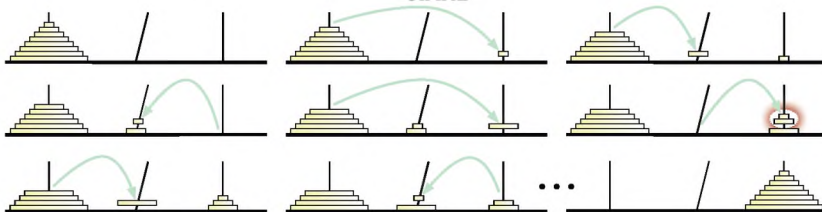


Рис. 1.19. Башни в Пизе. На пятом ходу два диска снимаются с «падающей» иглы (башни)

Благодаря новому правилу Фибоначчи священнослужители могли бы устроить конец света несколько быстрее в Пизе, чем это случи-

¹⁴ Кое-что в этой истории происходило на самом деле.

лось бы в Бенаресе. К счастью, храм был перенесен из Пизы обратно в Бенарес после того, как возведенный на престол новый Папа Римский Григорий IX отлучил от церкви Фридриха II, и местные священнослужители стали менее терпимо относиться к поддержке иностранных еретиков с чуждым математическим складом ума. Вскоре на месте перенесенного храма была воздвигнута колокольня, но она тоже сразу же начала крениться («падать»).

Описать алгоритм для передвижения стопки из n дисков с одной вертикальной иглы на другую вертикальную иглу с использованием минимально возможного числа ходов (перемещений). Сколько в точности ходов (перемещений) выполняет ваш алгоритм?

4. Рассмотрим еще несколько сокращенных вариантов головоломки «Ханойская башня». В каждом варианте задачи стержни пронумерованы 0, 1 и 2, а ваша задача – переместить стопку из n дисков со стержня 0 на стержень 2, как и в упражнении 1.

(а) Предположим, что вам запрещено перемещать любой диск непосредственно между стержнями 1 и 2, т. е. в каждом перемещении обязательно должен участвовать стержень 0. Описать алгоритм решения этой версии головоломки с минимально возможным числом перемещений. Сколько в точности ходов (перемещений) выполняет ваш алгоритм?

- ♣♥(b) Предположим, что вам разрешено перемещать диски только со стержня 0 на стержень 2, со стержня 2 на стержень 1 или со стержня 1 на стержень 0. Кроме того, предположим, что стержни расположены по окружности и пронумерованы по часовой стрелке, и вам разрешено перемещать диски только против часовой стрелки. Описать алгоритм решения этой версии головоломки с минимально возможным числом перемещений. Сколько ходов (перемещений) выполняет ваш алгоритм?

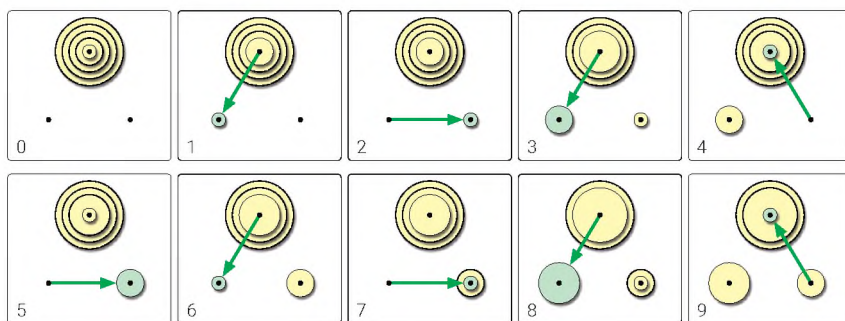


Рис. 1.20. Несколько первых ходов решения в версии головоломки «Ханойская башня» с перемещением дисков против часовой стрелки

♣♥(с) Наконец, предположим, что существует единственное ограничение: не разрешается перемещать диск напрямую со стержня 0 на стержень 2. Описать алгоритм решения этой версии головоломки с минимально возможным числом перемещений. Сколько ходов (перемещений) выполняет ваш алгоритм? [Подсказка: матрицы! Этот вариант существенно сложнее для анализа, чем предыдущие два.]

5. Рассмотрим еще более сложный вариант головоломки «Ханойская башня». Имеется ряд из k стержней, пронумерованных от 1 до k . За один ход разрешается перемещать наименьший диск со стержня i либо на стержень $i - 1$, либо на стержень $i + 1$ для любого индекса i . Как обычно, запрещается помещать диск большего размера на диск меньшего размера. *Цель:* переместить стопку из n дисков со стержня 1 на стержень k .

- (a) Описать рекурсивный алгоритм для случая $k = 3$. Сколько в точности ходов (перемещений) выполняет ваш алгоритм? (Это та же самая задача, что и в упражнении 4(a).)
- (b) Описать рекурсивный алгоритм для случая $k = n + 1$, который требует не более $O(n^5)$ перемещений. [Подсказка: использовать упражнение (a).]
- ♥(с) Описать рекурсивный алгоритм для случая $k = n + 1$, который требует не более $O(n^2)$ перемещений. [Подсказка: не использовать упражнение (a).]
- ♥(d) Описать рекурсивный алгоритм для случая $k = \sqrt{n}$, который требует алгебраического (полиномиального) числа перемещений. (Какого именно алгебраического??)
- ♥(e) Описать и проанализировать рекурсивный алгоритм для произвольных значений n и k . Насколько малым должно быть значение k (как функция от n), чтобы число перемещений было ограничено многочленом от n ?

Рекурсивные деревья

6. Использовать рекурсивные деревья для решения всех приведенных ниже рекурсивных выражений.

$$\begin{array}{lll}
 A(n) = 2A(n/4) + \sqrt{n} & B(n) = 2B(n/4) + n & C(n) = 2C(n/4) + n^2 \\
 D(n) = 3D(n/3) + \sqrt{n} & E(n) = 3E(n/3) + n & F(n) = 3F(n/3) + n^2 \\
 G(n) = 4G(n/2) + \sqrt{n} & H(n) = 4H(n/2) + n & I(n) = 4I(n/2) + n^2
 \end{array}$$

7. Использовать рекурсивные деревья для решения всех приведенных ниже рекурсивных выражений.

- (j) $J(n) = J(n/2) + J(n/3) + J(n/6) + n$
 (k) $K(n) = K(n/2) + 2K(n/3) + 3K(n/4) + n^2$
 (l) $L(n) = L(n/15) + L(n/10) + 2L(n/6) + \sqrt{n}$

♥8. Использовать рекурсивные деревья для решения всех приведенных ниже рекурсивных выражений.

- (m) $M(n) = 2M(n/2) + O(n \log n)$
 (n) $N(n) = 2N(n/2) + O(n / \log n)$
 (p) $P(n) = \sqrt{n} P(\sqrt{n}) + n$
 (q) $Q(n) = \sqrt{2n} Q(\sqrt{2n}) + \sqrt{n}$

Сортировка

9. Предположим, что имеется стопка из n блинов различных размеров. Необходимо отсортировать блины так, чтобы блин меньшего размера лежал поверх всех блинов большего размера. Единственная разрешенная операция – *переворот* (flip) – лопатка вставляется под k верхних блинов для некоторого целого числа k между 1 и n , и вся стопка из k блинов переворачивается, как показано на рис. 1.21.

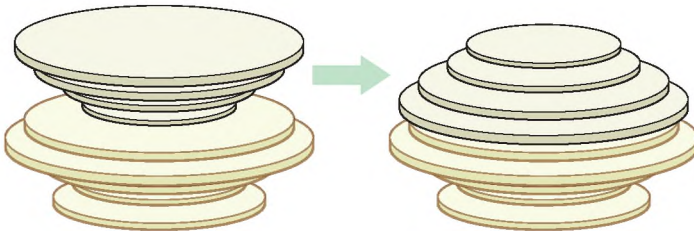


Рис. 1.21. Переворот четырех верхних блинов

- (a) Описать алгоритм сортировки произвольной стопки из n блинов с использованием $O(n)$ переворотов. Сколько в точности переворотов выполняет ваш алгоритм в наихудшем случае?¹⁵ [Подсказка: эта задача не имеет ничего общего с «Ханойской башней».]
- (b) Для каждого положительного целого числа n опишите стопку из n блинов, для сортировки которой требуется $\Omega(n)$ переворотов.
- (c) Теперь предположим, что одна сторона каждого блина пригорела. Описать алгоритм сортировки произвольной стопки из n блинов, чтобы каждый блин лежал пригоревшей стороной вниз, используя для этого $O(n)$ переворотов. Сколько в точности переворотов выполняет ваш алгоритм в наихудшем случае?

¹⁵ Точное определение наихудшего случая с оптимальным количеством переворотов, требуемым для сортировки n блинов (пригоревших или непригоревших), – задача, не решенная в течение длительного времени. Попробуйте получить самый лучший результат, какой только сможете.

10. Напомню, что эвристический метод «средний из трех» (median-of-three) рассматривает первый, последний и средний элементы массива и использует среднее значение этих трех элементов как опорный элемент в алгоритме быстрой сортировки (quicksort). Доказать, что алгоритм быстрой сортировки с применением эвристического метода «средний из трех» требует времени $\Omega(n^2)$ для сортировки массива размером n в наихудшем случае. В частности, для любого целого числа n описать операцию перестановки целых чисел от 1 до n , такую что при каждом рекурсивном вызове быстрой сортировки с определением среднего из трех опорным элементом всегда является второй наименьший элемент массива. Проектирование этой операции перестановки требует глубокого знания подпрограммы Partition.
- (a) В качестве подготовительного упражнения предположим, что подпрограмма Partition стабильна, т. е. сохраняет существующий порядок всех элементов, которые меньше и больше опорного элемента.
- ♥(b) Предположим, что подпрограмма Partition использует специализированный алгоритм, показанный на рис. 1.8 (стр. 55), который не является стабильным.
11. (a) Эй, Мо! Эй, Ларри! Докажите, что приведенный ниже алгоритм действительно сортирует свои входные данные.

```

StoogeSort(A[0..n - 1]):
  if n = 2 and A[0] > A[1]
    swap A[0] ↔ A[1]
  else if n > 2
    m = ⌊2n/3⌋
    StoogeSort(A[0..m - 1])
    StoogeSort(A[n - m..n - 1])
    StoogeSort(A[0..m - 1])

```

- (b) Сохраняет ли алгоритм StoogeSort корректность сортировки, если заменить $m = \lfloor 2n/3 \rfloor$ на $m = \lfloor 2n/3 \rfloor + 1$? Обоснуйте свой ответ.
- (c) Сформулировать рекуррентное выражение (включая самый простой случай (или случаи)) для числа сравнений, выполняемых алгоритмом StoogeSort.
- (d) Решить сформулированное рекуррентное выражение и доказать, что полученное решение верно. [Подсказка: не учитывать (отбросить) верхнюю границу.]
- (e) Доказать, что число перестановок (swaps), выполняемых алгоритмом StoogeSort, не превышает $\binom{n}{2}$.

12. Приведенный ниже brutalный и необычный алгоритм сортировки был предложен Гари Миллером (Gary Miller):

```

Cruel(A[1..n]):
  if n > 1
    Cruel(A[1..n/2])
    Cruel(A[n/2 + 1..n])
    Unusual(A[1..n])

```

```

Unusual(A[1..n]):
  if n = 2
    if A[1] > A[2]                <<Единственная операция сравнения!>>
      swap A[1] ↔ A[2]
  else
    for i ← 1 to n/4              <<Перестановка 2-й и 3-й четвертей>>
      swap A[i + n/4] ↔ A[i + n/2]
    Unusual(A[1..n/2])            <<Рекурсия в левой половине>>
    Unusual(A[n/2 + 1..n])        <<Рекурсия в правой половине>>
    Unusual(A[n/4 + 1..3n/4])     <<Рекурсия в средней половине>>

```

Сравнения, выполняемые этим алгоритмом, вообще не зависят от значений во входном массиве, и такой алгоритм сортировки называется «рассеянным» (oblivious). Для этой задачи предположим, что размер входных данных n всегда является степенью 2.

- Доказать методом индукции, что алгоритм Cruel правильно сортирует любой входной массив. [Совет: рассмотреть массив, содержащий $n/4$ 1s, $n/4$ 2s, $n/4$ 3s и $n/4$ 4s. Почему достаточно рассмотреть этот особый случай?]
 - Доказать, что алгоритм Cruel не должен правильно выполнять сортировку, если удалить цикл for из алгоритма Unusual.
 - Доказать, что алгоритм Cruel не должен правильно выполнять сортировку, если поменять местами две последние строки в алгоритме Unusual.
 - Определить время выполнения алгоритма Unusual. Обосновать свой ответ.
 - Определить время выполнения алгоритма Cruel. Обосновать свой ответ.
13. Перестановка (взаимная, или парная, перестановка, инверсия – inversion) в массиве $A[1..n]$ – это пара индексов, такая что $i < j$ и $A[i] > A[j]$. Число парных перестановок в массиве из n элементов является числом между 0 (если массив отсортирован) и $\binom{n}{2}$ (если массив отсортирован в обратном порядке). Описать и проанализировать ал-

горитм подсчета количества парных перестановок в любом массиве из n элементов за время $O(n \log n)$. [Подсказка: измените алгоритм сортировки слиянием.]

14. (а) Предположим, что имеются два множества из n точек: первое множество $\{p_1, p_2, \dots, p_n\}$ на прямой $y = 0$, второе множество $\{q_1, q_2, \dots, q_n\}$ на прямой $y = 1$. Создать множество из n отрезков прямой, соединяя каждую точку p_i с соответствующей точкой q_i . Описать и проанализировать алгоритм «разделяй и властвуй» для определения количества пересекающихся пар этих отрезков за время $O(n \log n)$. [Совет: см. предыдущую задачу.]
- (б) Теперь предположим, что два множества из n точек $\{p_1, p_2, \dots, p_n\}$ и $\{q_1, q_2, \dots, q_n\}$ заданы на единичной окружности. Соединить каждую точку p_i с соответствующей точкой q_i . Описать и проанализировать алгоритм «разделяй и властвуй» для определения количества пересекающихся пар этих отрезков за время $O(n \log^2 n)$. [Совет: использовать решение задачи (а).]
- ♥(с) Описать алгоритм решения задачи (б), который выполняется за время $O(n \log n)$. [Совет: использовать решение задачи (б).]

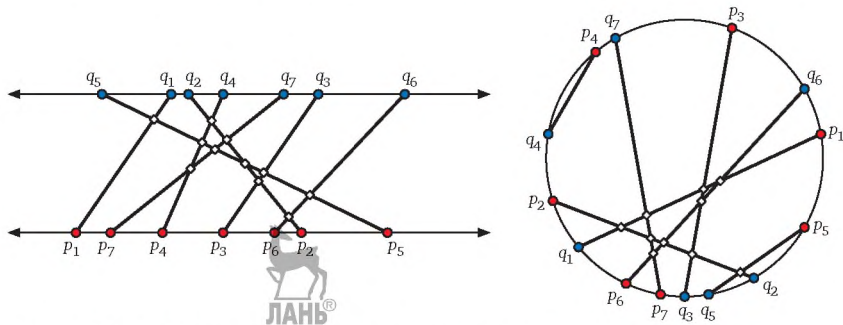


Рис. 1.22. Одиннадцать пересекающихся пар отрезков с конечными точками на параллельных прямых и десять пересекающихся пар отрезков с конечными точками на окружности

15. (а) Описать алгоритм, который сортирует входной массив $A[1..n]$, вызывая подпрограмму $\text{SqrtSort}(k)$, сортирующую подмассив $A[k+1..k+\sqrt{n}]$ на месте и принимающую в качестве входного параметра произвольное целое число k , значение которого находится между 0 и $n - \sqrt{n}$. (Для упрощения задачи принять, что $\sqrt{n}$ – целое число.) Вашему алгоритму разрешается просматривать или изменять входной массив только через вызов подпрограммы SqrtSort , т. е. ваш алгоритм не должен напрямую сравнивать, перемещать или копировать элементы массива. Сколько вызовов SqrtSort выполнит ваш алгоритм в наихудшем случае?
- ♣(б) Доказать, что алгоритм, разработанный в упражнении (а), является оптимальным с точностью до постоянных коэффициентов. Дру-

гими словами, если $f(n)$ – число вызовов подпрограммы `SqrtSort` вашим алгоритмом, то необходимо доказать, что не существует алгоритма, способного выполнить сортировку, используя $o(f(n))$ вызовов `SqrtSort`.

- (с) Теперь предположим, что подпрограмма `SqrtSort` реализована рекурсивно с применением вызова алгоритма сортировки, разработанного в упражнении (а). Например, на втором уровне рекурсии этот алгоритм сортирует массивы приблизительного размера $n^{1/4}$. Каково время выполнения в наихудшем случае для итогового алгоритма сортировки? (Для упрощения анализа будем считать, что массив размером n имеет форму 2^{2^k} , так что повторяющиеся вычисления квадратного корня всегда дают в результате целые числа.)

Выбор

16. Предположим, что задано множество S из n элементов, причем каждый элемент имеет *значение* (value) и *вес* (weight). Для каждого элемента $x \in S$ определены два подмножества:

- $S_{<x}$ – множество элементов S , значения которых меньше значения x ;
- $S_{>x}$ – множество элементов S , значения которых больше значения x .

Для любого подмножества $R \subseteq S$ пусть $w(R)$ обозначает сумму весов элементов из подмножества R . Взвешенное среднее значение (weighted median) подмножества R – любой элемент x , такой что $w(S_{<x}) \leq w(S)/2$ и $w(S_{>x}) \leq w(S)/2$.

Описать и проанализировать алгоритм вычисления взвешенного среднего значения для заданного взвешенного множества за время $O(n)$. Входные данные состоят из двух неотсортированных массивов $S[1..n]$ и $W[1..n]$, в которых для каждого индекса i соответствующий i -й элемент содержит значение $S[i]$ и вес $W[i]$. Можно допустить, что все значения различны, а все веса положительны.

17. (а) Описать алгоритм определения за время $O(n)$, содержит ли произвольный массив $A[1..n]$ более $n/4$ копий любого значения.
- (б) Описать и проанализировать алгоритм, который для произвольного массива $A[1..n]$ и целого числа k определяет, содержит ли массив A более k копий любого значения. Время выполнения вашего алгоритма необходимо сформулировать как функцию от n и k .

Запрещается использовать хеширование, или алгоритм поразрядной (цифровой) сортировки, или любой другой метод, который зависит от точности входных значений, а не от их порядка.

18. Описать алгоритм вычисления среднего значения в массиве $A[1..5]$ несовпадающих чисел с использованием не более шести операций сравнения. Вместо написания псевдокода опишите алгоритм с помощью дерева решений (decision tree): двоичного (бинарного) дерева, в котором каждый внутренний узел содержит операцию сравнения в форме « $A[i] \geq A[j]$?», а каждый лист содержит индекс в массиве.

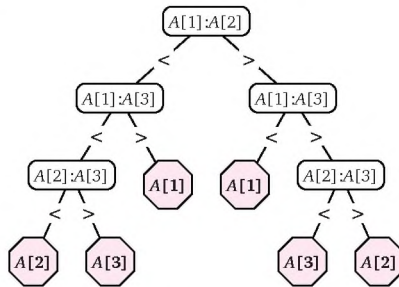


Рис. 1.23. Определение среднего значения в массиве из трех элементов с использованием не более трех сравнений

19. Рассмотреть обобщение алгоритма Блума–Флойда–Пратта–Ривеста–Тарьяна (Blum–Floyd–Pratt–Rivest–Tarjan) MomSelect , показанного на рис. 1.24. Алгоритм разделяет входной массив на $\lfloor n/b \rfloor$ блоков размером b вместо $\lfloor n/5 \rfloor$ блоков размером 5, но в остальном алгоритм тот же.

```

MombSelect( $A[1 \dots n]$ ,  $k$ ):
  if  $n \leq b^2$ 
    use brute force
  else
     $m \leftarrow \lfloor n/b \rfloor$ 
    for  $i \leftarrow 1$  to  $m$ 
       $M[i] \leftarrow \text{MedianOfB}(A[b(i-1)+1..bi])$ 
     $\text{mom}_b \leftarrow \text{Mom}_b\text{Select}(M[1..m], \lfloor m/2 \rfloor)$ 
     $r \leftarrow \text{Partition}(A[1..n], \text{mom}_b)$ 
    if  $k < r$ 
      return  $\text{Mom}_b\text{Select}(A[1..r-1], k)$ 
    else if  $k > r$ 
      return  $\text{Mom}_b\text{Select}(A[r+1..n], k-r)$ 
    else
      return  $\text{mom}_b$ 
  
```

Рис. 1.24. Параметризованное семейство алгоритмов выбора (см. задачу 19)

- (а) Сформулировать рекуррентное выражение для времени выполнения алгоритма $\text{Mom}_b\text{Select}$, полагая, что b – постоянная величина (поэтому подпрограмма MedianOfB выполняется за время $O(1)$). Также определить, как размеры рекурсивных подзадач зависят

от постоянной b . Отдельно рассмотреть случаи четной и нечетной постоянной b .

- (b) Каково время выполнения в наихудшем случае для алгоритма $\text{Mom}_1\text{Select}$? [Подсказка: это вопрос с подвохом.]
- ♣♥(c) Каково время выполнения в наихудшем случае для алгоритма $\text{Mom}_2\text{Select}$? [Подсказка: это нечестный вопрос.]
- ♥(d) Каково время выполнения в наихудшем случае для алгоритма $\text{Mom}_3\text{Select}$? Поиск верхней границы времени выполнения вполне понятен, самая трудная часть – показать, что этот анализ действительно является строгим. [Подсказка: см. задачу 10.]
- ♥(e) Каково время выполнения в наихудшем случае для алгоритма $\text{Mom}_4\text{Select}$? И в этом случае самой трудной частью становится подтверждение того, что анализ невозможно улучшить.¹⁶
- (f) Для любых постоянных $b \geq 5$ алгоритм $\text{Mom}_b\text{Select}$ выполняется за время $O(n)$, но при различных значениях b появляются разнообразные постоянные коэффициенты. Пусть $M(b)$ обозначает минимальное количество сравнений, требуемых для поиска среднего значения b чисел. Точное значение $M(b)$ известно только для $b \leq 13$ (см. табл. 1.1).

Таблица 1.1

b	1	2	3	4	5	6	7	8	9	10	11	12	13
$M(b)$	0	1	3	4	6	8	10	12	14	16	18	20	23

Для каждого b между 5 и 13 найти верхнюю границу времени выполнения алгоритма $\text{Mom}_b\text{Select}$ в форме $T(n) \leq \alpha_b n$ при некоторой явно определенной постоянной α_b . (Например, на стр. 66 в подразделе «Проверка достоверности» было показано, что $\alpha_5 \leq 16$.)

- (g) При каком значении b получается наименьшее постоянное значение α_b ? [Подсказка: это вопрос с подвохом.]

20. Доказать, что вариант алгоритма Блума–Флойда–Пратта–Ривеста–Тарьяна (Blum–Floyd–Pratt–Rivest–Tarjan) Select, показанного на рис. 1.25 и использующего дополнительный уровень малых (промежуточных) средних значений для выбора основного опорного элемента, выполняется за время $O(n)$.

21. (a) Предположим, что имеются два отсортированных массива $A[1..n]$ и $B[1..n]$. Описать алгоритм поиска среднего элемента в объединении

¹⁶ Средним значением четырех элементов является либо второй наименьший элемент, либо второй наибольший элемент. В 2014 г. Ке Чень (Ke Chen) и Адриан Думитреску (Adrian Dumitrescu) доказали, что если усовершенствовать алгоритм $\text{Mom}_4\text{Select}$ для поиска вторых наименьших элементов при $k < n/2$ и вторых наибольших элементов при $k > n/2$, то итоговый алгоритм выполняется за время $O(n)$. Более подробно см. статью этих авторов «Select with Groups of 3 or 4 Takes Linear Time» («Выбор в группах из трех или четырех элементов требует линейного времени») (WADS 2015, arXiv:1409.3600).

массивов A и B за время $\Theta(\log n)$. Можно считать, что массивы не содержат повторяющихся элементов.

- (b) Предположим, что имеются два отсортированных массива $A[1..m]$ и $B[1..n]$ и целое число k . Описать алгоритм поиска k -го наименьшего элемента в $A \cup B$ за время $\Theta(\log(m+n))$. Например, если $k = 1$, то алгоритм должен возвращать наименьший элемент объединения $A \cup B$. [Подсказка: использовать решение упражнения (a).]

```

MomomSelect(A[1..n], k):
  if n ≤ 81
    use brute force
  else
    m ← ⌊n/3⌋
    for i ← 1 to m
      M[i] ← MedianOf3(A[3i - 2..3i])
    mm ← ⌊m/3⌋
    for j ← 1 to mm
      Mom[j] ← MedianOf3(M[3j - 2..3j])
    momom ← MomomSelect(Mom[1..mm], ⌊mm/2⌋)
    r ← Partition(A[1..n], momom)
    if k < r
      return MomomSelect(A[1..r - 1], k)
    else if k > r
      return MomomSelect(A[r + 1..n], k - r)
    else
      return momom

```

Рис. 1.25. Выбор по среднему значению для средних значений из средних значений (median of moms) (см. задачу 20)

- ♥(c) Теперь предположим, что имеются три отсортированных массива $A[1..n]$, $B[1..n]$ и $C[1..n]$, а также целое число k . Описать алгоритм поиска k -го наименьшего элемента в объединении $A \cup B \cup C$ за время $O(\log n)$.
- (d) Наконец, предположим, что имеется двумерный массив $A[1..m, 1..n]$, в котором каждая строка $A[i, \cdot]$ отсортирована, и целое число k . Описать алгоритм поиска k -го наименьшего элемента в массиве A с наибольшей возможной скоростью. Как время выполнения вашего алгоритма зависит от m ? [Подсказка: сначала необходимо решить задачу 16.]

Арифметика

22. В 1854 г. археологи обнаружили шумерские глиняные таблички, датированные приблизительно 2000 г. до н. э., со списком квадратов целых чисел до 59 включительно. Благодаря этому открытию некоторые исследователи пришли к заключению о том, что древние

шумеры выполняли умножение методом сведения к возведению в квадрат, используя тождество $x \cdot y = (x^2 + y^2 - (x - y)^2)/2$. К сожалению, те же самые исследователи умалчивают о том, как предположительно шумеры возводили в квадрат большие числа. Четыре тысячи лет спустя мы можем окончательно освободить шумерских математиков от этого весьма тяжелого труда, используя мощь рекурсии.

- (a) Описать вариант алгоритма Карацубы, возводящий в квадрат любое n -значное число за время $O(n^{\lg 3})$ при помощи сведения к возведению в квадрат трех $\lceil n/2 \rceil$ -значных чисел. (Карацуба действительно сделал это в 1960 г.)
 - (b) Описать рекурсивный алгоритм, возводящий в квадрат любое n -значное число за время $O(n^{\lg_5 6})$ при помощи сведения к возведению в квадрат шести $\lceil n/3 \rceil$ -значных чисел.
 - ♥(c) Описать рекурсивный алгоритм, возводящий в квадрат любое n -значное число за время $O(n^{\lg_5 5})$ при помощи сведения к возведению в квадрат всего лишь пяти $(n/3 + O(1))$ -значных чисел. [Подсказка: чему равно выражение $(a + b + c)^2 + (a - b + c)^2$?]
23. (a) Описать и проанализировать вариант алгоритма Карацубы, умножающий любое m -значное число на любое n -значное число при любом $n \geq m$ за время $O(n^{\lg 3 - 1})$.
- (b) Описать алгоритм вычисления десятичного представления числа 2^n за время $O(n^{\lg 3})$, используя алгоритм из упражнения (a) как подпрограмму. (Стандартный алгоритм, поочередно вычисляющий одну цифру за один шаг, требует времени $\Theta(n^2)$.)
- (c) Описать алгоритм «разделяй и властвуй» для вычисления десятичного представления произвольного n -битового двоичного числа за время $O(n^{\lg 3})$. [Подсказка: обратите особое внимание на дополнительный логарифмический коэффициент в выражении времени выполнения.]
- ♥(d) Предположим, что существует возможность умножения двух n -значных чисел за время $O(M(n))$. Описать алгоритм вычисления десятичного представления произвольного n -битового числа за время $O(M(n) \log n)$. [Подсказка: самой сложной частью задачи является анализ; используйте метод преобразования области.]
24. Рассмотреть показанный ниже классический рекурсивный алгоритм вычисления факториала $n!$ для неотрицательного целого числа n .

```

Factorial(n):
  if n = 0
    return 1
  else
    return n · Factorial(n - 1)

```


- (a) Сколько операций умножения выполняет этот алгоритм?
- (b) Сколько битов требуется для записи $n!$ в двоичном представлении? Запишите ответ в форме $\Theta(f(n))$, используя некоторую известную функцию $f(n)$. [Подсказка: $(n/2)^{n/2} < n! < n^n$.]
- (c) Ответ в упражнении (b) должен убедить вас в том, что количество операций умножения не является правильной оценкой действительного времени выполнения алгоритма Factorial. Можно умножить любое k -значное число на любое l -значное число за время $O(k \cdot l)$, используя либо алгоритм решеточного умножения, либо метод удвоения и усреднения. Каким будет время выполнения алгоритма Factorial, если воспользоваться одним из этих алгоритмов умножения как подпрограммой?
- (d) Приведенный ниже рекурсивный алгоритм также вычисляет функцию факториала, но использует другой способ группировки операций умножения.

```

Falling(n,m):
  if m = 0
    return 1
  else if m = 1
    return n
  else
    return Falling(n, [m/2]) · Falling(n - [m/2], [m/2])
  
```

⟨⟨Вычисление $n!/(n-m)!$ ⟩⟩

Каким будет время выполнения алгоритма Falling(n, n), если использовать алгоритм умножения начальной школы? [Подсказка: как обычно, необходимо исключить нижние и верхние границы.]

- (e) Описать и проанализировать вариант алгоритма Карацубы, который умножает любое k -значное число на любое l -значное число при любом $k \geq l$ за время $O(k \cdot l^{\lg 3 - 1}) = O(k \cdot l^{0.585})$.
- ♥(f) Каким будет время выполнения алгоритмов Factorial(n) и Falling(n, n), если использовать измененный метод умножения Карацубы из упражнения (e)?
25. *Наибольший общий делитель* (greatest common divisor) двух положительных целых чисел x и y , обозначаемый как $\gcd(x, y)$, – это наибольшее целое число d , такое, что x/d и y/d – целые числа. Евклид в своем труде «Начала» («Elements»), написанном около 300 г. до н. э., описал следующий рекурсивный алгоритм вычисления $\gcd(x, y)$:¹⁷

¹⁷ Алгоритм Евклида иногда неправильно называют самым древним рекурсивным алгоритмом или даже самым древним нетривиальным алгоритмом, несмотря на то что египетский алгоритм удваивания и усреднения, который является нетривиальным и рекурсивным, опередил Евклида как минимум на 1500 лет.

```

EuclidGCD(x, y):
    if x = y
        return x
    else if x > y
        return EuclidGCD(x - y, y)
    else
        return EuclidGCD(x, y - x)

```

- (a) Доказать, что алгоритм EuclidGCD правильно вычисляет $gcd(x, y)$.¹⁸
А именно:
- i. доказать, что алгоритм EuclidGCD(x, y) выполняет деление обоих чисел x и y ;
 - ii. доказать, что каждый делитель x и y является также делителем EuclidGCD(x, y).
- (b) Каково время выполнения алгоритма EuclidGCD(x, y) в наихудшем случае как функции от x и y ? (Предположить, что вычисление $x - y$ требует времени $O(\log x + \log y)$.)
- (c) Доказать, что приведенный ниже алгоритм также вычисляет $gcd(x, y)$:

```

FastEuclidGCD(x, y):
    if y = 0
        return x
    else if x > y
        return FastEuclidGCD(y, x mod y)
    else
        return FastEuclidGCD(x, y mod x)

```

- (d) Каково время выполнения алгоритма FastEuclidGCD(x, y) в наихудшем случае как функции от x и y ? (Предположить, что вычисление $x \bmod y$ требует времени $O(\log x \cdot \log y)$.)
- (e) Доказать, что приведенный ниже алгоритм также вычисляет $gcd(x, y)$:



¹⁸ Евклид не доказал корректность своего алгоритма. Положение 1 в книге VII «Начал» утверждает, что если $EuclidGCD(x, y) = 1$, то x и y являются взаимно простыми числами (т. е. $gcd(x, y) = 1$), но доказательство рассматривает только особый случай $x \bmod (y \bmod (x \bmod y)) = 1$. Положение 2 утверждает, что если x и y не являются взаимно простыми числами, то $EuclidGCD(x, y) = gcd(x, y)$, но доказательство рассматривает только особые случаи $gcd(x, y) = y$ и $gcd(x, y) = y \bmod (x \bmod y)$. В итоге эти два положения не содержат полного доказательства корректности алгоритма EuclidGCD. Не следует уподобляться Евклиду.



```

BinaryGCD(x, y):
  if x = y
    return x
  else if x and y are both even
    return 2 · BinaryGCD(x/2, y/2)
  else if x is even
    return BinaryGCD(x/2, y)
  else if y is even
    return BinaryGCD(x, y/2)
  else if x > y
    return BinaryGCD((x - y)/2, y)
  else
    return BinaryGCD(x, (y - x)/2)

```

- (f) Каково время выполнения алгоритма $\text{BinaryGCD}(x, y)$ в наихудшем случае как функция от x и y ? (Предположить, что вычисление $x - y$ требует времени $O(\log x + \log y)$, а вычисление $z/2$ требует времени $O(\log z)$.)

Массивы

26. Предположим, что имеется шашечная доска размером $2^n \times 2^n$ с одной (произвольно выбранной) удаленной клеткой. Описать и проанализировать алгоритм вычисления раскладки на доске без пропусков и перекрытий плиток L-формы, причем каждая плитка составлена из трех клеток. Входные данные: целое число n и два n -битовых целых числа, представляющих строку и столбец (горизонталь и вертикаль) удаленной клетки. Требуемый вывод: список позиций и ориентаций $(4^n - 1)/3$ плиток. Алгоритм должен выполняться за время $O(4^n)$. [Подсказка: сначала необходимо доказать, что такая раскладка всегда существует.]
27. Вы присутствуете на объединенном съезде политических партий (или, возможно, на общем факультетском собрании) с участием n представителей (делегатов). Каждый делегат является членом одной и только одной политической партии. Невозможно определить, к какой политической партии принадлежит тот или иной делегат, и вас сразу же исключат из участников съезда, если вы спросите. Но вы можете определить, принадлежит ли любая пара делегатов к одной и той же партии, представив их друг другу. Члены одной политической партии всегда приветствуют друг друга улыбками и дружескими рукопожатиями, а члены разных партий всегда обмениваются угрожающими пристальными взглядами и оскорбительными выпадами.¹⁹

¹⁹ Политика в реальном мире намного сложнее, чем эта упрощенная модель, но это же теоретическая книга!

- (а) Дополнительно предположим, что половина делегатов принадлежит к одной политической партии. Описать эффективный алгоритм, идентифицирующий всех членов этой партии большинства.
- (б) Теперь предположим, что на съезде присутствует более двух партий, но одна партия обладает большинством: к этой партии принадлежит больше делегатов, чем к любой другой партии. Представить практическую процедуру точного выбора членов этой партии большинства настолько экономно, насколько это возможно, подразумевая, что в партии большинства насчитывается не менее p членов. Очень прошу сделать это.

28. На Острове Смаллиана (Smullyan Island) три типа жителей: рыцари (knights) всегда говорят правду, мошенники (knives) всегда лгут, а обычные люди (normals) иногда говорят правду, иногда нет. Всем на острове известно имя и тип (рыцарь, мошенник, обычный человек) каждого жителя. Необходимо определить тип каждого жителя.

Можно попросить любого жителя сообщить вам тип любого другого обитателя острова. Точнее говоря, если вы спросите: «Эй, X , к какому типу принадлежит Y ?», то X даст один из перечисленных ниже ответов:

- если X – рыцарь, то X сообщит правильный тип жителя Y ;
- если X – мошенник, то X может сообщить один из типов, к которому не принадлежит житель Y ;
- если X – обычный человек, то X может сообщить любой из трех типов.

Жители острова будут игнорировать все вопросы, заданные не в указанной выше точной форме, в частности, вам запрещено спрашивать жителя о его собственном типе. Один и тот же вопрос, многократно заданный одному и тому же жителю, всегда дает один и тот же ответ, поэтому нет никакого смысла повторять любой вопрос больше одного раза.

- (а) Предположим, вам известно, что строгим большинством жителей являются рыцари. Описать эффективный алгоритм идентификации типа каждого жителя.
- (б) Доказать, что если не более половины жителей составляют рыцари, то невозможно определить тип каждого жителя.

29. Большая часть графических аппаратных средств включает поддержку низкоуровневой операции *blit*, или **block transfer** (передача блоков), которая быстро копирует прямоугольный фрагмент карты пикселей (двумерный массив значений пикселей) из одной локации (памяти) в другую. Это двумерная версия функции стандартной библиотеки языка C `memcpy()`.

Предположим, что необходимо повернуть карту пикселей размером $n \times n$ на 90° по часовой стрелке. Один из способов, по крайней мере, если n представляет собой степень двойки: разделение карты пикселей на четыре блока размером $n/2 \times n/2$, перемещение каждого блока в требуемую позицию с использованием последовательности из пяти бит, а затем каждый блок рекурсивно поворачивается. (Почему пять бит? Да по той же причине, по которой для головоломки «Ханойская башня» требуется третий стержень.) Другой вариант: можно сначала рекурсивно повернуть каждый блок, а затем передать (blit) их в требуемую локацию.



Рис. 1.26. Два алгоритма для поворота карты пикселей

- Доказать, что обе версии алгоритма корректны, если n является степенью 2.
- Сколько в точности операций передачи блоков (blit) выполняет алгоритм, если n является степенью 2?
- Описать, как нужно изменить алгоритм, чтобы он работал для любого произвольного n , а не только для степеней 2. Сколько операций передачи блоков (blit) выполняет этот измененный алгоритм?
- Каким будет время выполнения этого алгоритма, если передача блока (blit) размером $k \times k$ требует времени $O(k^2)$?
- А что, если для передачи блока (blit) размером $k \times k$ требуется всего лишь время $O(k)$?

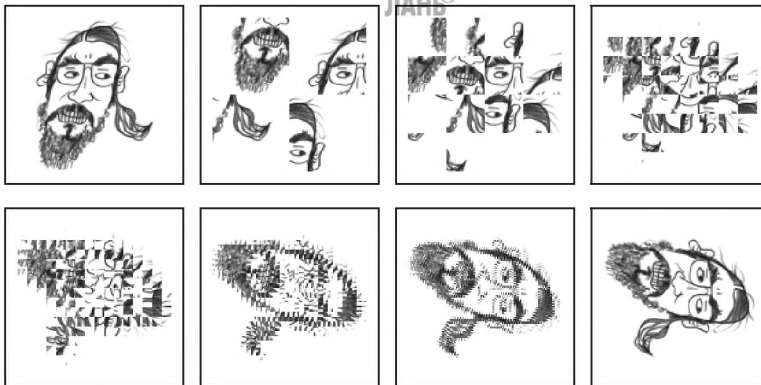


Рис. 1.27. Последовательные этапы работы первого алгоритма поворота (blit, затем рекурсия). (Копия портрета автора книги, выполненного цветной пастелью Тиней Эриксон (Tina Erickson) в 2000 г. Копия публикуется с разрешения автора-художника.)

30. Массив $A[0..n-1]$ из n различных чисел является битоническим (bitonic), если существуют единственные в своем роде индексы i и j , такие что $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$ и $A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$. Другими словами, битоническая последовательность состоит либо из возрастающей последовательности, за которой следует убывающая последовательность, либо может быть циклически сдвинута для получения такой комбинации последовательностей. Например:



Рис. 1.28. Пример битонической и небитонической последовательности

Описать и проанализировать алгоритм поиска наименьшего элемента в битоническом массиве из n элементов за время $O(\log n)$. Можно допустить, что все числа во входном массиве различны.

31. Предположим, что имеется массив $A[1..n]$ из n различных целых чисел, которые могут быть положительными, отрицательными или нулем. Массив отсортирован в возрастающем порядке, т. е. $A[1] < A[2] < \dots < A[n]$.
- (а) Описать быстрый алгоритм, который либо вычисляет индекс i , такой что $A[i] = i$, либо при необходимости сообщает, что такого индекса не существует.
 - (б) Допустим, заранее известно, что $A[1] > 0$. Описать еще более быстрый алгоритм, который либо вычисляет индекс i , такой что $A[i] = i$, либо при необходимости сообщает, что такого индекса не существует. [Подсказка: это действительно легко сделать.]
32. Предположим, что имеется массив $A[1..n]$ с особым свойством: $A[1] \geq A[2]$ и $A[n-1] \leq A[n]$. Мы говорим, что элемент $A[x]$ является локальным минимумом, если он меньше или равен обоим своим соседям, или более формально, если $A[x-1] \geq A[x]$ и $A[x] \leq A[x+1]$. На рис. 1.29 показан пример с шестью локальными минимумами в конкретном массиве.



Рис. 1.29. Шесть локальных минимумов в массиве

Разумеется, можно найти локальный минимум за время $O(n)$ с помощью прямого прохода по массиву (сканируя массив). Описать и проанализировать алгоритм, который находит локальный минимум за время $O(\log n)$. [Подсказка: при заданных граничных условиях

массив обязательно должен иметь не менее одного локального минимума. Почему?]

33. Предположим, что имеется отсортированный массив из n различных чисел, в котором было выполнено k шагов (элементарных операций) вращения при некотором неизвестном целом числе k со значением между 1 и $n - 1$. Таким образом, задан массив $A[1..n]$, такой что некоторый его префикс $A[1..k]$ отсортирован в возрастающем порядке, соответствующий суффикс $A[k + 1..n]$ отсортирован в возрастающем порядке и $A[n] < A[1]$.

Например, может быть задан массив из 16 элементов (при $k = 10$), показанный на рис. 1.30.

9	13	16	18	19	23	28	31	37	42	1	3	4	5	7	8
---	----	----	----	----	----	----	----	----	----	---	---	---	---	---	---

Рис. 1.30. Массив из 16 элементов после $k = 10$ операций вращения

- (а) Описать и проанализировать алгоритм вычисления неизвестного целого числа k .
- (б) Описать и проанализировать алгоритм, который определяет, содержит ли рассматриваемый массив заданное число x .
34. В конце второй серии боевика-блокбастера «Быстрый и невероятный XIII^{3/4}: Последние стражи законности одноразового пользования – перезагрузка» злодей Доктор Метафор гипнотизирует всю Лигу/Отряд/Команду героев, выстраивает их в длинный ряд на краю пропасти и приказывает каждому герою выстрелить в ближайшего более высокого героя слева и справа по условному сигналу.

Предположим, что заданы значения роста всех n героев в порядке слева направо и размещены в массиве $Ht[1..n]$. (Чтобы исключить споры об оплате, продюсеры постановили, что рост всех героев должен быть различным (т. е. в любой произвольно выбранной паре рост героев не одинаков).) Поэтому можно вычислить цель слева и цель справа для каждого героя за время $O(n^2)$, используя показанный ниже алгоритм прямого перебора («грубой силы»).

```
WhoTargetsWhom(Ht[1..n]):
  for j ← 1 to n
    «Поиск цели слева L[j] для героя j»
    L[j] ← None
    for i ← 1 to j - 1
      if Ht[i] > Ht[j]
        L[j] ← i
```

```

    <<Поиск цели справа R[j] для героя j>>
    R[j] ← None
    for k ← n down to j + 1
        if Ht[k] > Ht[j]
            R[j] ← k
    return L[1..n], R[1..n]

```

- (a) Описать алгоритм «разделяй и властвуй», вычисляющий результат WhoTargetsWhom за время $O(n \log n)$.
- (b) Доказать, что не менее $\lfloor n/2 \rfloor$ из n героев являются целями. То есть доказать, что выходные массивы $R[0..n-1]$ и $L[0..n-1]$ содержат не менее $\lfloor n/2 \rfloor$ различных значений (отличающихся от None).
- (c) Увы, дьявольский план Доктора Метафора оказался успешным. По условному сигналу все герои одновременно стреляют в свои цели, и все пораженные цели падают в пропасть и, несомненно, погибают. Метафор повторяет свой подлый эксперимент снова и снова – после каждой бойни он принуждает оставшихся героев выбирать новые цели, следуя тому же алгоритму, а затем поражать выбранные цели по очередному сигналу. В итоге в живых остается только самый низкорослый член Команды/Союза/Отряда героев.²⁰

Описать и проанализировать алгоритм вычисления количества раундов до полного окончания смертоносного процесса Доктора Метафора. Для получения максимальной оценки ваш алгоритм должен выполняться за время $O(n)$.

35. Вы являетесь участником популярного игрового шоу «Побеждай своих соседей!» (Beat Your Neighbors!). Вам представлено игровое поле размером $m \times n$ с коробками, в каждой из которых скрыты неповторяющиеся числа. Открытие одной коробки стоит 100 долл. Ваша цель – найти коробку с числом, которое больше всех своих соседей на игровом поле (выше, ниже, слева и справа). Если вы потратите меньше денег, чем любой из ваших соперников по игре, то выиграете недельную поездку на двоих в Лас-Вегас и годовое снабжение пищевым продуктом Rice-A-Roni™, к которому вы безнадежно пристрадались.

- (a) Допустим, $m = 1$. Описать алгоритм поиска числа, которое больше любого из его соседей. Сколько коробок ваш алгоритм открывает в наихудшем случае?

²⁰ В захватывающей концовке Реткон-Белка (retcon – аббр. от retroactive continuity – художественный прием, меняющий «более ранний» сюжет), последний выживший член Команды/Группы/Сообщества героев спасает всех, переместившись назад во времени и предусмотрительно заменив других $n-1$ героев очень похожими на них фигурами из воздушных шаров. Ну да, по существу, это «Мстители: Финал» (Avengers: Endgame).

мок вершины v (если такой существует), правый потомок вершины v (если такой существует) и родитель (предок) вершины v (если такой существует). Мы называем v центральной вершиной (central vertex), если каждое из перечисленных выше поддеревьев меньшего размера содержит максимум $n/2$ вершин. Пример показан на рис. 1.32.

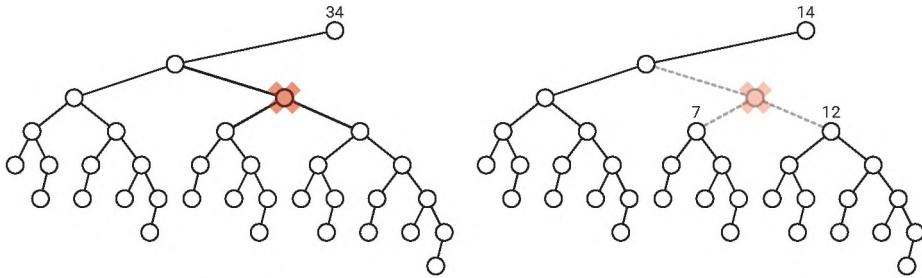


Рис. 1.32. После удаления центральной вершине в двоичном дереве с 34 узлами остаются поддеревья с 14, 7 и 12 узлами

Описать и проанализировать алгоритм поиска центральной вершины в заданном произвольном двоичном дереве. [Подсказка: сначала необходимо доказать, что в каждом дереве имеется центральная вершина.]

39. (a) Профессор Джордж О’Джангл (George O’Jungle) создал двоичное дерево с 27 узлами, в котором каждый узел помечен не повторяющейся буквой латинского алфавита или символом &. Обход дерева в *прямом* (preorder) и в *обратном* (postorder) порядке (обход с предварительной и с отложенной выборкой) обеспечивает посещение узлов в приведенных ниже последовательностях:

- в прямом порядке: I Q J H L E M V O T S B R G Y Z K C A & F P N U D W X;
- в обратном порядке: H E M L J V Q S G Y R Z B T C P U D N F W & X A K O I.

Изобразить графически двоичное дерево Джорджа.

- (b) Напомню, что двоичное дерево является полным (*full*), если каждый узел, не являющийся листом, имеет ровно двух потомков.
- i. Описать и проанализировать рекурсивный алгоритм восстановления любого произвольного полного двоичного дерева, если в качестве входных данных представлены последовательности обхода его узлов в прямом и обратном порядке.
 - ii. Доказать, что не существует алгоритма восстановления произвольного двоичного дерева по заданной последовательности обхода его узлов в прямом и обратном порядке.
- (c) Описать и проанализировать рекурсивный алгоритм восстановления произвольного двоичного дерева, если в качестве входных

данных представлена последовательность обхода его узлов в прямом порядке и упорядоченного обхода (обхода с порядковой выборкой – *inorder*).

- (d) Описать и проанализировать рекурсивный алгоритм восстановления произвольного двоичного дерева поиска (*binary search tree*), если в качестве входных данных представлена только последовательность обхода его узлов в прямом порядке.
- ♥(e) Описать и проанализировать рекурсивный алгоритм восстановления произвольного двоичного дерева поиска, если в качестве входных данных представлена только последовательность обхода его узлов в прямом порядке за время $O(n)$.

В упражнениях (b)–(e) принимается, что все ключи (значения узлов) различны и что входные данные логически согласованы по крайней мере с одним двоичным деревом.

40. Предположим, что существует n точек, произвольно размещенных внутри двумерной прямоугольной области. *kd-дерево* (*kd-tree*)²¹ рекурсивно подразделяет эти точки описанным ниже способом. Если внутри прямоугольной области нет точек, то задача решена. Иначе область делится вертикальной прямой на две прямоугольные области меньшего размера, при этом средняя точка внутри области (но не на ее границе) обеспечивает разделение точек настолько равномерно, насколько это возможно. Далее рекурсивно формируется *kd-дерево* для точек в каждой из двух меньших областей после поворота их на 90° . Таким образом, на каждом уровне рекурсии чередуется разделение по вертикали и по горизонтали. Окончательным результатом являются пустые прямоугольные области, называемые ячейками (*cells*).

- (a) Сколько получается ячеек – определить их количество как функцию от n . Доказать правильность вашего ответа.
- (b) Сколько в точности ячеек может пересечь горизонтальная прямая в наихудшем случае (как функция от n)? Доказать правильность вашего ответа. Допустим, что $n = 2^k - 1$ для некоторого целого числа k . [Подсказка: существует более одной функции f , такой что $f(16) = 4$.]

²¹ Термин *kd-tree* (произносится [kay dee tree]) как аббревиатура происходит от выражения «*k-dimensional tree*» (*k*-мерное дерево), но в современном толковании его этимология игнорируется, отчасти потому что никто в здравом уме никогда не должен использовать букву *k* для обозначения размерности (*dimension*) вместо явно доминирующей в этом плане буквы *d*. Этимологическая согласованность потребовала бы назвать структуру данных в этой задаче «*2d-tree*» (или, возможно, «*2-d tree*»), но в настоящее время стандартным обозначением является «*two-dimensional kd-tree*» (двумерное *kd-дерево*). Смотрите также этимологию: *B-дерево* (*B-tree*) (возможно), *альфа-форма* (*alpha shape*), *бета-остов* (*beta skeleton*), *эпсилон-сеть* (*epsilon net*), р. Потомак (*Potomac River*), р. Миссисипи (*Mississippi River*), оз. Мичиган (*Michigan Lake*), оз. Тахо (*Tahoe Lake*), о-в Манхэттен (*Manhattan Island*), Смоляные ямы Ла Бреа (*La Brea Tar Pits*), пустыня Сахара (*Sahara Desert*), гора Килиманджаро (*Mount Kilimanjaro*), Южный Вьетнам (*South Vietnam*), Восточный Тимор (*East Timor*), галактика Млечный Путь (*Milky Way Galaxy*), г. Таунсвилл (*City of Townsville*) и беспилотные автомобили (*self-driving automobiles*).

- (с) Предположим, что задано n точек, хранящихся в kd-дереве. Описать и проанализировать алгоритм, который подсчитывает количество точек над горизонтальной прямой (например, над штриховой линией на рис. 1.33) настолько быстро, насколько это возможно. [Подсказка: используйте упражнение (b).]

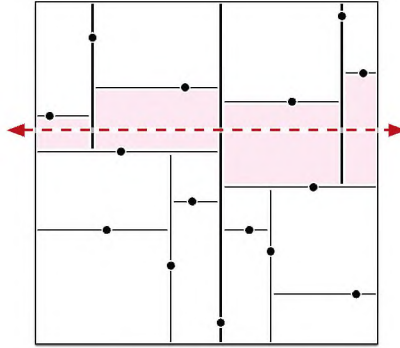


Рис. 1.33. Kd-дерево для 15 точек. Штриховая линия пересекает четыре закрашенные ячейки

- (d) Описать и проанализировать эффективный алгоритм, который подсчитывает с учетом kd-дерева, содержащего n точек, количество точек, лежащих внутри прямоугольника R с горизонтальными и вертикальными сторонами. [Подсказка: используйте упражнение (с).]
- ♥41. Боб Ратенбур (Bob Ratenbur), студент начального курса по специальности CS 225, пытается написать код для выполнения обходов в прямом и обратном направлениях (с предварительной и с отложенной выборкой) и упорядоченного обхода (с прямой выборкой) двоичных деревьев. Боб вроде бы понимает основную идею алгоритмов обхода, но когда он пытается реализовать их практически, то неизбежно путается в рекурсивных вызовах. За пять минут до срока сдачи задания Боб в отчаянной спешке выдает исходный код со структурой, показанной ниже:

```
PreOrder(v):
  if v = Null
    return
  else
    print label(v)
    [ ]Order(left(v))
    [ ]Order(right(v))
```

```
PreOrder(v):
  if v = Null
    return
  else
    print label(v)
    [ ]Order(left(v))
    [ ]Order(right(v))
```

```
PostOrder(v):
  if v = Null
    return
  else
    [ ]Order(left(v))
    [ ]Order(right(v))
    print label(v)
```

В этом псевдокоде каждый блок символов [] скрывает один из префиксов Pre, In или Post. Кроме того, каждый вызов перечислен-

ных ниже функций в коде, представленном Бобом, появляется ровно один раз:

PreOrder(left(v))	PreOrder(right(v))
InOrder(left(v))	InOrder(right(v))
PostOrder(left(v))	PostOrder(right(v))

Таким образом, существует в точности 36 возможных вариантов кода, написанного Бобом. К сожалению, Боб случайно удалил весь исходный код после успешной сдачи выполняемого файла, поэтому ни вам, ни ему неизвестно, какие именно функции вызывались в скрытых фрагментах кода.

Теперь предположим, что нам предоставлен вывод алгоритмов обхода Боба, выполненных для некоторого неизвестного двоичного дерева T . Выходные данные были предусмотрительно проанализированы и разделены на три массива $Pre[1..n]$, $In[1..n]$ и $Post[1..n]$. Можно предположить, что эти последовательности обхода согласованы ровно с одним двоичным деревом T , в частности, метки вершин этого неизвестного дерева T различны, и каждый внутренний узел в T имеет ровно двух потомков.

- Описать алгоритм восстановления неизвестного дерева T по заданным последовательностям обхода.
- Описать алгоритм, который либо восстанавливает исходный код Боба по заданным последовательностям обхода, либо верно сообщает о том, что эти последовательности обхода согласуются с несколькими группами алгоритмов.

Например, если заданы следующие входные данные:

$$\begin{aligned} Pre[1..n] &= [H A E C B I F G D] \\ In[1..n] &= [A H D C E I F B G] \\ Post[1..n] &= [A E I B F C D G H] \end{aligned}$$

то ваш первый (а) алгоритм должен возвращать дерево, показанное на рис. 1.34.

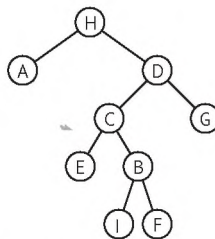


Рис. 1.34. Дерево T , восстановленное по заданным последовательностям обхода

Ваш второй (b) алгоритм должен восстановить следующий исходный код:

```
PreOrder(v):
    if v = Null
        return
    else
        print label(v)
        PreOrder(left(v))
        PostOrder(right(v))
```

```
InOrder(v):
    if v = Null
        return
    else
        PostOrder(left(v))
        print label(v)
        PreOrder(right(v))
```

```
PostOrder(v):
    if v = Null
        return
    else
        InOrder(left(v))
        InOrder(right(v))
        print label(v)
```

♥42. Пусть T – двоичное дерево, в узлах которого хранятся различные числовые значения. Напомню, что T является двоичным деревом поиска, если и только если (1) либо дерево T пустое, (2) либо дерево T соответствует перечисленным ниже рекурсивным условиям:

- левое поддерево дерева T является двоичным деревом поиска;
- все значения в левом поддереве меньше, чем значение в корне;
- правое поддерево дерева T является двоичным деревом поиска;
- все значения в правом поддереве меньше, чем значение в корне.

Рассмотрим следующую пару операций в двоичных деревьях:

- *поворот* (rotate) произвольного узла вверх²²;

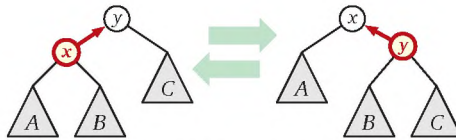


Рис. 1.35. Операция поворота узла в двоичном дереве

- *взаимная перестановка* (swar) левого и правого поддерева произвольного узла.

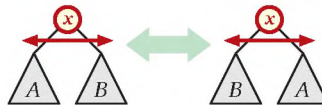


Рис. 1.36. Операция взаимной перестановки поддеревьев узла

В обеих операциях одно, все или ни одно из поддеревьев A, B, C могут быть пустыми.

(а) Описать алгоритм преобразования произвольного двоичного дерева с n узлами и с различными значениями узлов в двоичное

²² Повороты (rotations) сохраняют упорядоченную (inorder) последовательность обхода узлов в двоичном дереве. Отчасти по этой причине повороты используются для сопровождения некоторых типов сбалансированных двоичных деревьев поиска, включая AVL-деревья, красно-черные деревья, косые деревья, деревья «козла отпущения» и дуги (treaps). Конспекты лекций по большинству этих структур данных см. здесь: <http://algorithms.wtf>.

дерево поиска, использующий не более $O(n^2)$ операций поворота и взаимной перестановки. На рис. 1.37 показана последовательность из восьми операций, преобразующая двоичное дерево с пятью узлами в двоичное дерево поиска.

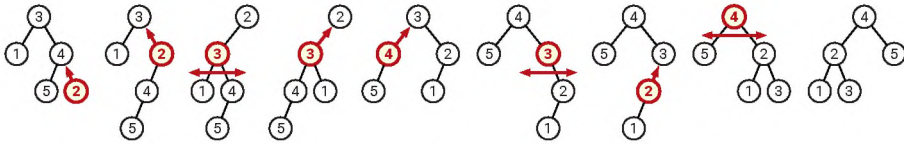


Рис. 1.37. «Сортировка» двоичного дерева: поворот узла 2, поворот узла 2, взаимная перестановка в узле 3, поворот узла 3, поворот узла 4, взаимная перестановка в узле 3, поворот узла 2, взаимная перестановка в узле 4

Этому алгоритму запрещено напрямую изменять указатели на предков или потомков, создавать новые узлы или удалять существующие узлы, единственный способ изменения дерева – выполнение операций поворота и взаимной перестановки.

С другой стороны, вы можете без ограничений выполнять любые вычисления при условии, что они не изменяют дерево, а время выполнения вашего алгоритма определяется числом выполняемых им операций поворота и взаимной перестановки.

- ♥(b) Описать алгоритм преобразования произвольного двоичного дерева с n узлами в двоичное дерево поиска, использующий не более $O(n \log n)$ операций поворота и взаимной перестановки.
- (c) Доказать, что любое двоичное дерево поиска с n узлами можно преобразовать в любое другое двоичное дерево поиска с теми же значениями узлов, используя только лишь $O(n)$ операций поворота (но без операций взаимной перестановки).
- ♥(d) Нерешенная (открытая) задача: описать алгоритм преобразования произвольного двоичного дерева с n узлами в двоичное дерево поиска, использующий только лишь $O(n)$ операций поворота и взаимной перестановки, или доказать, что такой алгоритм невозможен. [Подсказка: я полагаю, что такой алгоритм невозможен.]

Глава 2

.....

Поиск с возвратом

Однако там, где двусмысленность не может быть устранена ни с помощью канона веры, ни с помощью контекста, ничто не мешает нам придерживаться суждения, соответствующего любому методу, выбранному из тех, которые напрашиваются сами собой.

— Аврелий Августин Иннонийский (Augustine of Hippo),
О христианской доктрине (De doctrina Christiana) (397 г. н. э.)
[На англ. язык перевел Маркус Додс (Marcus Dods) (1892 г.)]

Я отказался от обеда и побежал обратно в лабораторию. Там в возбуждении я пробовал на вкус содержимое каждого стакана и чашки для выпаривания на лабораторном столе. К счастью для меня, ни один из них не содержал агрессивной или ядовитой жидкости.

— Константин Фальберг (Constantin Fahlberg) о своем открытии сахараина,
журнал Scientific American (1886 г.)

Самая большая сложность для любого мыслителя – сформулировать проблему таким образом, чтобы разрешить ее.

— приписывается Бертранду Расселу (Bertrand Russell)

Подходя к развилке дорог, не падайте духом.

— Йоги Берра (Yogi Berra) (ориентируясь в собственном доме)

В этой главе описывается другая важная рекурсивная стратегия – поиск с возвратом (backtracking). Алгоритм поиска (перебора) с возвратом пытается создавать решение вычислительной задачи постепенно, поочередно выполняя небольшие части работы. Когда алгоритм обязан выбрать один из нескольких возможных вариантов для перехода к следующему компоненту решения, он рекурсивно оценивает каждый возможный вариант, затем выбирает наилучший.

2.1. Задача об n ферзях

Типичная задача поиска с возвратом – классическая задача об n ферзях, впервые предложенная немецким любителем шахмат и составителем шахматных задач Максом Беццелем (Max Bezzel) в 1848 г. (под псевдонимом Schachfreund (товарищ по шахматам)) для стандартной доски размером 8×8 клеток, а в 1869 г. Франсуа-Жозеф Юсташ Лионетт (François-Joseph Eustache Lionet) предложил более обобщенную задачу для доски размером $n \times n$ клеток. На доске необходимо разместить n ферзей так, чтобы ни один из них не находился под боем любого другого. Для читателей, знакомых с правилами шахмат, это означает, что никакие два ферзя не должны находиться ни на одной горизонтали, ни на одной вертикали и ни на одной диагонали.

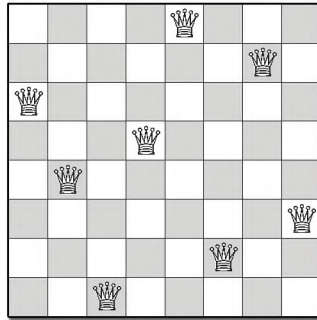


Рис. 2.1. Первое решение Гаусса задачи о восьми ферзях, представленное в виде массива [5,7,1,4,2,8,6,3]

В письме своему другу Генриху Шумахеру (Heinrich Schumacher) в 1850 г. знаменитый математик Карл Фридрих Гаусс (Carl Friedrich Gauss) писал, что можно с легкостью подтвердить заявление Франца Наука (Franz Nauck) о том, что задача о восьми ферзях имеет 92 решения, получаемые методом проб и ошибок за несколько часов. («*Schwer ist es übrigens nicht, durch ein methodisches Tatonniren sich diese Gewissheit zu verschaffen, wenn man 1 oder ein paar Stunden daran wenden will*». – «*Между прочим, нетрудно получить это подтверждение путем методичного объяснения, неважно, хотите ли вы потратить на это час или несколько часов*».) Употребляемое Гауссом слово «*Tatonniren*» происходит от французского «*tâtonner*», означающего «искать на ощупь» или «шарить вслепую», как в полной темноте.

В письме Гаусса описана следующая рекурсивная стратегия решения задачи об n ферзях, та же стратегия была описана в 1882 г. французским математиком Эдуардом Люка (Édouard Lucas), который считал автором этого метода Эммануэля Лакьера (Emmanuel Laquière). Ферзи размещаются на доске по одному на каждой горизонтали, начиная с самой верхней. Для размещения r -го ферзя мы методически проходим по всем n полям горизонтали r слева направо в простом цикле `for`. Если текущее поле на-

ходится под боем ранее размещенного ферзя, то это поле пропускается, иначе ферзь пока помещается на это поле, и продолжается рекурсивное «прощупывание» для согласованного размещения ферзей на следующих горизонталях.

На рис. 2.2 показан итоговый алгоритм, который рекурсивно вычисляет все полные решения задачи об n ферзях, связанные с заданным частным решением. Следуя стратегии Гаусса, представим позиции ферзей с помощью массива $Q[1..n]$, где $Q[i]$ обозначает поле, которое занимает ферзь на горизонтали i . При вызове `PlaceQueens` входной параметр r является индексом первой свободной горизонтали, а префикс $Q[1..r-1]$ содержит позиции первых (уже размещенных) $r-1$ ферзей. По существу, для вычисления всех решений с n ферзями без ограничений следовало бы вызвать `PlaceQueens(Q[1..n], 1)`. Внешний цикл `for` рассматривает все возможные варианты размещения ферзей на горизонтали r , внутренний цикл `for` проверяет, не находится ли рассматриваемое текущее поле на горизонтали r под боем ферзей, уже размещенных на первых $r-1$ горизонталях.

```
PlaceQueens(Q[1..n], r):
  if r = n + 1
    print Q[1..n]
  else
    for j ← 1 to n
      legal ← True
      for i ← 1 to r - 1
        if (Q[i] = j) or (Q[i] = j + r - i) or (Q[i] = j - r + i)
          legal ← False
      if legal
        Q[r] ← j
        PlaceQueens(Q[1..n], r + 1)    <<Рекурсия!>>
```

Рис. 2.2. Алгоритм поиска с возвратом Гаусса и Лакьера для задачи с n ферзями

Выполнение алгоритма `PlaceQueens` можно проиллюстрировать с помощью рекурсивного дерева (*recursion tree*). Каждый узел этого дерева соответствует рекурсивной подзадаче, следовательно, допустимому частному решению. В частности, корень соответствует пустой доске (при $r = 0$). Ребра в рекурсивном дереве соответствуют рекурсивным вызовам. Листья соответствуют частным решениям, которые невозможно улучшить, потому что на каждой горизонтали уже находится ферзь или потому что каждое поле на следующей пустой горизонтали находится под боем ранее расставленных ферзей. Поиск с возвратом (*backtracking*) для получения полных решений равнозначен поиску в глубину (*depth-first search*) в этом дереве.

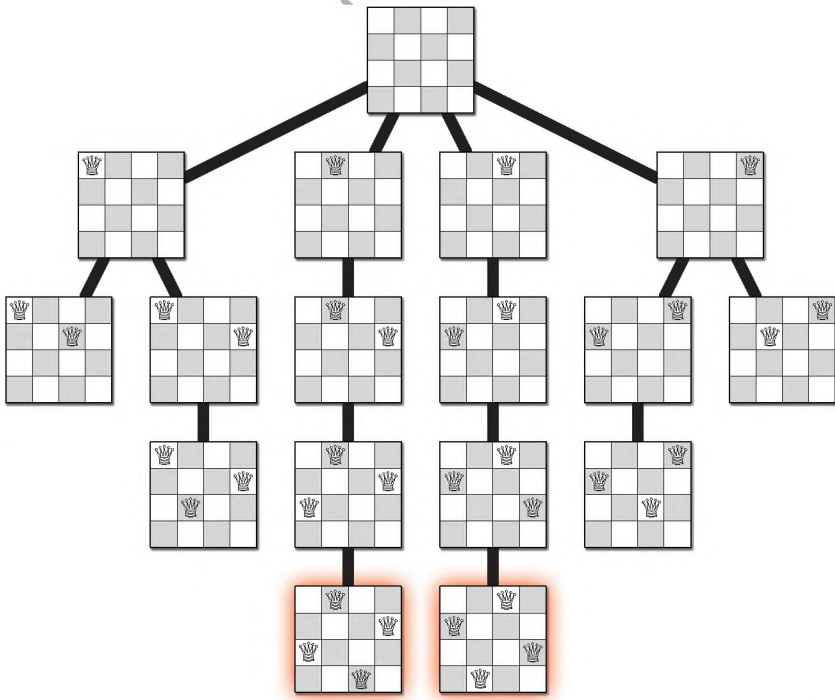


Рис. 2.3. Полное рекурсивное дерево алгоритма Гаусса и Лакьера для задачи с четырьмя ферзями

2.2. Деревья игры

Рассмотрим описанную ниже простую игру с двумя участниками¹ на квадратной доске размером $n \times n$ с границей из квадратных полей. Назовем игроков Гораций Фальберг-Ремзен (Horace Fahlberg-Remsen) и Вера Ребоди (Vera Rebaudi).² У каждого игрока имеется n фишек, которые он перемещает по доске от одного края к другому. Фишки Горация начинают игру у левого края доски, по одной фишке в каждом поле, и перемещаются по горизонтали вправо. Фишки Веры начинают игру у верхнего края доски, по одной фишке в каждом поле, и перемещаются по вертикали вниз. Игроки ходят по очереди. При каждом своем ходе Гораций может передвинуть одну фишку на соседнее свободное поле справа или перепрыгнуть через одну (и только одну) фишку Веры на свободное поле справа, т. е. переместить-

¹ Я не знаю, как называется эта игра, и я даже не уверен, что правильно описываю ее правила. Этой игре (или очень похожей на нее) меня научил Ленни Питт (Lenny Pitt), который рекомендовал играть пакетиками с заменителем сахара в ресторанах.

² Константин Фальберг (Constantin Fahlberg) и Айра Ремзен (Ira Remsen) первыми синтезировали сахарин в 1878 г., когда Фальберг занимал должность постдока в лаборатории Ремзена и исследовал продукты переработки каменного угля. В 1900 г. Овидио Ребоди (Ovidio Rebaudi) впервые опубликовал результаты химического анализа применяемого в медицине растения ka'a he'e – стевии (посконник крапиволистный), культивируемого индейцами гуарани более 1500 лет, в настоящее время общеизвестного как стевия медовая (*Stevia rebaudiana*).

ся сразу на два поля. Если нет разрешенных ходов и прыжков, то Гораций просто пропускает ход. Подобным образом Вера выполняет перемещения или прыжки одной своей фишкой вниз при каждом ходе, кроме тех случаев, когда разрешенных ходов нет. Выигрывает игрок, который первым передвинет все свои фишки к противоположному краю доски. (Легко доказать, что при наличии всех фишек на доске как минимум один игрок может сделать разрешенный ход, следовательно, в итоге кто-то обязательно должен выиграть.)

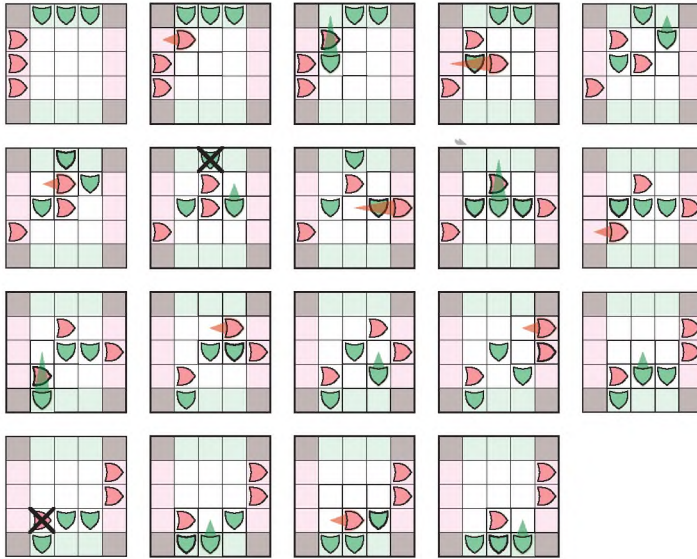


Рис. 2.4. Выигрыш Веры в игре пакетиками с заменителем сахара на доске 3×3

Если вы раньше не встречались с этой игрой³, то, вероятно, не знаете, как сыграть, чтобы добиться успеха. Как бы то ни было, существует относительно простой алгоритм поиска с возвратом, который может превосходно играть в эту игру или в любую игру с двумя игроками без случайной или скрытой информации, если игра завершается после конечного числа ходов. Таким образом, если вы случайно окажетесь в середине партии и существует возможность победы над другим безупречным игроком, то этот алгоритм подскажет вам, как выиграть.

Состояние (state) этой игры состоит из мест расположения всех фишек и обозначения текущего игрока. Такие состояния можно объединить в дерево игры (game tree), содержащее ребро от состояния x к состоянию y , если и только если текущий игрок в состоянии x может сделать допустимый ход, переводящий игру в состояние y . Корнем дерева игры является начальная позиция, и каждый путь от корня до листа соответствует завершенной игре.

³ Если встречались, то прошу сообщить мне, где именно вы ее увидели.

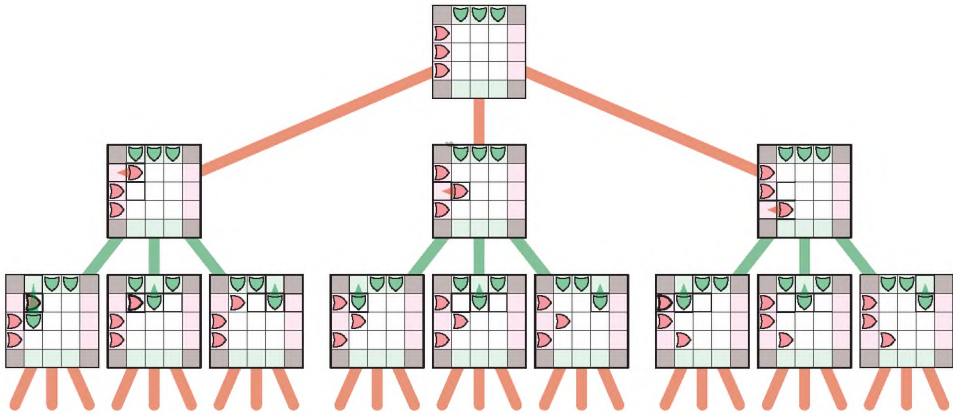


Рис. 2.5. Первые два уровня дерева игры в пакетики с заменителем сахара

Для перемещения по этому дереву игры рекурсивно определим ее состояние как хорошее (good) или плохое (bad) следующим образом:

- состояние игры хорошее, если текущий игрок уже выиграл или если текущий игрок своим ходом может перевести игру в плохое состояние для другого игрока;
- состояние игры плохое, если текущий игрок уже проиграл или если любой его доступный ход ведет к хорошему состоянию для другого игрока.

Равнозначно узел, не являющийся листом в дереве игры, называется хорошим, если он имеет хотя бы одного плохого потомка, и плохим, если все его потомки являются хорошими. С помощью индукции любой игрок, обнаруживший игру в хорошем состоянии при своем ходе, может выиграть, даже если соперник играет совершенно. С другой стороны, начиная с плохой позиции, игрок может выиграть, только если соперник допустит ошибку. Это рекурсивное определение было предложено Эрнстом Цермело (Ernst Zermelo) в 1913 г.⁴

Из этого рекурсивного определения естественным образом выводится следующий рекурсивный алгоритм поиска с возвратом, вычисляющий, является ли заданное состояние игры хорошим или плохим. По своей сути этот алгоритм фактически представляет собой поиск в глубину в дереве игры, а само дерево игры равнозначно рекурсивному дереву алгоритма. Простая модификация этого алгоритма поиска с возвратом находит хороший ход (или даже все возможные хорошие ходы), если на входе задано хорошее состояние игры.

⁴ В действительности Цермело рассматривал более узкий класс игр с конечным числом состояний, но с допуском бесконечной последовательности ходов. (Цермело определил, что бесконечная игра заканчивается ничьей.)

```

PlayAnyGame(X,player):
    if player has already won in state X
        return Good
    if player has already lost in state X
        return Bad
    for all legal moves  $X \rightsquigarrow Y$ 
        if PlayAnyGame(Y,-player) = Bad
            return Good
    return Bad

```

$\langle\langle X \rightsquigarrow Y \text{ - хороший ход.} \rangle\rangle$
 $\langle\langle \text{Нет хороших ходов.} \rangle\rangle$

Все игровые программы, по существу, основаны на этой простой стратегии поиска с возвратом. Но, поскольку в большинстве игр имеется огромное количество состояний, на практике невозможно выполнить обход всего дерева игры полностью. Вместо этого игровые программы применяют другие эвристические методы⁵ отсеечения (подрезания) дерева игры, игнорируя состояния, которые явно (или «очевидно») являются хорошими или плохими или, по крайней мере, лучше или хуже других состояний, или/и отсекая дерево на определенной глубине (на глубине просмотра позиции в полуходах – *ply*) и применяя более эффективные эвристические методы для вычисления листьев.

2.3. Задача о сумме подмножеств

Рассмотрим более сложную задачу под названием SubsetSum (задача о сумме подмножеств): дано множество X положительных целых чисел и целевое целое число T . Существует ли подмножество элементов множества X , сумма которых равна T ? Следует отметить, что возможно существование более одного такого подмножества. Например, если $X = \{8, 6, 7, 5, 3, 10, 9\}$ и $T = 15$, то ответ True (истина), потому что суммы всех подмножеств $\{8, 7\}$, $\{7, 5, 3\}$, $\{6, 9\}$ и $\{5, 10\}$ равны 15. С другой стороны, если $X = \{11, 6, 5, 1, 7, 13, 12\}$ и $T = 15$, то ответ False (ложь).

Существует два простейших случая. Если целевое значение T равно нулю, то можно немедленно возвращать ответ True, так как пустое множество является подмножеством любого множества X и сумма элементов пустого множества равна нулю⁶. Во втором случае если $T < 0$ или $T \neq 0$, но множество X пустое, то можно немедленно возвращать ответ False.

В общем случае необходимо рассматривать произвольный элемент $x \in X$. (Мы уже отработали случай, когда X – пустое множество.) Подмножество X , сумма элементов которого равна T , существует, если и только если одно из перечисленных ниже утверждений истинно:

- существует подмножество X , включающее элемент x , и его сумма равна T ;

⁵ Эвристический метод – это алгоритм, который не работает. (Исключение: работает на практике. Иногда. Возможно.)

⁶ ...Ну, потому что чему же еще могла бы равняться их сумма?..

- существует подмножество X , не включающее элемент x , и его сумма равна T .

В первом случае непременно должно существовать подмножество $X \setminus \{x\}$, сумма элементов которого равна $T - x$. Во втором случае непременно должно существовать подмножество $X \setminus \{x\}$, сумма элементов которого равна T . Таким образом, решить задачу $\text{SubsetSum}(X, T)$ можно сведением ее к двум более простым случаям: $\text{SubsetSum}(X \setminus \{x\}, T - x)$ и $\text{SubsetSum}(X \setminus \{x\}, T)$. Итоговый рекурсивный алгоритм показан ниже.

```

<<Существует ли какое-либо подмножество  $X$ , сумма элементов которого равна  $T$ ?>>
SubsetSum( $X, T$ ):
    if  $T = 0$ 
        return True
    else if  $T < 0$  or  $X = \emptyset$ 
        return False
    else
         $x \leftarrow$  any element of  $X$ 
        with  $\leftarrow \text{SubsetSum}(X \setminus \{x\}, T - x)$       <<Рекурсия!>>
        wout  $\leftarrow \text{SubsetSum}(X \setminus \{x\}, T)$       <<Рекурсия!>>
        return (with  $\vee$  wout)

```

Корректность

Доказательство корректности этого алгоритма является несложным упражнением в применении индукции. Если $T = 0$, то сумма элементов пустого подмножества равна T , следовательно, `True` является корректным выводом. Иначе если T – отрицательное число или X – пустое множество, то не существует подмножества X с суммой элементов, равной T , следовательно, `False` – корректный вывод. Иначе, если существует подмножество, сумма элементов которого равна T то такое подмножество либо содержит, либо не содержит элемент $X[n]$, и Фейя Рекурсия корректно проверит каждую из этих возможностей. Доказательство завершено.

Анализ

Для анализа этого алгоритма необходимо более точно описать некоторые подробности его реализации. Сначала предположим, что входное множество X задано как массив $X[1..n]$.

Приведенный выше рекурсивный алгоритм позволяет выбрать любой элемент $x \in X$ в основном рекурсивном процессе. Исклчительно в целях эффективности полезнее выбрать такой элемент x , чтобы оставшееся подмножество $X \setminus \{x\}$ имело наиболее краткое описание, которое можно было бы быстро вычислить, при этом рекурсивные вызовы требуют минимальных накладных расходов. Говоря конкретнее, мы выбираем в качестве x самый последний элемент $X[n]$, тогда подмножество $X \setminus \{x\}$ хранится в пре-

фиксе $X[1..n - 1]$. Передача полной копии этого префикса в рекурсивные вызовы займет слишком много времени – требуется время $\Theta(n)$ только для создания копии, – поэтому передаются только два значения: ссылка на массив (т. е. его начальный адрес) и длина префикса. (В другом варианте можно было бы исключить передачу ссылки на массив X в каждый рекурсивный вызов, если сделать X глобальной переменной.)

«Существует ли какое-либо подмножество $X[1..i]$, сумма элементов которого равна T ?»

SubsetSum(X, i, T):

if $T = 0$

return True

else if $T < 0$ or $i = 0$

return False

else

with $\leftarrow$ SubsetSum($X, i - 1, T - X[i]$) «Рекурсия!»

wout $\leftarrow$ SubsetSum($X, i - 1, T$) «Рекурсия!»

return (with $\vee$ wout)

При выборе описанных выше свойств реализации время выполнения $T(n)$ для этого алгоритма соответствует рекуррентному выражению $T(n) \leq 2T(n - 1) + O(1)$. Решение $T(n) = O(2^n)$ легко вычисляется либо с применением рекурсивных деревьев, либо с помощью еще более простого метода «Ах, да, ведь мы уже решили это рекуррентное выражение для задачи Ханойские башни». В наихудшем случае – например, если T больше суммы всех элементов X , – рекурсивное дерево для этого алгоритма является совершенным (полным) двоичным деревом с глубиной n , и алгоритм рассматривает все 2^n подмножеств X .

Варианты

При весьма незначительных изменениях можно решить несколько вариантов задачи SubsetSum. Например, на рис. 2.6 показан алгоритм, действительно выполняющий построение подмножества X , сумма элементов которого равна T , если такое подмножество существует, или возвращающий значение ошибки None, если такого подмножества не существует. Этот алгоритм использует точно такую же рекурсивную стратегию, что и ранее приведенные алгоритмы решения. Алгоритм также выполняется за время $O(2^n)$. Анализ предельно упрощается, если принять структуру данных, позволяющую вставлять один элемент за время $O(1)$ (например, связный список), но в действительности время выполнения остается равным $O(2^n)$, даже если операция вставки требует времени $O(n)$ (например, отсортированный связный список). Похожие варианты позволяют подсчитывать количество подмножеств, суммы которых равны конкретному значению, или выбирать наилучшее подмножество (по некоторому другому критерию), сумма элементов которого равна конкретному значению.


```

    <<Return a subset of X[1 .. i] that sums to T>>
    <<or None if no such subset exists>>
ConstructSubset(X, i, T):
    if T = 0
        return ?
    if T < 0 or n = 0
        return None
    Y ← ConstructSubset(X, i - 1, T)
    if Y ≠ None
        return Y
    Y ← ConstructSubset(X, i - 1, T - X[i])
    if Y ≠ None
        return Y ∪ {X[i]}
    return None

```

Рис. 2.6. Рекурсивный алгоритм поиска с возвратом для версии с построением подмножества для задачи SubsetSum

Большинство других задач, решаемых методом поиска с возвратом, обладают этим свойством: та же рекурсивная стратегия может использоваться для решения многих разнообразных вариантов одной задачи. Например, легко изменить описанную в предыдущем разделе рекурсивную стратегию, которая определяет, является ли заданная игровая позиция хорошей или плохой, на вариант, возвращающий хороший ход или список всех возможных хороших ходов. Поэтому при проектировании алгоритмов поиска с возвратом мы должны ориентироваться на самый простой возможный вариант задачи, вычисляющий единственное числовое значение или даже единственное логическое значение вместо более сложной информации или структуры.

2.4. Общий шаблон

Алгоритмы поиска с возвратом широко используются для принятия последовательности решений, целью которой является создание рекурсивно определяемой структуры, соответствующей определенным условиям. Часто (но не всегда) этой целевой структурой становится сама последовательность решений. Например:

- в задаче о расстановке n ферзей целью является последовательность позиций ферзей, по одному на каждой горизонтали, такая, что ни один из ферзей не атакует любого другого. Для каждой горизонтали алгоритм решает, где разместить ферзя;
- в задаче с деревом игры целью является последовательность допустимых ходов, такая что каждый ход является настолько хорошим, насколько это возможно, для игрока, выполняющего этот ход. Для

каждого состояния игры алгоритм решает, какой из возможных следующих ходов является наилучшим;

- в задаче `SubsetSum` целью является последовательность входных элементов, суммой которых является конкретное значение. Для каждого входного элемента алгоритм решает, включать его в выходную последовательность или нет.

(Постойте, а почему целью вычисления суммы подмножеств является последовательность? Это было преднамеренное проектное решение. Мы ввели удобное упорядочение во входном множестве – представив его с помощью массива в противоположность некоторым другим более бесформенным структурам данных, – которое можно использовать в нашем рекурсивном алгоритме.)

В каждом рекурсивном вызове алгоритма поиска с возвратом необходимо принять ровно одно решение, и этот выбор непременно должен быть логически согласован со всеми предыдущими решениями. Таким образом, каждый рекурсивный вызов требует не только определенную часть входных данных, но также соответствующий итоговый результат всех уже принятых ранее решений. Для сохранения эффективности этот итоговый результат ранее принятых решений должен иметь минимально возможный размер. Например:

- для задачи о расстановке n ферзей необходимо передавать не только количество пустых горизонталей, но и позиции всех ранее расставленных ферзей. К сожалению, здесь необходимо запоминать ранее принятые решения во всех подробностях;
- для задачи дерева игры требуется передача только текущего состояния игры с указанием игрока, выполняющего очередной ход. Нет необходимости в запоминании какой-либо информации о прошлых решениях, потому что определение победителя по заданному состоянию игры не зависит от ходов, которые создали это состояние;⁷
- для задачи `SubsetSum` необходимо передавать оставшиеся доступные целые числа и текущее целевое значение, которым является исходное целевое значение минус сумма всех ранее выбранных элементов. При этом абсолютно неважно, какие элементы были выбраны ранее.

При проектировании новых рекурсивных алгоритмов поиска с возвратом обязательно необходимо заранее четко определить, какая информация о предыдущих принятых решениях потребуется в середине (выполнения) алгоритма. Если это значимая информация, то проектируемый рекурсив-

⁷ Во многих играх кажется, что это условие независимости нарушается. Например, стандартные правила шахмат и шашек позволяют объявить ничью при трехкратном повторении одной и той же позиции фигур, в китайских правилах го просто запрещено повторение любого более раннего расположения камней. Таким образом, в этих играх состояние формально включает не только текущие позиции фигур, но еще и полную историю предыдущих ходов.

ный алгоритм может потребовать решения более общей задачи, нежели изначально предложенной для решения. (Ранее мы уже видели этот тип обобщения: для поиска среднего значения в неотсортированном массиве за линейное время была создана версия алгоритма выбора k -го наименьшего элемента при произвольном значении k .)

И после окончательного определения, какую именно рекурсивную задачу действительно необходимо решить, мы решаем эту задачу методом рекурсивного прямого перебора (*recursive brute force*): перебрать все возможные варианты следующего решения, которые согласовываются с ранее принятыми решениями, и позволить Фее Рекурсии позаботиться об остальном. Здесь не следует ничего усложнять. Не пропускать «очевидно» бессмысленные решения. Проверять все варианты. Сделать алгоритм быстрее можно и потом.

2.5. Сегментация текста (Interpunctio Verborum)

Предположим, что задана строка букв, представляющая текст на некотором иностранном языке, но без пробелов и знаков пунктуации, и необходимо разделить эту строку на отдельные осмысленные слова. Например, может быть задано следующее высказывание из знаменитой речи Цицерона (Cicero) в защиту Луция Лициния Мурены (Lucius Licinius Murena) в 62 г. до н. э. в форме стандартной непрерывной записи (*scriptio continua*), характерной для классической латыни⁸:

PRIMVSDIGNITASINTAMTENVISCIENTIANONPOTEST
ESSERESENIMSVNTPARVAEPROPEINSINGVLISLITTERIS
ATQVEINTERPVNCTIONIBUSVERBORVMOCVPATAE

Читатель, свободно владеющий латынью, должен распознать эту строку (в современной орфографии), как «*Primus dignitas in tam tenui scientia non potest esse; res enim sunt parvae, prope in singulis litteris atque interpunctionibus verborum occupatae*».⁹ Сегментация текста – это задача не только для классического латинского и греческого языков, но и для некоторых современных языков и форм письменности, включая бакийский, бирманский, китайский, японский, яванский, кхмерский, лаосский, тайский, тибетский и

⁸ В действительности в классических латинских рукописях слова разделены маленькими средними точками, называемыми интерпунктами (interpunct). Интерпункты в текстах полностью исчезли к III в. н. э., их заменила форма непрерывной записи (*scriptio continua*). Явные пробелы между словами были введены ирландскими монахами в VIII в. н. э. и медленно распространялись в Европе в течение нескольких следующих столетий. Форма непрерывной записи сохранилась в начале XXI в. в английском языке (и прочих языках) в виде URL и хештегов. [#octotherps4lyfe](#)

⁹ Вольный перевод: «Прежде всего, высокая должность невозможна при таких ничтожных знаниях; это очевидно и главным образом касается отдельных букв и расстановки точек между словами». Цицерон открыто высмеивает судебную экспертизу своего друга(!) и известного юриста Сервия Сульпиция Руфа (Servius Sulpicius Rufus), который обвинял Мурену в подкупе избирателей после того, как Мурена победил Руфа при выборах на должность консула. Мурена был оправдан, отчасти благодаря такой «активной» защите Цицерона, хотя почти наверняка он все же был виновен. [#librapondo](#) [#nunquamestfidelis](#)

вьетнамский языки. Аналогичные задачи возникают при разделении (сегментации) текста на английском языке без знаков пунктуации на отдельные предложения¹⁰, при разделении текста на строки для типографского набора, для распознавания речи и рукописного текста, для упрощения (сглаживания) кривых и для некоторых типов анализа временных рядов. Для наглядности я буду использовать примеры сегментации последовательности букв современного английского алфавита для получения современных английских слов.

Разумеется, некоторые строки можно сегментировать несколькими различными способами, например строку **BOTHEARTHANDSATURNSPIN** можно разделить на английские слова так: **BOTH · EARTH · AND · SATURN · SPIN**. Или так: **BOT · HEART · HANDS · AT · URNS · PIN** без учета других возможных вариантов. Но сейчас мы рассмотрим чрезвычайно простую задачу сегментации: если задана строка символов, можно ли вообще разделить ее на английские слова?

Чтобы сделать задачу более определенной (и независимой от языка), предположим, что у нас есть доступ к подпрограмме `IsWord(w)`, которая принимает строку `w` как входные данные и возвращает `True`, если `w` – «слово», или `False`, если `w` не является «словом». Например, если мы пытаемся разделить входную строку на палиндромы, то «слово» становится синонимом «палиндрома», следовательно, `IsWord(ROTATOR) = True`, но `IsWord(PALINDROME) = False`.

Как и в задаче `SubsetSum`, входной структурой является последовательность, на этот раз содержащая буквы, а не числа, поэтому естественно рассматривать процесс принятия решения, принимающий входные символы в порядке слева направо. Выходная структура является последовательностью слов, поэтому естественно рассматривать процесс, генерирующий итоговые слова в том же порядке: слева направо. Таким образом, перепрыгивая сразу в середину процесса сегментации, мы можем представить себе следующую картину, показанную на рис. 2.7.



Рис. 2.7. Середина процесса разделения строки на слова

На рис. 2.7 черная вертикальная полоса отделяет предыдущие принятые решения – разделение первых 17 букв на четыре слова – от той части входной строки, которая пока еще не обработана.

Следующий этап этого воображаемого процесса – принятие решения о том, где заканчивается следующее слово, включаемое в выходную последовательность. В рассматриваемом здесь конкретном примере существует четыре возможных варианта следующего выходного слова – **HE**, **HEAR**, **HEART** и **HEARTH**. Мы не имеем представления о том, какой из этих вариантов согла-

¹⁰ Святой Августин в трактате «О христианской доктрине» (*De doctrina Christiana*) целую главу посвятил устранению неоднозначности в латинском манускрипте с помощью добавления знаков пунктуации.

сован (если вообще есть согласованный вариант) с полной сегментацией входной строки. В этой точке процесса можно стать «разумным» и попытаться точно определить, какие варианты выбора хороши, но для этого потребуется мыслительный процесс. Вместо этого мы будем пытаться «тупо» перебирать все возможные варианты и позволим Фее Рекурсии сделать всю основную работу.

- Сначала для пробы примем **HE** как следующее слово и позволим Фее Рекурсии принять все остальные решения.

BLUE	STEM	UNIT	ROBOT	HE	ARTHANDSATURNSPIN
------	------	------	-------	----	-------------------

Рис. 2.8. Пробный выбор следующего слова **HE**

- Затем для пробы примем **HEAR** как следующее слово и позволим Фее Рекурсии принять все остальные решения.

BLUE	STEM	UNIT	ROBOT	HEAR	THANDSATURNSPIN
------	------	------	-------	------	-----------------

Рис. 2.9. Пробный выбор следующего слова **HEAR**

- Затем для пробы примем **HEART** как следующее слово и позволим Фее Рекурсии принять все остальные решения.

BLUE	STEM	UNIT	ROBOT	HEART	HANDSATURNSPIN
------	------	------	-------	-------	----------------

Рис. 2.10. Пробный выбор следующего слова **HEART**

- Затем для пробы примем **HEARTH** как следующее слово и позволим Фее Рекурсии принять все остальные решения.

BLUE	STEM	UNIT	ROBOT	HEARTH	ANDSATURNSPIN
------	------	------	-------	--------	---------------

Рис. 2.11. Пробный выбор следующего слова **HEARTH**

Как только Фей Рекурсия оповестит нас об успехе хотя бы один раз, мы сообщаем об успешном завершении процесса. С другой стороны, если Фей Рекурсия никогда не объявит об успехе, в частности, если множество возможных следующих слов пустое, то сообщаем о неудачном завершении процесса.

Ни одно из наших предыдущих принятых решений не влияет на то, какие варианты выбора доступны в текущий момент, значение имеет только суффикс из символов, пока оставшихся необработанными. В сущности, несколько различных последовательностей ранее принятых решений могут привести к одному и тому же суффиксу, тем не менее в любом случае остается абсолютно одинаковое множество вариантов выбора из этого суффикса.

BLUE	STEM	UNIT	ROBOT	HEARTHANDSATURNSPIN
BLUEST	EMU	NITRO	BOT	HEARTHANDSATURNSPIN

Рис. 2.12. Две различных последовательности решений, приводящие к одинаковому суффиксу

Поэтому можно упростить схему рекурсивного процесса на рис. 2.12, исключив часть, находящуюся слева от вертикальной черной полосы, как показано на рис. 2.13.

HEARTHANDSATURNSPIN

Рис. 2.13. Суффикс после исключения всех предыдущих решений

Теперь мы продолжаем придерживаться простой и естественной стратегии поиска с возвратом: выбрать первое выходное слово и рекурсивно сегментировать (разделить на слова) остаток входной строки.

Для получения полного рекурсивного алгоритма необходим базис рекурсии. Наша рекурсивная стратегия терпит неудачу при достижении конца входной строки, потому что нет следующего слова. К счастью, для пустой строки существует особый случай сегментации – разделение на ноль слов.

Собрав все составляющее в единое целое, в итоге получаем показанный ниже простой рекурсивный алгоритм:

```

Splittable(A[1..n]):
  if n = 0
    return True
  for i ← 1 to n
    if IsWord(A[1..i])
      if Splittable(A[i + 1..n])
        return True
  return False

```

Построение индекса

На практике передача массивов как входных параметров выполняется слишком медленно, поэтому следует найти более эффективный способ описания рекурсивных подзадач. На этапе проектирования алгоритма весьма полезно рассматривать начальный входной массив как глобальную переменную, а затем переопределить задачу и алгоритм с учетом индексов массива вместо явных подмассивов.

В рассматриваемой здесь задаче сегментации аргументом любого рекурсивного вызова всегда является суффикс $A[i..n]$ начального входного массива. Поэтому если рассматривать входной массив $A[1..n]$ как глобаль-

ную переменную, то можно переопределить поставленную рекурсивную задачу следующим образом:

Пусть задан индекс i . Найти сегментацию суффикса $A[i..n]$.

Для описания соответствующего алгоритма требуются две логические функции:

- для любых индексов i и j пусть $\text{IsWord}(i, j) = \text{True}$, если и только если подстрока $A[i..j]$ является словом (будем считать, что такая функция у нас есть);
- для любого индекса i пусть $\text{Splittable}(i) = \text{True}$, если и только если суффикс $A[i..n]$ можно разделить на слова. (Эту функцию необходимо реализовать.)

Например, $\text{IsWord}(1, n) = \text{True}$, если и только если вся входная строка является одним словом, а $\text{Splittable}(1) = \text{True}$, если и только если всю входную строку можно сегментировать (разделить на слова). Применяемая ранее рекурсивная стратегия позволяет получить следующее рекуррентное выражение:

$$\text{Splittable}(i) = \begin{cases} \text{True} & , \text{ если } i > n, \\ \bigvee_{j=i}^n (\text{IsWord}(i, j) \wedge \text{Splittable}(j + 1)) & \text{ иначе.} \end{cases}$$

Это в точности тот же алгоритм, который мы видели выше, единственным изменением является форма записи (нотация). Схожесть проявляется в еще большей степени, если записать это рекуррентное выражение на псевдокоде:

```

<<Является ли суффикс A[i..n] разделяемым на слова?>>
Splittable(i):
    if i > n
        return True
    for j ← i to n
        if IsWord(i, j)
            if Splittable(j+1)
                return True
    return False

```

Несмотря на то что это может выглядеть как простое различие в форме записи, использование индексной нотации вместо передачи массива является важным характерным приемом не только для ускорения работы алгоритмов поиска с возвратом на практике, но и для разработки алгорит-

мов динамического программирования, которые будут рассматриваться в следующей главе.

♥ Анализ

Не должен стать неожиданным тот факт, что большинство алгоритмов поиска с возвратом в наихудшем случае выполняются за экспоненциальное время. Для анализа точного времени выполнения многих из этих алгоритмов требуются методики, не относящиеся к теме данной книги. К счастью, большинство алгоритмов поиска с возвратом в этой книге представляет собой лишь промежуточные результаты на пути к более эффективным алгоритмам, т. е. точное время их выполнения в наихудшем случае в действительности не столь важно. (Сначала заставьте алгоритм работать, потому что улучшаете его скорость.)

Но в качестве дополнительного занимательного упражнения все же проанализируем время выполнения рекурсивного алгоритма `Splittable`. Поскольку неизвестно, что именно делает функция `IsWord`, мы не можем узнать, сколько времени требуется на вызов этой функции, поэтому придется анализировать время выполнения в количестве вызовов `IsWord`¹¹. Алгоритм `Splittable` вызывает функцию `IsWord` для каждого префикса входной строки, и, возможно, функция вызывает себя рекурсивно для каждого суффикса выходной строки. Таким образом, «время выполнения» алгоритма `Splittable` определяется жутковато выглядящим рекуррентным выражением:

$$T(n) \leq \sum_{i=0}^{n-1} T(i) + O(n).$$

На самом деле это не так страшно, как кажется, особенно после применения описанного ниже приема.

Сначала заменим член $O(n)$ на явное выражение an с некоторой неизвестной (и абсолютно неважной) постоянной a . Потом с изрядной долей осторожности предположим, что алгоритм действительно выполняет каждый возможный рекурсивный вызов.¹² Затем можно преобразовать рекуррентное выражение «полная хронология» в рекуррентное выражение «ограниченная хронология», рекуррентно выполняя вычитание для $T(n-1)$, как показано ниже:

¹¹ В действительности, поскольку функция `IsWord` выполняется за полиномиальное время, для выполнения алгоритма `Splittable` требуется время $O(2^n)$.

¹² Это предположение чрезмерно осторожно для сегментации английских слов, так как большинство строк, состоящих из букв, не является английскими словами, но не для похожей задачи сегментации последовательностей английских слов на грамматически правильные английские предложения. Например, рассмотрим последовательность из n копий английского слова «buffalo», или из n копий слова «police», или из n копий слова «can» при любом положительном целом n . (Так, в кабаре Мулен Руж танцы, изолируемые в металлических цилиндрах от исполнителей других танцев, могут превратиться в файер-шоу, которые случаются в контейнерах для мусора в тюремных туалетах.)

$$T(n) = \sum_{i=0}^{n-1} T(i) + \alpha n.$$

$$T(n-1) = \sum_{i=0}^{n-2} T(i) + \alpha(n-1).$$

$$\Rightarrow T(n) - T(n-1) = T(n-1) + \alpha.$$

Это конечное рекуррентное выражение упрощается до $T(n) = 2T(n-1) + \alpha$. В этот момент мы можем с уверенностью предсказать (или вывести с помощью рекурсивных деревьев, или вспомнить результат анализа алгоритма решения «Ханойских башен»), что $T(n) = O(2^n)$. Разумеется, эта верхняя граница не является жестким ограничением для доказательства по индукции от исходного рекуррентного выражения полной хронологии.

Более того, этот анализ является строгим. Существует ровно 2^{n-1} возможных способов сегментирования строки длиной n – каждый входной символ либо завершает слово, либо не завершает, за исключением самого последнего входного символа, который всегда завершает последнее слово. В наихудшем случае рассматриваемый здесь алгоритм `Splittable` исследует каждую из этих 2^{n-1} возможностей.

Варианты

Теперь, когда мы вооружены основным рекурсивным шаблоном, можно использовать его для решения множества разнообразных вариантов задачи сегментации точно так же, как это было сделано для задачи `SubsetSum`. Здесь я разберу только один пример, другие вариации рассматриваются в упражнениях. Как обычно, начальными входными данными для этой задачи является массив $A[1..n]$.

Если строку можно разделить на более чем одну последовательность слов, то, возможно, потребуется найти наилучший вариант сегментации по некоторому критерию; и наоборот, если входную строку невозможно разделить на слова, то, возможно, потребуется вычисление наилучшего варианта сегментации, который можно найти, вместо явного сообщения о критической ошибке. Для достижения обеих указанных целей предположим, что у нас есть доступ ко второй функции `Score`, принимающей строку как входные данные и возвращающей числовое значение. Например, можно присвоить высокие оценки более длинным или чаще всего употребляемым словам, а низкие оценки – коротким или редко употребляемым словам, небольшие отрицательные оценки – для незначительных ошибок в правописании, а более значимые отрицательные оценки – для последовательностей, которые, очевидно, не являются словами. Наша цель – найти вариант сегментации с максимальной суммой оценок выделенных сегментов.

Для любого индекса i пусть $\text{MaxScore}(i)$ обозначает максимальную оценку любой сегментации суффикса $A[i..n]$, и необходимо вычислить $\text{MaxScore}(1)$. Эта функция определяется следующим рекуррентным выражением:

$$\text{MaxScore}(i) = \begin{cases} 0 & , \text{ если } i > n, \\ \max_{i \leq j \leq n} (\text{Score}(A[i..j]) + \text{MaxScore}(j + 1)) & \text{ иначе.} \end{cases}$$

Это определено то же самое рекуррентное выражение, которое было выведено для алгоритма `Splittable`. Различие заключается лишь в том, что логические операторы $\vee$ и $\wedge$ заменены арифметическими операторами $\max$ и $+$.

2.6. Максимальная возрастающая подпоследовательность

Для любой последовательности S ее подпоследовательность (subsequence) является другой последовательностью, получаемой из S посредством удаления нуля или более элементов без изменения порядка оставшихся элементов. Элементы в подпоследовательности не обязательно должны быть смежными (соседними) в последовательности S . Например, когда вы едете на автомобиле по главной улице любого города, то проезжаете последовательность перекрестков со светофорами, но должны останавливаться только на подпоследовательности тех перекрестков, на которых встречается красный сигнал светофора. Если очень повезет, то вы проедете вообще без остановок: это пустая подпоследовательность последовательности S . Но если вы крайне невезучий человек, то, возможно, будете останавливаться на каждом перекрестке: S является подпоследовательностью самой себя.

Другой пример: все строки **BENT**, **ACKACK**, **SQUARING** и **SUBSEQUENT** являются подпоследовательностями строки **SUBSEQUENCEBACKTRACKING**, как и пустая строка и вся строка **SUBSEQUENCEBACKTRACKING**, но строки **QUEUE**, **EQUUS** и **TALLYNO** ее подпоследовательностями не являются. Подпоследовательность, элементы которой являются смежными (соседними) в исходной последовательности, называются подстрокой (substring). Например, **MASHR** и **LAUGHTER** – подпоследовательности строки **MANSLAUGHTER**, но только **LAUGHTER** является подстрокой.

Теперь предположим, что задана последовательность целых чисел, и необходимо найти самую длинную подпоследовательность, элементы которой расположены в возрастающем порядке. Более точно: входные данные представлены как массив целых чисел $A[1..n]$, необходимо вычислить самую длинную возможную последовательность индексов $1 \leq i_1 < i_2 < \dots < i_\ell \leq n$ такую, что $A[i_k] < A[i_{k+1}]$ для всех k .

Естественный подход к построению этой **максимальной возрастающей подпоследовательности** (*longest increasing subsequence*) заключается в принятии решения для каждого индекса j в порядке от 1 до n , нужно или не нужно включать элемент $A[j]$ в искомую подпоследовательность. Переходя сразу в середину этой последовательности принятия решений, мы можем представить себе ситуацию, изображенную на рис. 2.14.



Рис. 2.14. Середина процесса выбора максимальной возрастающей подпоследовательности

Как и в ранее приведенных примерах сегментации текста, вертикальная черная полоса отделяет ранее принятые решения от пока еще не обработанной части входных данных. Числа, которые решено включить в подпоследовательность, выделены полужирным шрифтом и красным цветом, а исключенные числа затенены. (Обратите внимание: числа, которые мы решили включить, возрастают.) Алгоритм должен решить, включать или не включать в подпоследовательность число, следующее непосредственно за вертикальной черной полосой.

В рассматриваемом здесь примере определенно нельзя включить в искомую подпоследовательность число 5, потому что выбранные числа уже не будут располагаться в возрастающем порядке. Поэтому пропускаем 5 и переходим к следующему шагу принятия решения, как показано на рис. 2.15.

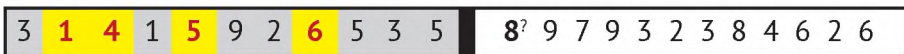


Рис. 2.15. Следующий шаг выбора максимальной возрастающей последовательности

Теперь можно включить число 8, но пока не ясно, следует ли это сделать. Вместо попытки быть «интеллектуальным», применяемый здесь алгоритм поиска с возвратом использует «грубую силу», т. е. прямой перебор.

- Сначала мы экспериментально включаем число 8 в искомую подпоследовательность и позволяем Фее Рекурсии принять остальные решения.
- Затем мы экспериментально исключаем число 8 из искомой подпоследовательности и позволяем Фее Рекурсии принять остальные решения.

Вариант выбора, который приводит к получению максимальной возрастающей подпоследовательности, является правильным решением. (Это в точности тот же самый рекурсивный шаблон, который использовался для решения задачи SubsetSum.)

Теперь главный вопрос: что необходимо запоминать о предыдущих решениях? Можно включить только элемент $A[j]$, если итоговая подпоследовательность расположена в возрастающем порядке. Если мы предполагаем (индуктивно!), что ранее выбранные числа из интервала $A[1..j-1]$ расположены в возрастающем порядке, то можно включить элемент $A[j]$, если и только если $A[j]$ больше, чем самое последнее число, выбранное из $A[1..j-1]$. Таким образом, необходима только информация о самом **последнем выбранном к текущему моменту числе**. Теперь можно изменить графические схемы, убрав все ненужное, как показано на рис. 2.16.

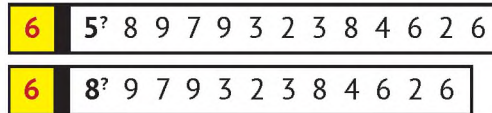


Рис. 2.16. Принятие решения на основе сохраненной информации о последнем выбранном числе

Таким образом, задача с применением выбранной рекурсивной стратегии в действительности решается методом, описанным ниже:

Дано: целое число $prev$ и массив $A[1..n]$. Найти максимальную возрастающую подпоследовательность массива A , в которой каждый элемент больше $prev$.

Как обычно, для применяемой здесь рекурсивной стратегии требуется простейший базовый вариант. Текущая стратегия перестает работать корректно при достижении конца массива, потому что «отсутствует следующее рассматриваемое число». Но пустой массив содержит ровно одну подпоследовательность, а именно пустую последовательность. Бессмысленно говорить о том, что каждый элемент в пустой последовательности больше любого заданного значения и что любая пара элементов в пустой последовательности расположена в возрастающем порядке. Следовательно, максимальная возрастающая подпоследовательность пустого массива имеет длину 0.

Ниже приведен итоговый рекурсивный алгоритм:

```

LISbigger(prev, A[1..n]):
    if n = 0
        return 0
    else if A[1] ≤ prev
        return LISbigger(prev, A[2..n])
    else
        skip ← LISbigger(prev, A[2..n])
        take ← LISbigger(A[1], A[2..n]) + 1
        return max{skip, take}

```


Но при этом следует помнить, что передача массивов в стеке вызовов увеличивает накладные расходы. Попробуем изменить решение с учетом индексов массива, считая, что массив $A[1..n]$ является глобальной переменной. Обычно целое число $prev$ – это элемент массива $A[i]$, а остальной массив всегда представлен суффиксом $A[j..n]$ начального входного массива. Поэтому можно изменить формулировку рассматриваемой здесь рекурсивной задачи следующим образом:

Дано: два индекса i и j , причем $i < j$. Найти максимальную возрастающую подпоследовательность массива $A[j..n]$, в которой каждый элемент больше $A[i]$.

Пусть $LISbigger(i, j)$ обозначает длину максимальной возрастающей подпоследовательности массива $A[j..n]$, в которой каждый элемент больше $A[i]$. Применяемая здесь рекурсивная стратегия дает следующее рекуррентное выражение:

$$Splittable(i) = \begin{cases} 0 & , \text{ если } j > n, \\ LISbigger(i, j+1) & , \text{ если } A[i] \geq A[j], \\ \max \left\{ \begin{array}{l} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{array} \right\} & \text{ иначе.} \end{cases}$$

Или если вы предпочитаете псевдокод:

```
LISbigger(i, j):
  if j > n
    return 0
  else if A[i] ≥ A[j]
    return LISbigger(i, j+1)
  else
    skip ← LISbigger(i, j+1)
    take ← LISbigger(j, j+1)+1
    return max{skip, take}
```

Для завершения необходимо объединить используемую рекурсивную стратегию с исходной задачей: поиск максимальной возрастающей подпоследовательности любого массива без каких-либо ограничений. Самый простой подход: добавить искусственное контрольное значение $-\infty$ в начало массива.

```

LIS(A[1..n]):
  A[0] ← -∞
  return LISbigger(0,1)

```

Время выполнения LISbigger соответствует рекуррентному выражению для алгоритма «Ханойских башен» $T(n) \leq 2T(n-1) + O(1)$, из которого обычно выводится $T(n) = O(2^n)$. Это время не должно удивлять, поскольку в наихудшем случае алгоритм проверяет каждую из 2^n подпоследовательностей входного массива.

2.7. Максимальная возрастающая подпоследовательность, дубль 2

Рассматриваемая в предыдущем разделе стратегия поиска с возвратом не единственная методика, которую можно использовать для поиска максимальных возрастающих подпоследовательностей. Вместо перебора по одному элементу входной последовательности можно попытаться сформировать выходную последовательность по одному элементу за один шаг. То есть вместо вопроса «Является ли $A[i]$ следующим элементом выходной последовательности?» можно спросить напрямую: «Где находится следующий элемент выходной последовательности (если он существует)?».

Переходя сразу в середину этой стратегии, можно получить схему, показанную на рис. 2.17. Предположим, что мы только что решили включить в выходную последовательность число 6, находящееся непосредственно слева от вертикальной черной линии, и теперь нужно решить, какой элемент справа от вертикальной линии должен быть следующим в выходной последовательности.

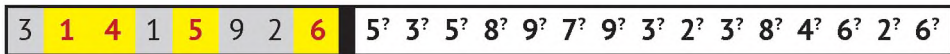


Рис. 2.17. Определение следующего элемента выходной последовательности

Разумеется, можно выбирать только те числа справа, которые больше 6, иначе выходная последовательность не будет возрастающей.



Рис. 2.18. Допустимые варианты выбора следующего элемента

Но мы не можем сразу определить, какое из этих чисел, больших шести, является наилучшим выбором, а попытка разумного вычисления наилучшего выбора требует слишком большого объема работы, и это в любом случае приведет к возникновению проблемы. Вместо этого мы пронумеруем все возможные варианты методом прямого перебора и позволим Фее Рекурсии вычислить каждый вариант.

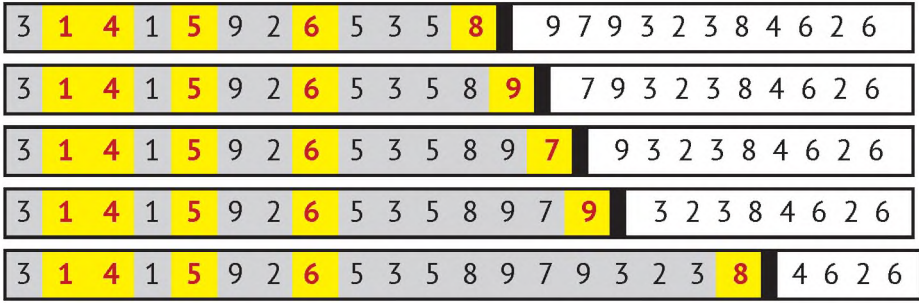


Рис. 2.19. Обозначение всех возможных вариантов выбора следующего элемента

Подмножество чисел, которые можно рассматривать как следующий элемент, зависит только от самого последнего числа, которое мы решили включить в выходную последовательность. Следовательно, можно упростить схему процесса принятия решения, удалив все элементы слева от вертикальной линии за исключением самого последнего числа, включенного в выходную последовательность.

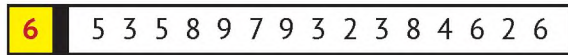


Рис. 2.20. Измененная схема процесса принятия решения

Оставшаяся последовательность чисел – это просто суффикс начального входного массива. Таким образом, если мы считаем входной массив $A[1..n]$ глобальной переменной, то можем сформулировать эту рекурсивную задачу с учетом индексов массива, как показано ниже:

Дано: индекс i . Найти максимальную возрастающую подпоследовательность массива $A[i..n]$, которая начинается с элемента $A[i]$.

Пусть $LISfirst(i)$ обозначает длину максимальной возрастающей подпоследовательности массива $A[i..n]$, которая начинается с элемента $A[i]$. Тогда можно сформулировать рекурсивную стратегию поиска с возвратом в виде следующего рекурсивного определения:

$$LISfirst(i) = 1 + \max\{LISfirst(j) \mid j > i \text{ и } A[j] > A[i]\}.$$

Поскольку мы имеем дело с множествами натуральных чисел, определим $\max \emptyset = 0$. Далее автоматически получаем $LISfirst(i) = 1$, если $A[j] \leq A[i]$ для всех $j > i$, в частности $LISfirst(n) = 1$. Это простейшие базовые варианты для применяемого здесь рекуррентного выражения.

Приведенное выше рекуррентное выражение можно записать и на псевдокоде, как показано ниже:

```

LISfirst(i):
    best ← 0
    for j ← i + 1 to n
        if A[j] > A[i]
            best ← max{best, LISfirst(j)}
    return 1 + best

```

Для завершения необходимо (как и в предыдущем разделе) объединить этот рекурсивный алгоритм с исходной задачей поиска максимальной возрастающей подпоследовательности без какой-либо информации о ее первом элементе. Естественный подход, который работает, – попытка прямого перебора всех возможных первых элементов. И в этом случае также можно добавить контрольное значение $-\infty$ в начало массива, найти максимальную возрастающую подпоследовательность, начинающуюся с этого контрольного значения, а в конце процесса отбросить это значение.

```

LIS(A[1..n]):
    best ← 0
    for i ← 1 to n
        best ← max{best, LISfirst(i)}
    return best

```

```

LIS(A[1..n]):
    A[0] ←  $-\infty$ 
    return LISfirst(0) - 1

```

2.8. Оптимальные двоичные деревья поиска

Последний пример объединяет рекурсивный поиск с возвратом и стратегию «разделяй и властвуй». Напомню, что время выполнения успешного поиска в двоичном дереве поиска пропорционально числу предков целевого узла¹³. В итоге время поиска в наихудшем случае пропорционально глубине дерева. Следовательно, для минимизации времени поиска в наихудшем случае высота дерева должна быть минимальной, насколько это возможно, – по этой метрике идеальное дерево является полностью сбалансированным.

Но во многих приложениях двоичных деревьев поиска более важно минимизировать общую стоимость нескольких операций поиска, а не стоимость наихудшего случая одной операции поиска. Если x – более частая цель поиска, чем y , то можно сэкономить время, сформировав дерево, в котором глубина x меньше глубины y , даже если это означает увеличение общей глубины дерева. Полностью сбалансированное дерево не является наилучшим вариантом выбора, если некоторые элементы значительно чаще запрашиваются, нежели прочие. На практике абсолютно несбалансированное дерево с глубиной $\Omega(n)$ может действительно оказаться наилучшим вариантом выбора.

¹³ Предок (ancestor) узла v – это либо сам узел, либо предок родителя узла v . Истинный предок (proper ancestor) узла v – это либо родитель узла v , либо истинный предок родителя узла v .

В подобной ситуации предлагается следующая задача. Предположим, что задан отсортированный массив ключей $A[1..n]$ и массив соответствующих частот доступа $f[1..n]$. Наша цель – сформировать двоичное дерево поиска, минимизирующее общее время поиска, предполагая, что будет выполняться ровно $f[i]$ операций поиска для каждого ключа $A[i]$.

Прежде чем задуматься над тем, как решить эту задачу, в первую очередь мы должны найти правильное рекурсивное определение функции, которую мы пытаемся оптимизировать. Предположим, что задано также двоичное дерево поиска T с n узлами. Пусть $v_1, v_2, \dots, v_n$ – узлы дерева T , проиндексированные в отсортированном порядке, так что каждый узел v_i хранит соответствующий ключ $A[i]$. Если не учитывать постоянные коэффициенты, то общая стоимость выполнения всех операций двоичного поиска определяется следующим выражением:

$$Cost(T, f[1..n]) := \sum_{i=1}^n f[i] \cdot \# \text{число предков } v_i \text{ в } T. \quad (*)$$

Теперь предположим, что v_r – корень дерева T . По определению v_r является предком каждого узла дерева T . Если $i < r$, то все предки узла v_i , за исключением корня, находятся в левом поддереве дерева T . А если $i > r$, то все предки узла v_i , за исключением корня, находятся в правом поддереве дерева T . Таким образом, можно разделить функцию стоимости на три части, как показано ниже:

$$\begin{aligned} Cost(T, f[1..n]) = & \sum_{i=1}^n f[i] + \sum_{i=1}^{r-1} f[i] \cdot \# \text{число предков } v_i \text{ в } left(T) \\ & + \sum_{i=r+1}^n f[i] \cdot \# \text{число предков } v_i \text{ в } right(T). \end{aligned}$$

Здесь вторая и третья суммы выглядят точно так же, как исходное определение (*) для $Cost(T, f[1..n])$. Далее простая подстановка дает рекуррентное выражение для $Cost$:

$$\begin{aligned} Cost(T, f[1..n]) = & \sum_{i=1}^n f[i] + Cost(left(T), f[1..r-1]) \\ & + Cost(right(T), f[r+1..n]) \end{aligned}$$

Базовым вариантом для этого рекуррентного выражения, как обычно, является $n = 0$, т. е. стоимость выполнения ни одной операции поиска в пустом дереве равна нулю.

Теперь наша цель – вычисление дерева T_{opt} , минимизирующего эту функцию стоимости. Предположим, мы каким-то чудесным образом

узнали, что корнем дерева T_{opt} является v_r . Тогда из рекурсивного определения $\text{Cost}(T, f)$ естественным образом следует, что левое поддереву $\text{left}(T_{\text{opt}})$ непременно должно быть оптимальным деревом поиска для ключей $A[1..r-1]$ и частот доступа $f[1..r-1]$. Точно так же правое поддереву $\text{right}(T_{\text{opt}})$ обязательно должно быть оптимальным деревом поиска для ключей $A[r+1..n]$ и частот доступа $f[r+1..n]$. После выбора правильного ключа для хранения в корневом узле Фейя Рекурсия сформирует остальную часть оптимального дерева.

В более общем смысле пусть $\text{OptCost}(i, k)$ обозначает общую стоимость оптимального дерева поиска для интервала частот $f[i..k]$. Эта функция должна соответствовать следующему рекуррентному выражению:

$$\text{OptCost}(i, k) = \begin{cases} 0 & , \text{ если } i > k, \\ \sum_{j=i}^k f[j] + \min_{i \leq r \leq k} \left\{ \begin{array}{l} \text{OptCost}(i, r-1) \\ + \text{OptCost}(r+1, k) \end{array} \right\} & \text{ иначе.} \end{cases}$$

В базовом варианте корректно показано, что минимальная возможная стоимость для выполнения нуля операций поиска в пустом множестве равна нулю. А исходная задача заключается в вычислении $\text{OptCost}(1, n)$.

Это рекурсивное определение можно механически преобразовать в рекурсивный алгоритм поиска с возвратом для вычисления $\text{OptCost}(1, n)$. Неудивительно, что время выполнения этого алгоритма является экспоненциальным. В следующей главе мы увидим, как можно уменьшить время выполнения до полиномиального, поэтому здесь нет особого смысла вычислять точное время выполнения...

♥ Анализ

...если вы не специалист по этой части. Просто в качестве очередного небольшого упражнения определим, насколько медленно выполняется этот алгоритм поиска с возвратом в действительности. Время выполнения определяется следующим рекуррентным выражением:

$$T(n) = \sum_{k=1}^n (T(k-1) + T(n-k)) + O(n).$$

Член $O(n)$ взят из вычисления общего количества операций поиска $\sum_{i=1}^n f[i]$. Да, это некрасивое рекуррентное выражение, но его можно решить, используя тот же самый прием с вычитанием, который применялся выше. Заменим форму записи $O(\)$ на явное постоянное значение, перегруппируем и объединим одинаковые члены, вычтем из рекуррентного выражения $T(n-1)$, чтобы исключить суммирование, затем еще раз перегруппируем члены полученного выражения.

$$T(n) = 2 \sum_{k=0}^{n-1} T(k) + \alpha n$$

$$T(n-1) = 2 \sum_{k=0}^{n-2} T(k) + \alpha(n-1)$$

$$T(n) - T(n-1) = 2T(n-1) + \alpha$$

$$T(n) = 3T(n-1) + \alpha.$$

После всех выполненных преобразований выражение уже не выглядит так страшно. Метод рекурсивного дерева сразу же дает решение $T(n) = O(3^n)$ (его можно было предположить и подтвердить методом индукции).

Этот анализ полагает, что рассматриваемый здесь рекурсивный алгоритм не исследует все возможные двоичные деревья поиска. Количество двоичных деревьев поиска с n вершинами определяется рекуррентным выражением

$$N(n) = \sum_{r=1}^{n-1} (N(r-1) \cdot N(n-r)),$$

которое имеет аналитическое решение $N(n) = \Theta(4^n/\sqrt{n})$. (Но это не очевидно.) Рассматриваемый здесь алгоритм позволяет сэкономить значительное время, выполняя поиск независимо в оптимальном правом и левом поддереве для каждого корня. Полное перечисление двоичных деревьев поиска должно рассматривать все возможные пары левых и правых поддеревьев, из чего и следует произведение в рекуррентном выражении для $N(n)$.

Упражнения

- Описать рекурсивные алгоритмы для следующих обобщенных вариантов задачи *SubsetSum*:
 - задан массив положительных целых чисел $X[1..n]$ и целое число T , вычислить количество подмножеств X , сумма элементов которых равна T ;
 - заданы два массива положительных целых чисел $X[1..n]$ и $W[1..n]$ и целое число T , при этом $W[i]$ обозначает вес (весовой коэффициент) соответствующего элемента $X[i]$, вычислить максимальный вес подмножества X , сумма элементов которого равна T . Если не существует подмножества X , сумма элементов которого равна T , то алгоритм должен возвращать $-\infty$.
- Описать рекурсивные алгоритмы для следующих вариантов задачи сегментации текста. Предполагается, что в вашем распоряжении

имеется подпрограмма `IsWord()`, в качестве входных данных принимающая массив символов и возвращающая `True`, если принятая строка является «словом».

- (а) Задан массив символов $A[1..n]$, вычислить количество вариантов разделения A на слова. Например, если задана строка **ARTISTOIL**, то алгоритм должен вернуть 2 в соответствии с вариантами разделения **ARTIST · OIL** и **ART · IS · TOIL**.
- (б) Заданы два массива символов $A[1..n]$ и $B[1..n]$, определить, можно ли разделить A и B на слова по одинаковым индексам. Например, строки **BOTHEARTHANDSATURNSPIN** и **PINSTARTRAPSANDRAGSLAP** можно разделить на слова по одинаковым индексам, как показано ниже:

BOT · HEART · HAND · SAT · URNS · PIN
PIN · START · RAPS · AND · RAGS · LAP.

- (с) Заданы два массива символов $A[1..n]$ и $B[1..n]$, вычислить количество различных способов, которыми A и B можно разделить на слова по одинаковым индексам.
3. Аддитивная цепочка для целого числа n – это возрастающая последовательность целых чисел, начинающаяся с 1 и заканчивающаяся n , такая, что каждый элемент после первого является суммой двух предшествующих элементов. Более формальное определение: последовательность целых чисел $x_0 < x_1 < x_2 < \dots < x_\ell$ является аддитивной цепочкой для числа n , если и только если:

- $x_0 = 1$;
- $x_\ell = n$;
- для каждого индекса $k > 0$ существуют индексы $i \leq j < k$ такие, что $x_k = x_i + x_j$.

Длина ℓ аддитивной цепочки равна количеству элементов – 1, так как первый элемент в подсчет не включается. Например, последовательность $\{1, 2, 3, 5, 10, 20, 23, 46, 92, 184, 187, 374\}$ является аддитивной цепочкой для числа 374, а ее длина равна 11.

- (а) Описать рекурсивный алгоритм поиска с возвратом для вычисления аддитивной цепочки минимальной длины для заданного положительного целого числа n . Не анализируйте и не оптимизируйте время выполнения предложенного вами алгоритма, если только вы не пожелаете удовлетворить собственное любопытство. Корректный алгоритм с экспоненциальным временем выполнения относительно n вполне достаточен для получения высшей оценки. (Подсказка: эта задача в большей степени похожа на задачу о расстановке n ферзей, чем на задачу сегментации текста.)

- ♥(b) Описать рекурсивный алгоритм поиска с возвратом для вычисления аддитивной цепочки минимальной длины для заданного положительного целого числа n за субэкспоненциальное время относительно n . (Подсказка: результаты некоторых египетских «вязателей веревок», мастеров стихосложения с реки Инд и русских крестьян могут оказаться полезными.)
4. (a) Пусть $A[1..m]$ и $B[1..n]$ – два произвольных массива. Общая подпоследовательность A и B – это подпоследовательность, которая содержится и в A , и в B . Сформулировать простое рекурсивное определение функции $lcs(A, B)$, возвращающей длину максимальной общей подпоследовательности массивов A и B .
- (b) Пусть $A[1..m]$ и $B[1..n]$ – два произвольных массива. Общей надпоследовательностью A и B является другая последовательность, содержащая A и B как подпоследовательности. Сформулировать простое рекурсивное определение функции $scs(A, B)$, возвращающей длину минимальной общей надпоследовательности массивов A и B .
- (c) Последовательность чисел $X[1..n]$ называют битонической (*bitonic*), если существует индекс i при $1 < i < n$ такой, что префикс $X[1..i]$ возрастающий, а суффикс $X[i..n]$ убывающий. Сформулировать простое рекурсивное определение функции $lbs(A)$, возвращающей длину максимальной битонической подпоследовательности в произвольном массиве целых чисел A .
- (d) Последовательность чисел $X[1..n]$ называют колебательной (*oscillating*), если $X[i] < X[i + 1]$ для всех четных i и $X[i] > X[i + 1]$ для всех нечетных i . Сформулировать простое рекурсивное определение функции $los(A)$, возвращающей длину максимальной колебательной подпоследовательности в произвольном массиве целых чисел A .
- (e) Сформулировать простое рекурсивное определение функции $sos(A)$, возвращающей длину минимальной колебательной подпоследовательности в произвольном массиве целых чисел A .
- (f) Последовательность чисел $X[1..n]$ называют выпуклой (*convex*), если $2 \cdot X[i] < X[i - 1] + X[i + 1]$ для всех i . Сформулировать простое рекурсивное определение функции $lxs(A)$, возвращающей длину максимальной выпуклой подпоследовательности в произвольном массиве целых чисел A .
5. Для каждой из приведенных ниже задач входные данные состоят из двух массивов $X[1..k]$ и $Y[1..n]$, при этом $k \leq n$.
- (a) Описать рекурсивный алгоритм поиска с возвратом, определяющий, является ли X подпоследовательностью Y . Например, строка **PPAP** является подпоследовательностью строки **PENPINEAPPLEPEN**.

(b) Описать рекурсивный алгоритм поиска с возвратом для вычисления минимального количества символов, которые можно удалить из последовательности Y , чтобы X уже не являлась подпоследовательностью Y . Кроме того, алгоритм должен находить максимальную подпоследовательность в Y , которая не является надпоследовательностью X . Например, после удаления двух выделенных символов из строки **PENPINEAPPLEPEN** строка **PPAP** уже не является ее подпоследовательностью.

♥(c) Описать рекурсивный алгоритм поиска с возвратом, определяющий, встречается ли X как две непересекающиеся (не имеющие общих символов) подпоследовательности в Y . Например, строка **PPAP** встречается как две непересекающиеся подпоследовательности в строке **PENPINEAPPLEPEN**.

Не анализируйте время выполнения предложенных вами алгоритмов, если только вы не пожелаете удовлетворить собственное любопытство. Все три алгоритма выполняются за экспоненциальное время. Мы улучшим эти алгоритмы несколько позже, поэтому точное время их выполнения сейчас не имеет особого значения.

6. В этой задаче вам предлагается спроектировать алгоритмы поиска с возвратом для вычисления стоимости оптимального двоичного дерева поиска с учетом дополнительных ограничений по их балансу. Входные данные состоят из отсортированного массива ключей поиска $A[1..n]$ и массива счетчиков частот $f[1..n]$, где $f[i]$ – число операций поиска для ключа $A[i]$. Это точно такая же функция стоимости, как и описанная в разделе 2.8. Но теперь ваша задача – вычислить оптимальное дерево с учетом некоторых дополнительных ограничений.

а) **АВЛ-дерево** – самое первое сбалансированное двоичное дерево поиска с самобалансировкой – было впервые описано в 1962 г. Георгием Адельсон-Вельским и Евгением Ландисом. АВЛ-дерево – это двоичное дерево поиска, в котором для каждого узла v высота его левого и правого поддеревьев различается не более чем на единицу.

Описать рекурсивный алгоритм поиска с возвратом для создания оптимального АВЛ-дерева для заданного множества ключей поиска и соответствующих частот.

б) **Симметричные двоичные В-деревья** – другой вид самобалансирующихся двоичных деревьев, впервые описанный Рудольфом Байером (Rudolf Bayer) в 1972 г. Они стали более известны как **красно-черные деревья** (*red-black trees*) после некоторой упрощенной новой формулировки, предложенной Леонидасом Гимбасом (Leonidas Guibas) и Робертом Седжвиком (Robert

Sedgwick) в 1978 г. Красно-черное дерево – это двоичное дерево поиска со следующими дополнительными ограничениями:

- каждый узел является либо красным, либо черным;
- каждый красный узел имеет черного родителя;
- каждый путь от корня к листу содержит одинаковое количество черных узлов.

Описать рекурсивный алгоритм поиска с возвратом для создания оптимального красно-черного дерева для заданного множества ключей поиска и соответствующих частот.

- с) **АА-деревья** были предложены Арне Андерссоном (Arne Andersson) в 1993 г. и немного упрощены (и поименованы) Марком Алленом Вайсом (Mark Allen Weiss) в 2000 г. АА-деревья также известны как *красно-черные деревья с уклоном влево* (*left-leaning red-black trees*) после симметричного переформулирования (с различными алгоритмами изменения баланса) Робертом Седжвиком в 2006 г. АА-дерево – это красно-черное дерево с единственным дополнительным ограничением:

- левый узел-потомок не должен быть красным¹⁴.

Описать рекурсивный алгоритм поиска с возвратом для создания оптимального АА-дерева для заданного множества ключей поиска и соответствующих частот.

Не анализируйте время выполнения предложенных вами алгоритмов, если только вы не пожелаете удовлетворить собственное любопытство. Все три алгоритма выполняются за экспоненциальное время. Мы улучшим эти алгоритмы несколько позже, поэтому точное время их выполнения сейчас не имеет особого значения.

В следующей главе вы найдете еще больше примеров алгоритмов поиска с возвратом.

¹⁴ Переформулировка Седжвика требовала, чтобы правый узел-потомок не являлся красным. Никакой разницы. Странно, что Андерссон и Седжвик ничего не говорят о том, с какого конца нужно разбивать и есть яйцо.

Глава 3

Динамическое программирование

Вы можете видеть здесь, на полях, как мы работали с этими числами; ясно, что мы объединили первое число со вторым, а именно: 1 с 2, второе с третьим, третье с четвертым, четвертое с пятым и так далее...

— Леонардо Пизанский (Фибоначчи) (Leonardo Pisano), «Liber Abaci» (1202 г.)

Те, кто не способны помнить о прошлом, обречены его повторять.

— Хорхе Агустин Николас Руис де Сантьяна и Боррас
(Jorge Agustín Nicolás Ruiz de Santayana y Borrás), «Жизнь разума» («The Life of Reason»),
«Книга I: Введение и Разум в общем смысле»
(«Introduction and Reason in Common Sense») (1905 г.)

Вы знаете, что такое обучение на собственном опыте?

Обучение на собственном опыте – это одна из тех штук, которые говорят нам:

«Вы осознаете, что вы только что сделали? Не делайте этого (больше)».

— Дуглас Адамс (Douglas Adams), «Лосось сомнения» («The Salmon of Doubt») (2002 г.)

3.1. Mātrāvṛtta

Один из самых ранних примеров применения рекурсии появился в Индии более 2000 лет назад в науке о поэтической метрике, или просодии (prosody). В классической санскритской поэзии различаются два типа слогов (*akṣara*): легкий (*laghu*) и тяжелый (*guru*). В одном классе метрик, называемом либо *mātrāvṛtta*, либо *mātrāchandas*, каждая строка стихотворения состоит из строго ограниченного количества «тактов» (beats; *mātrā*), где длительность каждого легкого слога составляет один такт, а длительность каждого тяжелого слога – два такта. Формальная наука *mātrā-vṛtta* ведет свое начало от трактата «*Chandaḥśāstra*», написанного ученым Пингала (Piṅgala) между 600 и 200 гг. до н. э. Пингала заметил, что существует ровно пять четырехтактовых метрик: — —, — • •, • — •, • • — и • • • •. (Здесь каждый

символ «—» представляет длинный слог, а каждый символ • обозначает короткий слог.)¹

Несмотря на то что Пингала в своем трактате советует следовать систематическому правилу рассчитывать метрику с использованием определенного числа тактов,² потребовалось около тысячи лет, чтобы это правило было высказано в явной форме. В VII в. н. э. другой индийский ученый Вираханка (Virahanka) написал комментарии к работе Пингалы, в которых указал, что число метрик с n тактами является суммой числа метрик с $(n - 2)$ тактами и числа метрик с $(n - 1)$ тактами. В более современной форме записи наблюдение Вираханки представляет собой рекуррентное выражение для общего числа $M(n)$ n -тактовых метрик:

$$M(n) = M(n - 2) + M(n - 1).$$

Легко заметить, что $M(0) = 1$ (существует только одна пустая метрика) и $M(1) = 1$ (единственная однотоковая метрика состоит из одного короткого слога).

То же самое рекуррентное выражение было независимо выведено в Европе приблизительно через 500 лет после комментария Вираханки Леонардо Пизанским (Фибоначчи) в трактате «Liber Abaci» («Книга вычислений», 1202 г.), одной из наиболее значимых раннеевропейских работ по «алгоритмам». В полном соответствии с законом Стиглера об эпонимии³ современный ряд чисел Фибоначчи определяется с использованием рекуррентной формулы Вираханки, но с другими базовыми вариантами:

$$F_n = \begin{cases} 0 & , \text{ если } n = 0; \\ 1 & , \text{ если } n = 1; \\ F_{n-1} + F_{n-2} & \text{ иначе.} \end{cases}$$

В частности, получаем $M(n) = F_{n+1}$ для всех n .

¹ В коде азбуки Морзе продолжительность тире (dah) в три раза больше продолжительности точки (dit), но за каждой точкой или тире следует пауза, длительность которой та же самая, что и у точки. Таким образом, каждая «точка-пауза» — это *laghu akṣara* (легкий слог), а каждая «тире-пауза» — это *guru akṣara* (тяжелый слог), и существует ровно пять букв (M, D, R, U, H), длительность кодов которых равна четырем *mātrā* (тактам).

² Трактат «*Chandaḥśāstra*» содержит два систематические правила для перечисления всех метрик с использованием определенного числа слогов, что приблизительно соответствует записи чисел в двоичной форме слева направо (как это делали греки) или справа налево (как это делали египтяне). В это трактат также включен рекурсивный алгоритм вычисления 2^n (числа метрик с n слогами) с использованием повторяющегося возведения в квадрат и (вероятно) рекурсивный алгоритм вычисления биномиальных коэффициентов (числа метрик с k короткими слогами и n слогами в целом).

³ «Никакое научное открытие не было названо в честь первооткрывателя». В своей работе 1980 г., которая дала этому закону его имя, профессор статистики Стивен Стиглер (Stephen Stigler) в шутку заявил, что этот закон первым предложил социолог Роберт Мертон (Robert K. Merton). Но похожие мысли еще раньше высказывались Владимиром Арнольдом в 1970-х гг. («Открытия редко приписываются истинному первооткрывателю»), Карлом Бойером (Carl Boyer) в 1968 г. («Клио, муза истории, часто ненадежна в присваивании имен теоремам»), Альфредом Нортум Уайтхедом (Alfred North Whitehead) в 1917 г. («Все важное уже было сказано кем-то, кто этого не обнаруживал»). И даже отец Стивена Джордж Стиглер (George Stigler) в 1966 г. говорил: «Если мы вдруг обнаружим случай, когда теория названа по имени ее истинного автора, это будет заслуживать особого внимания». В этой книге мы еще увидим множество других примеров, подтверждающих закон Стиглера.

Алгоритм поиска с возвратом может быть медленным

Рекурсивное определение ряда чисел Фибоначчи естественным образом приводит нас к рекурсивному алгоритму их вычисления. Ниже показан этот алгоритм, записанный на псевдокоде.

```
RecFibo(n):
  if n = 0
    return 0
  else if n = 1
    return 1
  else
    return RecFibo(n - 1) + RecFibo(n - 2)
```

К сожалению, этот примитивный рекурсивный алгоритм выполняется ужасающе медленно. Если исключить рекурсивные вызовы, то алгоритм в целом требует только лишь постоянного количества шагов: одна операция сравнения и, возможно, одна операция сложения. Пусть $T(n)$ обозначает число рекурсивных вызовов RecFibo, тогда эта функция определяется рекуррентным выражением

$$T(0) = 1, T(1) = 1, T(n) = T(n - 1) + T(n - 2) + 1,$$

которое выглядит так же отталкивающе, как и рекуррентное выражение для самих чисел Фибоначчи. Запись нескольких первых значений $T(n)$ наводит на мысль о существовании аналитического решения $T(n) = 2F_{n+1} - 1$, которое можно проверить методом индукции с аргументом (hint, hint). Поэтому вычисление F_n с использованием этого алгоритма отнимет приблизительно вдвое больше времени, чем простой счет до F_n . Методы, не относящиеся к теме этой книги⁴, подразумевают, что $F_n = \Theta(\phi^n)$, где $\phi = (\sqrt{5} + 1)/2 \approx 1.61803$ – так называемое золотое сечение. Короче говоря, время выполнения этого рекурсивного алгоритма является экспоненциальным со степенью n .

Это экспоненциальное возрастание можно наблюдать наглядно, как показано ниже. Мысленно представим рекурсивное дерево для RecFibo как двоичное дерево операций сложения, листьями которого являются только нули и единицы. Поскольку конечный результат (вывод) – это F_n , ровно F_n листьев непременно должны иметь значение 1, эти листья представляют вызовы RecFibo(1). Простой индуктивный аргумент (hint, hint) подразумевает, что RecFibo(0) вызывается ровно F_{n-1} раз. (Если нам вдруг потребуется асимптотическая граница, то достаточно заметить, что число вызовов RecFibo(0) не превосходит числа вызовов RecFibo(1).) Таким образом, рекурсивное дерево имеет ровно $F_n + F_{n-1} = F_{n+1} = O(F_n)$ листьев, следовательно, поскольку это полное двоичное дерево, в целом оно содержит $2F_{n+1} - 1 = O(F_n)$ узлов.

⁴ Конспекты по решению рекуррентных выражений поиска с возвратом см. здесь: <http://algorithms.wtf>.

Мемоизация (запоминание): помнить все

Явной причиной медленной работы приведенного выше рекурсивного алгоритма является тот факт, что он вычисляет одни и те же числа Фибоначчи снова и снова. Единственный вызов $\text{RecFibo}(n)$ приводит к одному рекурсивному вызову $\text{RecFibo}(n - 1)$, двум рекурсивным вызовам $\text{RecFibo}(n - 2)$, трем рекурсивным вызовам $\text{RecFibo}(n - 3)$, пяти рекурсивным вызовам $\text{RecFibo}(n - 4)$, а в общем к F_{k-1} рекурсивным вызовам $\text{RecFibo}(n - k)$ для любого целого $0 \leq k < n$. При каждом вызове некоторое число Фибоначчи вычисляется заново.

Можно существенно ускорить этот рекурсивный алгоритм, записывая результаты выполненных рекурсивных вызовов и в дальнейшем обращаться к ним при необходимости.

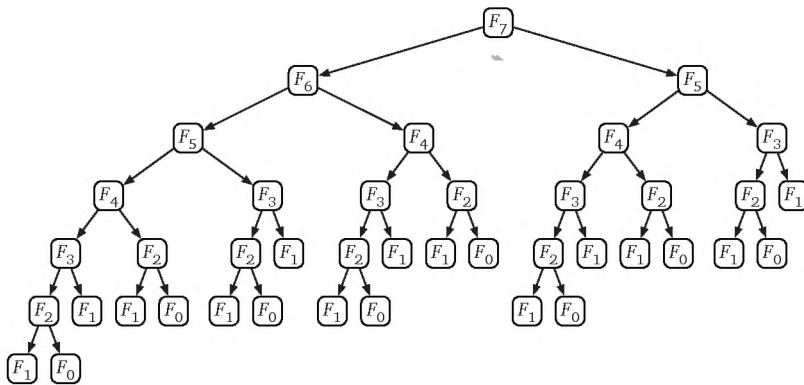


Рис. 3.1. Рекурсивное дерево для вычисления F_7 , стрелки представляют рекурсивные вызовы

Создание этой методики оптимизации, известной в настоящее время как мемоизация (memoization – именно так, без буквы R), приписывается Дональду Мичи (Donald Michie) в 1967 г., но точно такую же методику предложил Артур Самуэл (Arthur Samuel) в 1959 г.⁵

Мемоизация явно уменьшает время выполнения этого алгоритма, но насколько он становится быстрее? Если действительно выполнить трассировку рекурсивных вызовов, выполняемых в MemFibo , то обнаружится, что массив $F[]$ заполняется снизу вверх: сначала $F[2]$, потом $F[3]$ и т. д. до $F[n]$. Эту

⁵ Мичи предлагал следующее: языки программирования должны поддерживать абстракцию, которую он называл «memo function» (функцией запоминания), состоящую из стандартной функции (rule) и словаря (rote) вместо раздельной поддержки массивов и функций. Когда мемо-функция вычисляет свое значение в первый раз, она «запоминает» (memorises – именно так, с буквой R) это значение в своем словаре. Стимулом для Мичи послужило использование Самуэлом методики «rote learning» (обучение с механическим запоминанием) для рекурсивной обработки деревьев игры в шашки. Мичи описал свое более обобщенное предложение как «обеспечение для программиста возможности «самуэлизировать» (Samuelize) любую функцию, которую он пожелает». (Насколько я знаю, сам Мичи никогда не использовал термин «мемоизация».) Мемоизация использовалась даже еще раньше Клодом Шенноном (Claude Shannon) для решения задачи поиска пути в лабиринте роботом Theseus (Тесей), спроектированным и сконструированным в 1950 г.

схему можно проверить методом индукции: каждый элемент $F[i]$ заполняется только после заполнения предшествующего ему элемента $F[i - 1]$. Если не учитывать время, затраченное на рекурсивные вызовы, то потребуется всего лишь постоянное время для вычисления рекуррентного выражения для каждого числа Фибоначчи F_i . Но по определению рекуррентное выражение для F_i вычисляется только один раз для каждого индекса i . Мы приходим к выводу, что MemFibo выполняет лишь $O(n)$ операций сложения, – это экспоненциальное улучшение, по сравнению с примитивным рекурсивным алгоритмом.

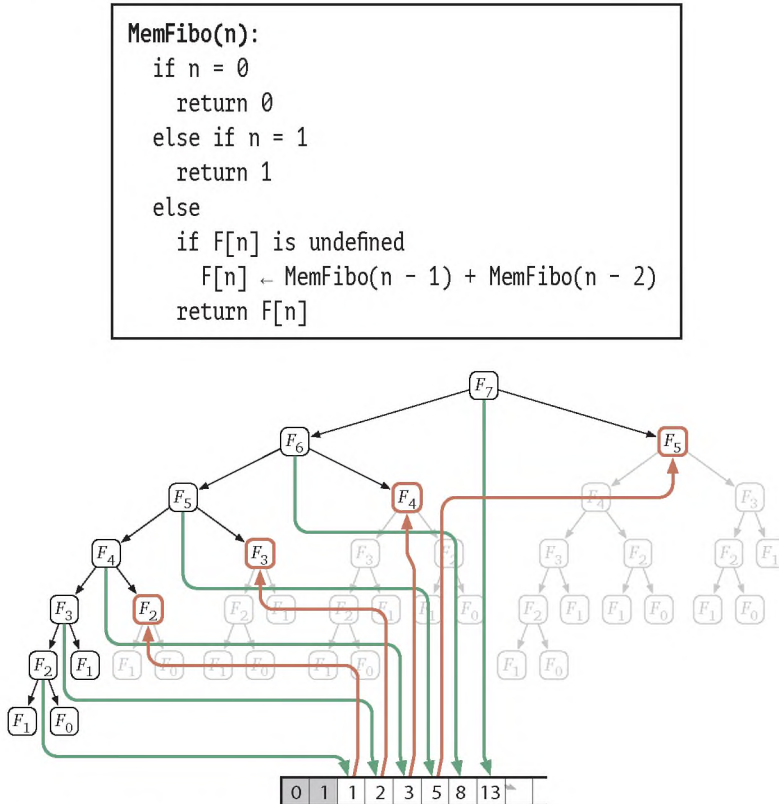


Рис. 3.2. Рекурсивное дерево для вычисления F_7 , обработанное методом мемоизации. Зеленые стрелки, направленные вниз, обозначают запись в массив мемоизации, красные стрелки, направленные вверх, обозначают чтение из массива мемоизации

Динамическое программирование: осмысленное заполнение

Мы наблюдали, как заполняется массив $F[]$, и теперь можем заменить мемоизованное рекуррентное выражение на простой цикл `for`, который преднамеренно заполняет этот массив в том же порядке, и исключить бо-

лее сложный рекурсивный алгоритм, случайным образом выполняющий операции записи.

```
IterFibo(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
    F[i] ← F[i - 1] + F[i - 2]
  return F[n]
```

Теперь анализ времени выполнения предельно прост: IterFibo явно использует $O(n)$ операций сложения и сохраняет $O(n)$ целых чисел.

Это наш первый явный алгоритм динамического программирования. Парадигма (концептуальная модель) динамического программирования (dynamic programming) была формализована и распространялась Ричардом Беллманом (Richard Bellman) в середине 1950-х гг., когда он работал в военном исследовательском центре RAND Corporation, хотя Беллман далеко не первым использовал эту методику. В частности, этот итеративный алгоритм для чисел Фибоначчи был предложен Вираханкой и более поздними санскритскими поэтами-просодистами еще в XII в., а потом уже Фибоначчи в XIII в.⁶

Много лет спустя Беллман признался, что умышленно выбрал термин «динамическое программирование», чтобы скрыть математическую сущность своей работы от своих начальников-военных, которые вели «активные боевые действия» в отношении всего, что в той или иной степени относилось к математическим исследованиям.⁷ Здесь слово «программирование» не означает «написание кода», а употребляется в более старом смысле, т. е.

⁶ Более обобщенные методики динамического программирования разрабатывались несколько раз независимо друг от друга в конце 1930-х и в начале 1940-х гг. Например, Пьер Массе (Pierre Massé) использовал алгоритмы динамического программирования для оптимизации работы гидроэлектростанций во Франции при режиме Виши. Джон фон Нейман (John von Neumann) и Оскар Моргенштерн (Oskar Morgenstern) разработали алгоритмы динамического программирования для определения победителя в любой игре с двумя игроками и с абсолютно полной информацией (например, шашки). Алан Тьюринг (Alan Turing) и его коллеги использовали аналогичные методы как часть их работы по взлому кодов в Блетчли-парке. Работы Массе и фон Неймана и Моргенштерна были впервые опубликованы в 1944 г., за шесть лет до того, как Беллман употребил термин «динамическое программирование». Подробности метода Banburismus Тьюринга хранились в секрете до середины 1980-х гг.

⁷ Чарльз Эрвин Уилсон (Charles Erwin Wilson) стал министром обороны в январе 1953 г. после десяти лет президентства в General Motors. «Чарли-мотор» (Engine Charlie) реорганизовал министерство обороны и существенно сократил его бюджет в первый же год в этой должности с явной целью максимально приблизить стиль работы министерства к стилю работы промышленной корпорации. Беллман описывает Уилсона в своей автобиографии (1984 г.) так: «В то время в Вашингтоне мы наблюдали весьма любопытного джентльмена по имени Уилсон. Он был министром обороны, но при этом паталогически боялся слова “исследование” и ненавидел его. Я использую этот термин не бездумно, я имею в виду его абсолютно точное значение. Лицо Уилсона багровело, и он приходил в ярость, если кто-то использовал слово “исследование” в его присутствии. Теперь можете себе представить, что он чувствовал, услышав слово “математический”... Поэтому я чувствовал, что надо как-то оградить Уилсона и ВВС от осознания того факта, что я действительно занимался математикой в RAND Corporation. Так какое же название, какое имя я мог бы выбрать?» Но Беллман открыто использовал термин «динамическое программирование», появившийся еще в 1952 г., за несколько месяцев до вступления Уилсона в должность министра обороны, так что этот рассказ как минимум слегка приукрашен.

как «планирование» (planning, scheduling) – обычно методом заполнения таблицы. Например, в программах спортивных соревнований и театраль-ных программах планируются важные события (с рекламными объяв-лениями), телевизионные программы заполняют все доступные интервалы времени телешоу (и рекламными роликами), образовательные программы планируют учебные курсы (с рекламой). Группа Air Force основана Беллманом и др. для разработки методов формирования планов обучения и логи-стики (снабжения), или, как они называли эти планы, «программ». Слово «динамическое» означало не только многоэтапность и гибкость времен-ных характеристик процессов, которые Беллман и его коллеги пытались оптимизировать, но также являлось модным маркетинговым термином, перекликающимся с Futuristic Can-Do Zeitgeist™ в послевоенной Амери-ке конца 1940-х гг.⁸ Отчасти благодаря усилиям Беллмана по обращению в новую веру динамическое программирование в настоящее время является стандартным инструментом многоэтапного планирования в экономике, робототехнике, теории управления и в некоторых других областях.

И все же не следует запоминать все подряд

Во многих алгоритмах динамического программирования нет необхо-димости сохранять все промежуточные результаты, получаемые в процес-се вычислений. Например, мы можем значительно снизить требования к объему памяти в алгоритме IterFibo, сохраняя только два самых свежих элемента массива:

```
IterFibo2(n):
  prev ← 1
  curr ← 0
  for i ← 1 to n
    next ← curr + prev
    prev ← curr
    curr ← next
  return curr
```

(Этот алгоритм использует нестандартный, но логически обоснованный базовый случай $F_{-1} = 1$, таким образом, IterFibo2(0) возвращает правильное значение 0.) Хотя экономия памяти может оказаться чрезвычайно важной на практике, в данной книге мы не будем уделять внимание этой теме.

♥ 3.2. Небольшое отступление: еще более быстрое определение чисел Фибоначчи

Несмотря на то что последний алгоритм из предыдущего раздела прост и вполне приемлем, он не является самым быстрым алгоритмом определе-

⁸ ...и, возможно, с привлекательным названием легендарного бренда Dynamic-Tension (Динами-ческое растяжение) для широко известного комплекса упражнений Чарльза Атласа, который Чарльз Роман разработал в 1928 г. Герой пляжей!

ния чисел Фибоначчи. Можно получить более быстрый алгоритм, воспользовавшись следующим матричным представлением рекуррентной последовательности Фибоначчи:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ x+y \end{bmatrix}.$$

Другими словами, умножение двумерного вектора на матрицу $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ дает точно такой же результат, как одна итерация внутреннего цикла в алгоритме IterFibo2. Отсюда следует, что умножение на эту матрицу n раз равнозначно выполнению n итераций внутреннего цикла:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix}.$$

Таким образом, если требуется n -е число Фибоначчи, необходимо лишь вычислить n -ю степень матрицы $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$. Если использовать многократно повторяющееся возведение в квадрат⁹, для вычисления n -й степени какого-либо объекта потребуется всего лишь $O(\log n)$ операций умножения. В рассматриваемом здесь случае, поскольку «каким-либо объектом» является матрица 2×2 , это означает $O(\log n)$ операций умножения таких матриц, а каждая операция умножения сводится к постоянному количеству операций целочисленного умножения и сложения. Следовательно, можно вычислить F_n всего лишь за $O(\log n)$ целочисленных арифметических операций.

Такого же повышения скорости можно достичь, используя тождество $F_n = F_m F_{n-m-1} + F_{m+1} F_{n-m}$, которое верно (по индукции) для всех целых чисел m и n . В частности, это тождество позволяет вывести следующее взаимно рекуррентное выражение для пар соседних чисел Фибоначчи, впервые предложенное Эдуардом Люка (Édouard Lucas) в 1898 г.:

$$\begin{aligned} F_{2n-1} &= F_{n-1}^2 + F_n^2 \\ F_{2n} &= F_n(F_{n-1} + F_{n+1}) = F_n(2F_{n-1} + F_n). \end{aligned}$$

(Это взаимно рекуррентное выражение также можно вывести непосредственно из алгоритма возведения матриц в квадрат.) Показанные выше рекуррентные формулы легко преобразовать в следующий алгоритм:

```

<<Вычисление пары  $F_{n-1}, F_n$ >>
FastRecFibo(n):
    if n = 1
        return 0, 1
    m ← ⌊n/2⌋
    hprv, hcur ← FastRecFibo(m)    << $F_{m-1}, F_m$ >>
    prev ← hprv2 + hcur2        << $F_{2m-1}$ >>

```

⁹ Как предлагает Пингала для вычисления степеней 2 в другой части трактата «Chandaḥśāstra».

<code>curr ← hcur · (2 · hprv + hcur)</code>	$\langle\langle F_{2m} \rangle\rangle$
<code>next ← prev + curr</code>	$\langle\langle F_{2m+1} \rangle\rangle$
<code>if n is even</code>	
<code>return prev, curr</code>	
<code>else</code>	
<code>return curr, next</code>	

Методика стандартного рекурсивного дерева позволяет определить, что этот алгоритм выполняет всего лишь $O(\log n)$ операций целочисленной арифметики.

Это экспоненциальное улучшение скорости по сравнению со стандартным итеративным алгоритмом, который сам дает экспоненциальное улучшение скорости, по сравнению с исходным рекурсивным алгоритмом, не так ли?

Стоп! Не так быстро

Ну, не совсем. Ряд чисел Фибоначчи возрастает с экспоненциальной скоростью. N -е число Фибоначчи содержит приблизительно $n \log_{10} \phi \approx n/5$ десятичных цифр или $n \log_2 \phi \approx 2n/3$ бит. Поэтому, вероятно, мы не можем вычислить F_n за логарифмическое время – необходимо время $\Omega(n)$ только для того, чтобы записать результат.

Результатом этого явного парадокса является следующее наблюдение: невозможно выполнить арифметическую операцию с произвольной точностью за постоянное время. Пусть $M(n)$ обозначает время, требуемое для умножения двух n -разрядных чисел. Время выполнения алгоритма FastRecFibo определяется рекуррентным выражением $T(n) = T(\lfloor n/2 \rfloor) + M(n)$, решением которого с помощью рекурсивного дерева является $T(n) = O(M(n))$. Самый быстрый известный (на 2019 г.) алгоритм целочисленного умножения выполняется за время $O(n \log n)$, т. е. это также время выполнения самого быстрого известного (на 2019 г.) алгоритма определения чисел Фибоначчи.

Этот алгоритм медленнее, чем ранее рассмотренные итеративные алгоритмы «с линейным временем выполнения»? На самом деле нет, потому что для операции сложения тоже необходимо некоторое время. Для сложения двух n -разрядных чисел требуется время $O(n)$, так что итеративные алгоритмы IterFibo и IterFibo2 в действительности выполняются за время $O(n^2)$. (Вы понимаете почему?) Поэтому FastRecFibo значительно быстрее, чем итеративные алгоритмы, но не экспоненциально быстрее.

В исходном рекурсивном алгоритме дополнительная стоимость операций арифметики с произвольной точностью становится не столь существенной, по сравнению с огромным количеством рекурсивных вызовов. Правильным рекуррентным выражением является $T(n) = T(n-1) + T(n-2) + O(n)$, решением которого остается $T(n) = O(\phi^n)$.

3.3. Interpunctio verborum redux (И снова о пунктуации)

Для следующего алгоритма динамического программирования рассмотрим задачу сегментации текста из предыдущей главы. Дана строка $A[1..n]$ и подпрограмма $IsWord$, определяющая, является ли заданная строка словом (что бы это ни значило). Необходимо узнать, можно ли разделить строку A на последовательность слов.

Мы решили эту задачу, определив функцию $Splittable(i)$, которая возвращает $True$, если и только если суффикс $A[i..n]$ можно разделить на последовательность слов. Необходимо вычислить базовый случай $Splittable(1)$. Функция определяется следующим рекуррентным выражением:

$$Splittable(i) = \begin{cases} True & , \text{ если } i > n; \\ \bigvee_{j=i}^n (IsWord(i, j) \wedge Splittable(j + 1)) & \text{ иначе,} \end{cases}$$

где $IsWord(i, j)$ – это сокращенная форма $IsWord(A[i..j])$. Это рекуррентное выражение преобразуется непосредственно в рекурсивный алгоритм поиска с возвратом, вызывающий подпрограмму $IsWord$ $O(2^n)$ раз в наихудшем случае.

Но для любой строки $A[1..n]$ с конкретным конечным размером существует только лишь n различных способов вызова рекурсивной функции $Splittable(i)$ – по одному для каждого значения i между 1 и $n + 1$ – и всего лишь $O(n^2)$ различных способов вызова подпрограммы $IsWord(i, j)$ – по одному для каждой пары (i, j) , такой что $1 \leq i \leq j \leq n$. Так почему же затрачивается экспоненциальное время на вычисление всего лишь полиномиального количества исходной сущности?

```
FastSplittable(A[1..n]):
    SplitTable[n + 1] ← True
    for i ← n down to 1
        SplitTable[i] ← False
        for j ← i to n
            if IsWord(i, j) and SplitTable[j + 1]
                SplitTable[i] ← True
    return SplitTable[1]
```

Рис. 3.3. Быстрая сегментация текста

Каждая рекурсивная подзадача определяется целым числом от 1 до $n + 1$, поэтому можно выполнить мемоизацию функции $Splittable$ в массив $SplitTable[1..n + 1]$. Каждая подзадача $Splittable(i)$ зависит только от результатов подзадач $Splittable(j)$, где $j > i$, так что мемоизованный рекурсивный

алгоритм заполняет массив в порядке убывания индексов. Если преднамеренно заполнить массив в таком порядке, то получим алгоритм динамического программирования, показанный на рис. 3.3. Алгоритм выполняет $O(n^2)$ вызовов `IsWord` – это экспоненциальное улучшение, по сравнению с ранее рассмотренным алгоритмом поиска с возвратом.

3.4. Шаблон: интеллектуальная рекурсия

Если говорить кратко, то динамическое программирование – это рекурсия без повторений. Алгоритмы динамического программирования сохраняют решения промежуточных подзадач часто, но не всегда в массиве некоторого типа или таблице. Многие из тех, кто изучает алгоритмы (а также преподаватели и книги по алгоритмам), совершают ошибку, сосредоточивая основное внимание на таблицах (поскольку таблицы просты и хорошо знакомы) вместо гораздо более важной (и сложной) задачи поиска правильного рекуррентного выражения. После мемоизации этого правильного рекуррентного выражения таблица в явной форме становится в действительности не такой уж необходимой. Но если рекуррентное выражение некорректно, то мы оказываемся в абсолютно безнадёжной ситуации.

**Динамическое программирование –
это не просто заполнение таблиц.
Это интеллектуальная рекурсия.**

Наилучшая методика разработки алгоритмов динамического программирования включает два различных этапа.

1. **Рекурсивная формулировка задачи.** Записывается рекурсивная формула или алгоритм для задачи в целом в форме ответов на более мелкие подзадачи. Это самая трудный этап работы. Полная рекурсивная формулировка содержит две части:
 - а) **спецификацию.** Описание задачи, которую необходимо решить рекурсивно, на предельно понятном и точном естественном языке (русском, английском и т. п.), – но не как решать задачу, а что представляет собой задача, которую вы пытаетесь решить. Без такой спецификации невозможно – даже теоретически – определить, является ли ваше решение корректным;
 - б) **решение.** Привести точную рекурсивную формулу или алгоритм для задачи в целом в форме ответов на более мелкие (частные) варианты в точности той же самой общей задачи.
2. **Построение решений полученного на первом этапе рекуррентного выражения методом снизу вверх.** Записывается алгоритм, начинающийся с базовых случаев рекуррентного выражения и выполняющий его обработку вплоть до получения окончательного ре-

шения с учетом промежуточных подзадач, выполняемых в правильном порядке. Этот этап можно разделить на несколько более мелких, относительно простых (и даже выполняемых механически) шагов:

- a) **определение подзадач.** Какими всевозможными различными способами ваш рекурсивный алгоритм может вызывать сам себя, начиная с некоторых начальных входных данных? Например, аргументом для `RecFibo` всегда является целое число от 0 до n ;
- b) **выбор структуры данных для мемоизации.** Определение структуры данных, в которой можно сохранять решения каждой подзадачи, определенной на шаге (a). Обычно (но не всегда) это многомерный массив;
- c) **определение зависимостей.** За исключением базовых случаев, каждая подзадача зависит от других подзадач – от каких? Необходимо графически изобразить схему выбранной структуры данных, выбрать основной общий элемент, затем изобразить стрелки от каждого из прочих элементов, от которых зависит основной элемент. После этого формализовать итоговую схему;
- d) **определение правильного порядка вычислений.** Упорядочить подзадачи так, чтобы каждая подзадача следовала за подзадачами, от которых она зависит. В первую очередь необходимо рассматривать базовые случаи, затем подзадачи, зависящие только от базовых случаев, и т. д. Это построение завершается переходом к исходной задаче самого верхнего уровня. Зависимости, выявленные на предыдущем шаге, определяют отношения частичного порядка для подзадач, поэтому необходимо найти линейное расширение этих отношений частичного порядка. Будьте внимательны;
- e) **анализ пространственной сложности и времени выполнения.** Количество различных подзадач определяет пространственную сложность разрабатываемого мемоизованного алгоритма. Для вычисления общего времени выполнения необходимо просуммировать времена выполнения всех возможных подзадач, предполагая, что все более глубокие рекурсивные вызовы уже мемоизированы. В действительности это можно сделать сразу после шага (a);
- f) **запись алгоритма.** Вам известно, в каком порядке рассматривать подзадачи. Вам известно, как решить каждую подзадачу. Так сделайте это. Если выбранной структурой данных является массив, обычно это означает запись нескольких вложенных циклов, охватывающих исходное рекуррентное выражение, и замену рекурсивных вызовов на операции поиска в массиве.

Разумеется, необходимо доказательство того, что каждый из перечисленных выше шагов является корректным. Если рекуррентное выражение

некорректно или вы пытаетесь выстроить ответы на подзадачи в неправильном порядке, то ваш алгоритм работать не будет.

3.5. Внимание: жадность – это глупость

Если бы мы были невероятными счастливицами, то могли бы обойти стороной все эти рекуррентные выражения, таблицы и пр. и решить задачу, применив жадный (greedy) алгоритм. Как и алгоритм поиска с возвратом, жадный алгоритм формирует решение задачи из последовательности принятия решений, но принимает эти (частные) решения напрямую, без получения решений всех рекурсивных подзадач. Несмотря на то что такой подход выглядит весьма естественным, он почти никогда не работает – корректно решить задачи оптимизации с помощью жадного алгоритма удается довольно редко. Тем не менее для многих задач, которые должны решаться методами поиска с возвратом или динамического программирования, большинство обучающихся в первую очередь интуитивно выбирает применение жадной стратегии.

Например, жадный алгоритм для задачи сегментации текста может найти самый короткий (или, если хотите, самый длинный) префикс во входной строке, который является словом, принять этот префикс как первое слово в процессе разделения, а затем рекурсивно сегментировать оставшийся суффикс входной строки. Аналогично жадный алгоритм для задачи поиска максимальной возрастающей подпоследовательности может искать наименьший элемент входного массива, принять этот элемент в качестве начала целевой подпоследовательности, а затем рекурсивно выполнять поиск максимальной возрастающей подпоследовательности справа от этого элемента. Если описанные выше приемы показались вам глупыми, то можете себя похвалить – это даже и близко не правильные решения.

Каждый должен вытатуировать на тыльной стороне руки следующее высказывание, прямо под правилами логарифмов и нотацией «большое O»:

Жадные алгоритмы никогда не работают!
Вместо них используйте динамическое программирование.

Неужели никогда?

Никогда!

Что, действительно *никогда*?

Ну... почти никогда¹⁰.

¹⁰ Они почти никогда, никогда не работают! Так поприветствуем же трижды и еще раз сурового капитана Пинафоры! Так поприветствуем же трижды и еще раз сурового капитана Пинафоры! (Then give three cheers, and one cheer more, for the rigorous Captain of the Pinafore! Then give three cheers, and one cheer more, for the Captain of the Pinafore! – Акт 1 оперы «Н. М. S. Pinafore» Артура Салливана (Arthur Sullivan), либретто У. С. Гилберта (W. S. Gilbert). Предыдущие четыре строки в основном тексте также взяты из первого акта этой оперы – см. <http://www.aria-database.com/search.php?individualAria=1292>

Искушение применить жадную стратегию весьма велико, но она чрезвычайно редко бывает корректной, поэтому я настоятельно рекомендую использование следующей стратегии в каждом курсе по алгоритмам, даже (или, возможно, особенно) в тех курсах, в которых обычно не требуется доказательство корректности.¹¹

Вы никогда не получите зачет за использование любого жадного алгоритма – ни при выполнении каждого домашнего задания, ни на экзамене, даже если этот алгоритм корректен без формального доказательства его корректности.

Более того, подавляющее большинство задач, при решении которых обучающиеся поддаются искушению применить жадный алгоритм, в действительности лучше всего решаются с использованием динамического программирования. Поэтому я всегда даю изучающим алгоритмы в моем курсе следующий совет.

Когда вы пишете или даже мысленно произносите слово «жадный», «жаДИНА» («greeDY»), ваше подсознание подсказывает, что нужно использовать ДИНАмическое программирование (DYnamic programming).

Даже для задач, которые могут быть правильно решены с помощью жадных алгоритмов, обычно более эффективным способом является разработка алгоритма поиска с возвратом или алгоритма динамического программирования. Сначала заставьте его работать, потом сделайте его быстрым. Методики доказательства корректности жадных алгоритмов будут рассматриваться в следующей главе.

3.6. Максимальная возрастающая подпоследовательность

Еще одна задача, рассматриваемая в предыдущей главе, – вычисление длины максимальной возрастающей подпоследовательности заданного массива чисел $A[1..n]$. Для решения этой задачи были разработаны два различных рекурсивных алгоритма поиска с возвратом. Оба алгоритма в наихудшем случае выполнялись за время $O(2^n)$, и оба алгоритма можно существенно ускорить с помощью динамического программирования.

Первое рекуррентное выражение: кто следующий?

Первый алгоритм поиска с возвратом вычислял функцию $LISbigger(i, j)$, определяемую как длина максимальной возрастающей подпоследова-

¹¹ Ознакомление с этой стратегией в моих собственных курсах по алгоритмам существенно улучшает оценки качества знаний обучающихся, потому что значительно снижает частоту применения некорректных жадных алгоритмов.

тельности массива $A[j..n]$, в которой каждый элемент больше $A[i]$. Для этой функции было выведено следующее рекуррентное выражение:

$$LISbigger(i, j) = \begin{cases} 0 & , \text{если } j > n; \\ LISbigger(i, j + 1) & , \text{если } A[i] \geq A[j]; \\ \max \left\{ \begin{array}{l} LISbigger(i, j + 1) \\ 1 + LISbigger(j, j + 1) \end{array} \right\} & \text{иначе.} \end{cases}$$

Для решения исходной задачи можно добавить контрольное значение $A[0] = -\infty$ в массив и вычислить $LISbigger(0, 1)$.

Каждая рекурсивная подзадача определяется двумя индексами i и j , поэтому здесь имеется лишь $O(n^2)$ различных рекурсивных подзадач, которые необходимо рассмотреть. Можно мемоизировать результаты этих подзадач в двумерном массиве $LISbigger[0..n, 1..n]$ ¹². Кроме того, каждую подзадачу можно решить за время $O(1)$ без учета рекурсивных вызовов, так что мы должны ожидать, что окончательный алгоритм динамического программирования будет выполняться за время $O(n^2)$.

Порядок, в котором мемоизованный рекурсивный алгоритм заполняет соответствующий массив, на этом этапе неизвестен, и по рекуррентному выражению можно определить только то, что каждый элемент $LISbigger[i, j]$ записывается в массив после элементов $LISbigger[i, j + 1]$ и $LISbigger[j, j + 1]$ в следующем столбце, как показано на рис. 3.4.

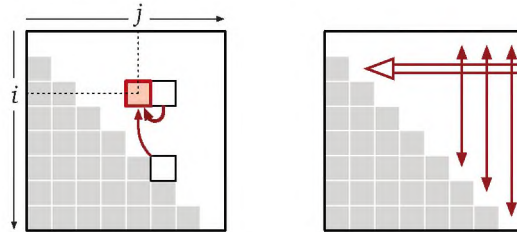


Рис. 3.4. Зависимости подзадач для нахождения максимальной возрастающей подпоследовательности и верный порядок вычислений

К счастью, этой частичной информации достаточно, чтобы определить верный порядок вычислений. Если заполнять таблицу по одному столбцу справа налево, то при переходе к конкретному элементу в таблице элементы, от которых он зависит, уже доступны. Возможно, это не тот порядок, который должен использовать рекурсивный алгоритм, но он работает, и мы принимаем его для продолжения. Схема справа на рис. 3.4 показывает порядок вычислений: здесь двойная стрелка обозначает внешний цикл, а одиночные стрелки – внутренний цикл. В рассматриваемом здесь случае одиночные стрелки изображены как двунаправленные, так как порядок заполнения каждого столбца не имеет значения.

¹² В действительности нам необходима только половина этого массива, потому что всегда соблюдается неравенство $i < j$. Но, даже если учитывать постоянные коэффициенты в этой книге (а они не учитываются), это будет напрасной потерей времени. Сначала заставьте алгоритм работать, потом улучшайте его.

Задание выполнено. Псевдокод для этого алгоритма динамического программирования показан ниже. Как и ожидалось, алгоритм явно выполняется за время $O(n^2)$. При необходимости можно снизить границу пространственной сложности с $O(n^2)$ до $O(n)$, если обеспечить поддержку только двух самых последних столбцов таблицы $LISbigger[:, j]$ и $LISbigger[:, j + 1]$.¹³

FastLIS($A[1..n]$):

```

A[0] ← -∞                                «Добавление контрольного значения.»
for i ← 0 to n                             «Базовые случаи.»
    LISbigger[i, n + 1] ← 0
for j ← n down to 1
    for i ← 0 to j - 1                     «...или что-то другое.»
        keep ← 1 + LISbigger[j, j + 1]
        skip ← LISbigger[i, j + 1]
        if A[i] ≥ A[j]
            LISbigger[i, j] ← skip
        else
            LISbigger[i, j] ← max{keep, skip}
return LISbigger[0, 1]
```



Второе рекуррентное выражение: что дальше?

Второй алгоритм поиска с возвратом вычислял функцию $LISfirst(i)$, определенную как длина максимальной возрастающей подпоследовательности массива $A[i..n]$, которая начинается с $A[i]$. Для этой функции было выведено следующее рекуррентное выражение:

$$LISfirst(i) = 1 + \max\{LISfirst(j) \mid j > i \text{ и } A[j] > A[i]\}.$$

Здесь мы предположили, что $\max \emptyset = 0$, т. е. что базовые случаи, подобные $LISfirst(n) = 1$, автоматически выпадают из этого рекуррентного выражения. Для решения исходной задачи можно добавить контрольное значение $A[0] = -\infty$ в массив и вычислить $LISfirst(0) - 1$.

В этом случае рекурсивные подзадачи обозначаются одним индексом i , поэтому можно мемоизировать рекуррентное выражение в одномерный массив $LISfirst[1..n]$. Каждый элемент $LISfirst[i]$ зависит только от элементов $LISfirst[j]$ при $j > i$, поэтому массив можно заполнять в порядке убывания индексов. Для вычисления каждого элемента $LISfirst[i]$ необходимо учитывать все элементы $LISfirst[j]$ для всех индексов $j > i$, но при этом нет необходимости рассматривать эти индексы j в каком-либо конкретном порядке. Итоговый алгоритм динамического программирования выполняется за время $O(n^2)$ и использует пространство $O(n)$.

¹³ Я же вам говорил, что не стоит беспокоиться о постоянных коэффициентах.

```
FastLIS2(A[1..n]):
```

```
  A[0] = -∞
```

⟨⟨Добавление контрольного значения.⟩⟩

```
  for i ← n down to 0
```

```
    LISfirst[i] ← 1
```

```
    for j ← i + 1 to n
```

⟨⟨...или что-то другое.⟩⟩

```
      if A[j] > A[i] and 1 + LISfirst[j] > LISfirst[i]
```

```
        LISfirst[i] ← 1 + LISfirst[j]
```

```
  return LISfirst[0] - 1
```

⟨⟨Не считать контрольное значение.⟩⟩

3.7. Расстояние редактирования

Расстояние редактирования (edit distance) между двумя строками – это минимальное количество операций вставки, удаления и замены букв, требуемое для преобразования одной строки в другую. Например, расстояние редактирования между строками **FOOD** и **MONEY** не больше четырех:

FOOD → MOOD → MOND → MONED → MONEY.

Эта функция расстояния редактирования была независимо предложена Владимиром Левенштейном в 1965 г. (работы по теории кодирования), Тарасом Винцюком в 1968 г. (работы по распознаванию речи) и Станиславом Уламом (Stanislaw Ulam) в 1972 г. (работы по биологическим рядам). Поэтому расстояние редактирования иногда называют расстоянием Левенштейна или расстоянием Улама (но весьма странно, что никогда не называют расстоянием Винцюка).

Процесс редактирования можно представить в наглядной форме, выравнивая строки с расположением одна над другой, с разрывом в первом слове для каждой операции вставки и с разрывом во второй строке для каждой операции удаления. Столбцы с двумя различными символами соответствуют операциям замены. При таком представлении число шагов редактирования – это просто число столбцов, которые не содержат два одинаковых символа.

F O O D
M O N E Y

Вполне очевидно, что невозможно преобразовать **FOOD** в **MONEY** за три шага, поэтому расстояние редактирования между **FOOD** и **MONEY** равно в точности четырем. К сожалению, не так просто в общем случае определить, является ли последовательность операций редактирования наикратчайшей. Например, приведенная ниже схема выравнивания показывает, что расстояние редактирования между строками **ALGORITHM** и **ALTRUISTIC** не более 6. Но является ли это самым лучшим результатом?

A L G O R I T H M
A L T R U I S T I C

Рекурсивная структура

Для разработки алгоритма динамического программирования вычисления расстояния редактирования сначала необходимо сформулировать эту задачу рекурсивно. Предложенное выше наглядное представление с выравниванием для редактирования последовательностей обладает чрезвычайно важным свойством «оптимальной подструктуры» (optimal substructure). Предположим, что у нас имеется представление с разрывами для самой короткой последовательности операций редактирования для двух строк. Если удалить самый последний столбец, то оставшиеся столбцы должны представлять самую короткую последовательность для оставшихся префиксов. Можно легко доказать это наблюдение от противного: если для префиксов существует более короткая последовательность операций редактирования, то возврат ранее удаленного последнего столбца дает более короткую последовательность операций редактирования для исходных строк. И как только мы определяем, что должно происходить в самом последнем столбце, Фея Рекурсия может определить остальную часть оптимального интервального представления.

Другими словами, выравнивание, наблюдаемое в таких представлениях последовательностей операций редактирования, упорядочено (без какой-либо конкретной причины) справа налево. Решение задачи вычисления редакторского расстояния требует создания последовательности принятия решений, по одному решению для каждого столбца в выходном представлении с выравниванием. В середине этой последовательности принятия решений мы получаем суффикс одной строки, уже выровненный по суффиксу другой строки.

ALGOR	I		T	H	M
ALTRU	I	S	T	I	C

Рис. 3.5. Середина процесса выравнивания строк

Поскольку стоимость выравнивания в точности равна количеству несопадающих столбцов, оставшиеся операции принятия решений не зависят от операций редактирования, которые уже были выбраны ранее, а зависят только от префиксов, которые пока еще не выровнены.

ALGOR
ALTRU

Рис. 3.6. Невыровненные префиксы строк

Следовательно, для любых двух входных строк $A[1..m]$ и $B[1..n]$ задачу вычисления редакторского расстояния можно сформулировать рекурсивно следующим образом: для любых индексов i и j пусть $Edit(i, j)$ обозначает редакторское расстояние между префиксами $A[1..i]$ и $B[1..j]$. Необходимо вычислить $Edit(m, n)$.

Рекуррентное выражение

Если i и j – положительные числа, то существует ровно три возможных варианта для самого последнего столбца в оптимальной схеме выравнивания строк $A[1..i]$ и $B[1..j]$:

- **вставка:** последний элемент в верхней строке пустой. В этом случае редакторское расстояние равно $Edit(i, j - 1) + 1$. Член $+1$ – это стоимость последней операции вставки, и рекурсивное выражение определяет минимальную стоимость оставшихся операций выравнивания;



Рис. 3.7. Операция вставки

- **удаление:** последний элемент в нижней строке пустой. В этом случае редакторское расстояние равно $Edit(i - 1, j) + 1$. Член $+1$ – это стоимость последней операции удаления, и рекурсивное выражение определяет минимальную стоимость оставшихся операций выравнивания;



Рис. 3.8. Операция удаления

- **замена:** в обеих строках имеются символы в последнем столбце. Если эти два символа различны, то редакторское расстояние равно $Edit(i - 1, j - 1) + 1$. Если оба символа равны, то замена не требуется, так что редакторское расстояние равно $Edit(i - 1, j - 1)$.



Рис. 3.9. Операция замены

В этом общем случае анализ разделяется на два варианта: либо $i = 0$, либо $j = 0$, но оба граничных варианта легко обрабатываются напрямую.

- Преобразование пустой строки в строку длиной j требует j операций вставки, поэтому $Edit(0, j) = j$.
- Преобразование строки длиной i в пустую строку требует i операций удаления, поэтому $Edit(i, 0) = i$.

Проверка правильности: оба эти базовых случая корректно определяют, что редакторское расстояние между двумя пустыми строками равно нулю.

Мы приходим к выводу, что функция $Edit$ определяется следующим рекуррентным выражением:

$$Edit(i, j) = \begin{cases} i & , \text{если } i = 0; \\ j & , \text{если } j = 0; \\ \min \begin{cases} Edit(i, j - 1) + 1 \\ Edit(i - 1, j) + 1 \\ Edit(i - 1, j - 1) + [A[i] \neq B[j]] \end{cases} & \text{иначе.} \end{cases}$$

Динамическое программирование

После получения рекуррентного выражения можно преобразовать его в алгоритм динамического программирования, следуя обычному механически выполняемому набору правил.

- **Подзадачи:** каждая рекурсивная подзадача определяется двумя индексами $0 \leq i \leq m$ и $0 \leq j \leq n$.
- **Структура мемоизации:** можно мемоизировать все возможные значения функции $Edit(i, j)$ в двумерном массиве $Edit[0..m, 0..n]$.
- **Зависимости:** каждый элемент $Edit[i, j]$ зависит только от трех смежных с ним элементов $Edit[i - 1, j]$, $Edit[i, j - 1]$ и $Edit[i - 1, j - 1]$.
- **Порядок вычислений:** если заполнять этот массив в стандартном порядке «сначала строки» – строка за строкой, сверху вниз, каждая строка слева направо, – то при переходе к любому элементу массива все элементы, от которых он зависит, уже доступны. (Это не единственный порядок вычислений, который можно было бы использовать, но он работает, поэтому продолжим с этим порядком.)

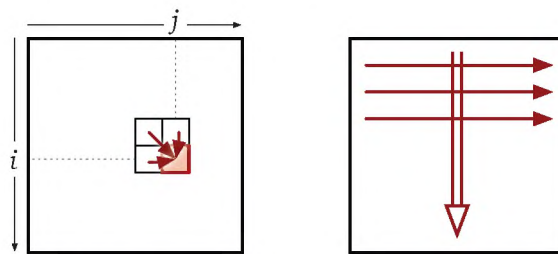


Рис. 3.10. Порядок вычисления массива мемоизации

- **Пространственная и временная сложность:** структура мемоизации использует пространство $O(mn)$. Каждый элемент массива $Edit[i, j]$ можно вычислить за время $O(1)$, как только мы узнаем предшествующие ему элементы, поэтому весь алгоритм в целом выполняется за время $O(mn)$.

Ниже приведен итоговый алгоритм динамического программирования:

```

EditDistance(A[1..m], B[1..n]):
  for j ← 0 to n
    Edit[0, j] ← j
  for i ← 1 to m
    Edit[i, 0] ← i
    for j ← 1 to n
      ins ← Edit[i, j - 1] + 1
      del ← Edit[i - 1, j] + 1
      if A[i] = B[j]
        rep ← Edit[i - 1, j - 1]
      else
        rep ← Edit[i - 1, j - 1] + 1
      Edit[i, j] ← min{ins, del, rep}
  return Edit[m, n]

```

Создателями этого алгоритма чаще всего считают Роберта Вагнера (Robert Wagner) и Майкла Фишера (Michael Fischer), описавших его в 1974 г. Но в полном соответствии с законом Стиглера об эпонимии точно такие же или более обобщенные алгоритмы были независимо разработаны Тарасом Винцюком в 1968 г., В. М. Величко и Н. Г. Загоруйко в 1970 г., Давидом Санкоффом (David Sankoff) в 1972 г., Питером Селлерсом (Peter Sellers) в 1974 г. и почти наверняка несколькими другими авторами.¹⁴ Странно, что ни один из перечисленных авторов не ссылается ни на Левенштейна, ни на Улама.

Таблица мемоизации для входных строк **ALGORITHM** и **ALTRUISTIC** показана на рис. 3.11. Числа, выделенные полужирным шрифтом и цветом, показывают позиции, в которых символы в обеих строках одинаковы. Расстояние редактирования между строками **ALGORITHM** и **ALTRUISTIC**, несомненно, равно шести.

Стрелки в этой таблице показывают, какие предшествующие элементы в действительности определяют каждый элемент. Направление каждой стрелки соответствует отдельной операции редактирования: горизонтальная стрелка = удаление, вертикальная стрелка = вставка, диагональная стрелка = замена. Утолщенные красные диагональные стрелки обозначают «нетребуемые» операции замены букв на самих себя. Любой путь по стрелкам из верхнего левого угла в нижний правый угол этой таблицы представляет оптимальную последовательность операций редактирования для

¹⁴ Этот алгоритм иногда ошибочно приписывают Саулу Нидлману (Saul Needleman) и Кристиану Вуншу (Christian Wunsch), датируя его создание 1970 г. Алгоритм Нидлмана–Вунша чаще называют стандартным алгоритмом динамического программирования для вычисления максимальной общей подпоследовательности двух строк (или равнозначно: расстояние редактирования для случая, когда разрешены только операции вставки и удаления) за время $O(mn)$, но и такое наименование также неверно. В действительности алгоритм Нидлмана–Вунша вычисляет (взвешенные) максимальные общие подпоследовательности (возможно, с штрафами за разрыв) за время $O(m^2n^2)$, используя другое рекуррентное выражение. Санкофф подробно описывает свой алгоритм с временем выполнения $O(mn)$ как улучшение алгоритма Нидлмана–Вунша.

двух рассматриваемых здесь строк. Пример массива мемоизации содержит ровно три направленных пути из верхнего левого в нижний правый угол, при этом каждый путь обозначает соответствующую отдельную последовательность операций редактирования, преобразующих строку **ALGORITHM** в строку **ALTRUISTIC**, как показано ниже.

A L G O R I T H M
A L T R U I S T I C

A L G O R I T H M
A L T R U I S T I C

A L G O R I T H M
A L T R U I S T I C

Рассматриваемый здесь алгоритм EditDistance в действительности не вычисляет и не сохраняет какие-либо стрелки в приведенной выше таблице, но стрелки, ведущие к любому элементу этой таблицы, можно воспроизвести динамически за время $O(1)$ по числовым значениям. Таким образом, после заполнения таблицы мемоизации можно воспроизвести наикратчайшую последовательность операций редактирования за дополнительное время $O(n + m)$.

	A	L	G	O	R	I	T	H	M	
	0	1	2	3	4	5	6	7	8	9
A	1	0	1	2	3	4	5	6	7	8
L	2	1	0	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	4	5	6
R	4	3	2	2	2	2	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	3	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	4	5	6
I	9	8	7	7	7	7	6	5	5	6
C	10	9	8	8	8	8	7	6	6	6

Рис. 3.11. Таблица мемоизации для строк **ALGORITHM** и **ALTRUISTIC**

3.8. Задача о сумме подмножеств

Напомню, что в задаче о сумме подмножеств необходимо определить, является ли сумма элементов какого-либо подмножества заданного масси-

ва положительных целых чисел $X[1..n]$ равной заданному целому числу T . В предыдущей главе был разработан рекурсивный алгоритм `SubsetSum`, который можно сформулировать по-другому, как показано ниже. Определим исходный массив $X[1..n]$ и логическую функцию:

$$SS(i, t) = \text{True},$$

если и только если сумма элементов некоторого подмножества $X[i..n]$ равна t .

Необходимо вычислить $SS(1, T)$. Эта функция определяется следующим рекуррентным выражением:

$$SS(i, t) = \begin{cases} \text{True} & , \text{ если } t = 0; \\ \text{False} & , \text{ если } t < 0 \text{ или } i > n; \\ SS(I + 1, t) \vee SS(I + 1, t - X[i]) & \text{ иначе.} \end{cases}$$

Это рекуррентное выражение можно преобразовать в алгоритм динамического программирования, следуя обычному шаблону.

- **Подзадачи:** каждая подзадача описывается целым числом i , таким что $1 \leq i \leq n + 1$, и целым числом $t \leq T$. Но подзадачи при $t < 0$ являются тривиальными, поэтому не имеет смысла их мемоизировать.¹⁵ Безусловно, мы можем изменить рекуррентное выражение так, чтобы эти подзадачи вообще никогда не возникали:

$$SS(i, t) = \begin{cases} \text{True} & , \text{ если } t = 0; \\ \text{False} & , \text{ если } i > n; \\ SS(I + 1, t) & , \text{ если } t < X[i]; \\ SS(I + 1, t) \vee SS(I + 1, t - X[i]) & \text{ иначе.} \end{cases}$$

- **Структура данных:** приведенное выше рекуррентное выражение можно мемоизировать в двумерный массив $S[1..n + 1, 0..T]$, где в $S[i, t]$ сохраняется значение функции $SS(i, t)$.
- **Порядок вычислений:** каждый элемент $S[i, t]$ зависит не более чем от двух других элементов, записываемых в форме $SS[I + 1, \cdot]$. Поэтому можно заполнять массив, рассматривая строки снизу вверх во внешнем цикле, а элементы в каждой строке рассматривать в произвольном порядке во внутреннем цикле.
- **Пространственная и временная сложность:** структура мемоизации использует пространство $O(nT)$. Если элементы $S[I + 1, t]$ и $S[I + 1, t - X[i]]$ уже известны, то можно вычислить $S[i, t]$ за постоянное время, поэтому алгоритм выполняется за время $O(nT)$.

Ниже приведен псевдокод итогового алгоритма динамического программирования:

¹⁵ Да, здесь я нарушаю собственное правило о преждевременной оптимизации.

FastSubsetSum($X[1..n], T$):

```

S[n + 1, 0] ← True
for t ← 1 to T
    S[n + 1, t] ← False
for i ← n down to 1
    S[i, 0] = True
    for t ← 1 to X[i] - 1
        S[i, t] ← S[i + 1, t]
    for t ← X[i] to T
        S[i, t] ← S[i + 1, t] ∨ S[i + 1, t - X[i]]
return S[1, T]

```

(Чтобы исключить случай $t < 0$.)



В наихудшем случае время выполнения $O(nT)$ этого алгоритма существенно улучшено по сравнению с временем $O(2^n)$ рекурсивного алгоритма поиска с возвратом при малых значениях T ¹⁶. Но если целевая сумма T значительно больше 2^n , то этот итеративный алгоритм в действительности медленнее, чем простейший рекурсивный алгоритм, потому что основное время тратится на решение подзадач, которые рекурсивный алгоритм вообще не рассматривает. Динамическое программирование не всегда является улучшением.¹⁷

3.9. Оптимальные двоичные деревья поиска

Последней задачей, которая рассматривалась в предыдущей главе, была задача об оптимальном двоичном дереве поиска. Входными данными является отсортированный массив ключей поиска $A[1..n]$ и массив счетчиков частот $f[1..n]$, где $f[i]$ – число операций поиска для $A[i]$. Наша цель – создать двоичное дерево поиска для этого множества так, чтобы общая стоимость всех операций поиска была как можно меньше.

Пусть определен массив частот f и пусть $OptCost(i, k)$ обозначает общее время поиска в оптимальном дереве поиска для подмассива $A[i..k]$. Тогда для функции $OptCost$ выводится следующее рекуррентное выражение:

$$OptCost(i, k) = \begin{cases} 0 & , \text{ если } i > k; \\ \sum_{j=i}^k f[j] + \min_{i \leq r \leq k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ + OptCost(r+1, k) \end{array} \right\} & \text{ иначе.} \end{cases}$$

¹⁶ Даже если задача о сумме подмножества является NP-трудной (NP-hard), эта временная граница не означает, что $P = NP$, потому что значение T не обязательно ограничено полиномиальной функцией от размера входных данных.

¹⁷ В своем отчете об исследованиях в 1967 г., где были предложены функции мемоизации (мето-функции), Дональд Мичи (Donald Michie) писал: «Табулировать значения функции, которая не понадобится, – пустая трата места, но пересчитывать одни и те же значения более одного раза – пустая трата времени». Но на самом деле табуляция значений функции, которая не потребуются, также является пустой тратой времени.

Вероятно, вы сможете предположить, что мы сделаем с этим рекуррентным выражением в итоге, но сначала избавимся от этого уродливого суммирования.

Для любой пары индексов $i \leq k$ пусть $F(i, k)$ обозначает общий счетчик частот для всех ключей в интервале $A[i..k]$:

$$F(i, k) := \sum_{j=i}^k f[j].$$

Эта функция определяется следующим простым рекуррентным выражением:

$$F(i, k) = \begin{cases} f[i] & , \text{ если } i = k; \\ F(i, k-1) + f[k] & \text{ иначе.} \end{cases}$$

Мы можем вычислить все возможные значения функции $F(i, k)$ за время $O(n^2)$, используя – вы угадали! – динамическое программирование. Обычная процедура с механически выполняемыми шагами дает следующий алгоритм динамического программирования:

```
InitF(f[1..n]):
  for i ← 1 to n
    F[i, i-1] ← 0
    for k ← i to n
      F[i, k] ← F[i, k-1] + f[k]
```

Этот короткий алгоритм будет использоваться как подпрограмма инициализации. Такая инициализация позволяет упростить первоначальное рекуррентное выражение для $OptCost$, как показано ниже:

$$OptCost(i, k) = \begin{cases} 0 & , \text{ если } i > k; \\ f[i, k] + \min_{i \leq r \leq k} \left\{ OptCost(i, r-1) + OptCost(r+1, k) \right\} & \text{ иначе.} \end{cases}$$

Теперь повернем рычаг.

- **Подзадачи:** каждая рекурсивная подзадача определяется двумя целыми числами i и k , таким что $1 \leq i \leq n+1$ и $0 \leq k \leq n$.
- **Мемоизация:** все возможные значения функции $OptCost$ можно сохранить в двумерном массиве $OptCost[1..n+1, 0..n]$. (В действительности будут использоваться только элементы $OptCost[i, j]$ при $j \geq i-1$, но это неважно.)
- **Зависимости:** каждый элемент $OptCost[i, k]$ зависит от элементов $OptCost[i, j-1]$ и $OptCost[j+1, k]$ для всех j , таких что $i \leq j \leq k$. Другими

словами, каждый элемент таблицы зависит от всех элементов, расположенных непосредственно слева или непосредственно ниже, как показано на рис. 3.12.

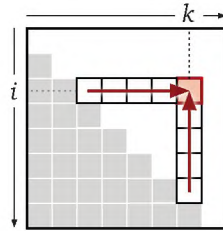


Рис. 3.12. Зависимость элементов таблицы

Показанная ниже подпрограмма заполняет элемент $OptCost[i, k]$, предполагая, что все элементы, от которых он зависит, уже были вычислены ранее.

```

ComputeOptCost(i, k):
  OptCost[i, k] ← ∞
  for r ← i to k
    tmp ← OptCost[i, r - 1] + OptCost[r + 1, k]
    if OptCost[i, k] > tmp
      OptCost[i, k] ← tmp
  OptCost[i, k] ← OptCost[i, k] + F[i, k]

```

- **Порядок вычислений:** существуют как минимум три различных порядка, которые можно использовать для заполнения массива. Первый часто встречается в работах большинства обучающихся: проход по таблице по одной диагонали за один шаг, начиная с простейших базовых случаев $OptCost[i, i - 1]$ и продолжая обработку по направлению к окончательному ответу $OptCost[1, n]$, как показано ниже:

```

OptimalBST(f[1..n]):
  InitF(f[1..n])
  for i ← 1 to n + 1
    OptCost[i, i - 1] ← 0
  for d ← 0 to n - 1
    for i ← 1 to n - d
      ComputeOptCost(i, i + d)
  return OptCost[1, n]

```

⟨...или что-то другое.⟩



Также можно выполнить проход по массиву строка за строкой снизу вверх, обрабатывая каждую строку слева направо, или стол-

бец за столбцом слева направо, обрабатывая каждый столбец снизу вверх.

```
OptimalBST2(f[1..n]):
  InitF(f[1..n])
  for i ← n + 1 downto 1
    OptCost[i, i - 1] ← 0
    for j ← i to n
      ComputeOptCost(i, j)
  return OptCost[1, n]
```

```
OptimalBST3(f[1..n]):
  InitF(f[1..n])
  for j ← 0 to n + 1
    OptCost[j + 1, j] ← 0
    for i ← j downto 1
      ComputeOptCost(i, j)
  return OptCost[1, n]
```

Как и ранее, можно проиллюстрировать эти варианты порядка вычисления, как показано на рис. 3.13, используя двойные стрелки для отображения внешнего цикла, а одиночные – для отображения внутреннего. Двухнаправленные одиночные стрелки на схеме первого варианта порядка вычисления (рис. 3.13, слева) означают, что порядок внутренних циклов не имеет значения.

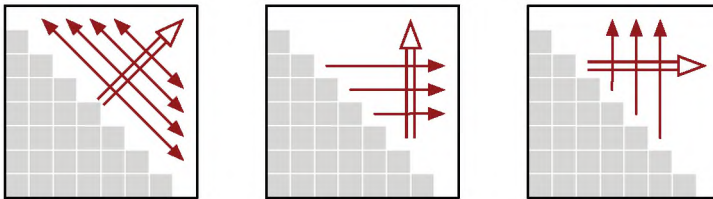


Рис. 3.13. Варианты порядка вычисления элементов массива

- **Временная и пространственная сложность:** структура мемоизации использует пространство $O(n^2)$. Вне зависимости от выбранного порядка вычислений требуется время $O(n)$ для вычисления каждого элемента $OptCost[i, k]$, поэтому весь алгоритм выполняется за время $O(n^3)$.

Как обычно, можно было бы спрогнозировать конечные временные и пространственные границы непосредственно по исходному рекуррентному выражению:

$$OptCost(i, k) = \begin{cases} 0 & , \text{ если } i > k; \\ f[i, k] + \min_{i \leq r \leq k} \left\{ OptCost(i, r - 1) + OptCost(r + 1, k) \right\} & \text{ иначе.} \end{cases}$$

Функция $OptCost$ принимает два аргумента, каждый из которых может принимать приблизительно n различных значений, поэтому, вероятно, потребуется структура данных размером $O(n^2)$. С другой стороны, в составе рекуррентного выражения имеются три переменные (i, k и r), каждая из ко-

торых может принимать приблизительно n различных значений, поэтому должно быть затрачено время $O(n^3)$ для вычисления всех значений.

3.10. Динамическое программирование для деревьев

До сих пор во всех примерах динамического программирования использовались многомерные массивы для сохранения результатов рекурсивных подзадач. Но эта структура данных не всегда является наиболее подходящей, как будет показано в следующем примере.

Независимое множество в графе – это подмножество вершин, между которыми нет соединяющих их ребер. Поиск максимального независимого множества вершин в произвольном графе представляет собой чрезвычайно трудную задачу, в действительности это одна из классических NP-трудных задач, которые будут рассматриваться в главе 12. Но в некоторых особых классах графов можно найти максимальные независимые множества вершин достаточно быстро. В частности, если входной граф является деревом с n вершинами, то можно действительно вычислить максимальное независимое множество вершин за время $O(n)$.

Предположим, что задано дерево T . Без потери для обобщения предположим, что T является деревом с корнем, т. е. в T есть особый узел, называемый *корнем* (root), и все ребра неявно направлены из этой вершины. (Если T является деревом без корня – связным ациклическим ненаправленным графом, – то в качестве корня можно выбрать произвольную вершину.) Вершина w называется потомком вершины v , если единственный путь от w к корню включает вершину v . Также потомком v является сама вершина v и потомки «детей» вершины v . Поддерево с корнем в вершине v состоит из всех потомков v и ребер между ними.

Для любого узла v пусть $MIS(v)$ обозначает размер максимального независимого множества вершин в поддереве с корнем v . Любым независимым множеством вершин в этом поддереве, за исключением самой вершины v , является объединение независимых множеств в поддеревьях с корнями в узлах-потомках вершины v . С другой стороны, любое независимое множество, включающее v , неизбежно исключает всех потомков вершины v , следовательно, включает независимые множества в поддеревьях с корнями в узлах-«внуках» (потомках следующего поколения) вершины v . Таким образом, функция MIS должна определяться следующим рекуррентным выражением, где нестандартная нотация $w \downarrow v$ означает « w – прямой потомок v »:

$$MIS(v) = \max \left\{ \sum_{w \downarrow v} MIS(w), 1 + \sum_{w \downarrow v} \sum_{x \downarrow w} MIS(x) \right\}.$$

Необходимо вычислить $MIS(r)$, где r – корень дерева T .

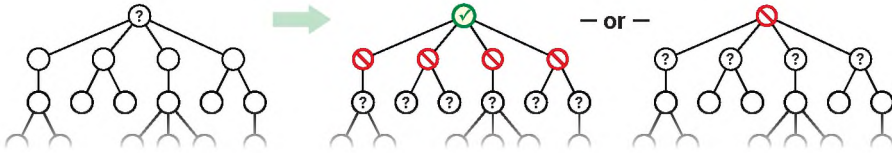


Рис. 3.14. Вычисление максимального независимого множества вершин в дереве

Какую структуру данных мы должны использовать для мемоизации этого рекуррентного выражения? Самый естественный вариант выбора – само дерево T . В частности, для каждой вершины в дереве T результат вычисления функции $MIS(v)$ сохраняется в новом поле $v.MIS$. (Теоретически можно было бы использовать массив, но в дальнейшем нам потребуются прямые и обратные указатели между каждым узлом и соответствующим ему элементом массива, так зачем же усложнять дело?)

Каков правильный порядок рассмотрения подзадач? Подзадачи, связанные с любым узлом v , зависят от подзадач, связанных с «детьми» и «внуками» узла v . Поэтому можно посещать узлы в любом предпочитаемом порядке, но с обеспечением посещения каждого узла прежде посещения его родителя, в частности, можно использовать стандартный обход в обратном порядке (post-order traversal).

Каково время выполнения этого алгоритма? Нерекурсивное время, связанное с каждым узлом v , пропорционально количеству «детей» и «внуков» вершины v . Это количество может значительно отличаться для различных вершин. Но можно изменить направление анализа: каждый узел добавляет постоянное количество времени к характеристике своего родителя и «деда». Поскольку каждая вершина имеет не более одного родителя и не более одного «деда», алгоритм выполняется за время $O(n)$.

Ниже приведен итоговый алгоритм динамического программирования. Да, он остается рекурсивным, потому что это наиболее естественный способ реализации обхода в обратном порядке.

```

TreeMIS(v):
  skipv ← 0
  for each child w of v
    skipv ← skipv + TreeMIS(w)
  keepv ← 1
  for each grandchild x of v
    keepv ← keepv + x.MIS
  v.MIS ← max{keepv, skipv}
  return v.MIS
  
```

Можно вывести даже еще более простой алгоритм с линейным временем выполнения, определив две отдельные независимые функции для узлов дерева T :

- пусть $MISyes(v)$ обозначает размер максимального независимого множества вершин поддерева с корнем в узле v , которое включает сам узел v ;
- пусть $MISno(v)$ обозначает размер максимального независимого множества вершин поддерева с корнем в узле v , которое не включает сам узел v .

И здесь необходимо вычислить $\max\{MISyes(r), MISno(r)\}$, где r – корень дерева T . Первые две функции определяются следующими взаимно рекуррентными выражениями:

$$MISyes(v) = 1 + \sum_{w \downarrow v} MISno(w);$$

$$MISno(v) = \sum_{w \downarrow v} \max\{MISyes(w), MISno(w)\}.$$

И в рассматриваемом здесь случае можно мемоизировать эти функции в самом дереве, определив два новых поля для каждой вершины. Простая версия обхода дерева в обратном порядке вычисляет обе функции в каждом узле за время $O(n)$. Приведенный ниже алгоритм не только выполняет мемоизацию значений обеих функций в узле v , но также возвращает большее из этих двух значений.

```

TreeMIS2(v):
  v.MISno ← 0
  v.MISyes ← 1
  for each child w of v
    v.MISno ← v.MISno + TreeMIS2(w)
    v.MISyes ← v.MISyes + w.MISno
  return max{v.MISyes, v.MISno}

```

Во второй строке внутреннего цикла используется значение $v.MISno$, которое было мемоизовано рекурсивным вызовом в предыдущей строке.

Упражнения

Для всех предлагаемых ниже упражнений – и в более общем смысле: при разработке любого нового алгоритма динамического программирования – настоятельно рекомендуется следовать пошаговой процедуре, описанной в разделе 3.4. Но при этом никогда не начинайте думать о таблицах или циклах `for` до тех пор, пока не получите полное рекурсивное решение, включающее четкую спецификацию на естественном языке (русском, английском и т. д.) всех рекурсивных подзадач, которые вы в действительнос-

ти решаете.¹⁸ Сначала заставьте алгоритм работать, потом сделайте его более быстрым.

Последовательности/Массивы

1. В своей предыдущей жизни вы работали кассиром в затерянной антарктической колонии Надирия (Nadiria), потратив лучшую часть своей жизни на размен денег для своих клиентов. Поскольку в Антарктике бумага является весьма редким и потому ценным ресурсом, по закону от кассиров требовалось минимально возможное количество банкнот при выдаче разменных денег. Благодаря склонности к нумерологии одного из основателей колонии валюта Надирии, которая называлась мечта-доллары (Dream-Dollars), была доступна в следующих номиналах: 1, 4, 7, 13, 28, 52, 91 и 365.¹⁹
 - ♣(а) Жадный алгоритм размена постоянно берет банкноту наибольшего номинала, не превышающего целевую сумму. Например, для размена 122 долл. с использованием жадного алгоритма сначала берется банкнота 91 долл., затем банкнота 28 долл., наконец, три банкноты 1 долл. Приведите пример, в котором этот жадный алгоритм использует больше банкнот мечта-долларов, чем минимально возможно. (Подсказка: возможно, проще написать небольшую программу, чем выполнять необходимые расчеты вручную.)
 - (б) Описать и проанализировать рекурсивный алгоритм, вычисляющий при заданном целом числе k минимальное количество банкнот, необходимых для размена k мечта-долларов. (Не беспокойтесь о скорости своего алгоритма, просто обеспечьте его корректность.)
 - (с) Описать алгоритм динамического программирования, вычисляющий при заданном целом числе k минимальное количество банкнот, необходимых для размена k мечта-долларов. (А этот алгоритм должен быть быстрым.)
2. Описать эффективные алгоритмы для описанных ниже вариантов задачи сегментации текста. Предположить, что имеется подпрограмма `IsWord`, принимающая в качестве входных данных массив символов и возвращающая `True`, если и только если принятая строка является «словом». Проанализировать предлагаемые алгоритмы по ограничению количества вызовов подпрограммы `IsWord`.

¹⁸ В моих курсах по алгоритмам любое решение с применением динамического программирования, не включающее спецификацию на естественном языке рекурсивных подзадач более низкого уровня, автоматически получает оценку «ноль», даже если решение в остальном превосходно. Введение этой стратегии существенно улучшило качество обучения студентов, поскольку значительно сократило число случаев, в которых студенты представляют некорректные (или непоследовательные) алгоритмы динамического программирования.

¹⁹ Более подробно об истории и культуре Надирии, включая изображения купюр мечта-долларов различных номиналов, см. здесь: <http://moneyart.biz/dd/>.

- (a) Задан массив символов $A[1..n]$, вычислить количество вариантов разделения массива A на слова. Например, если задана строка **ARTISTOIL**, то алгоритм должен возвращать 2 с учетом вариантов разделения **ARTIST · OIL** и **ART · IS · TOIL**.
- (b) Заданы два массива символов $A[1..n]$ и $B[1..n]$, определить, можно ли разделить A и B на слова по одинаковым индексам. Например, строки **BOTHEARTHANDSATURN · SPIN** и **PINSTARTRAPSANDRAGSLAP** можно разделить на слова по одинаковым индексам, как показано ниже:

BOT · HEART · HAND · SAT · URNS · PIN
PIN · START · RAPS · AND · RAGS · LAP

- (c) Заданы два массива символов $A[1..n]$ и $B[1..n]$, вычислить количество различных способов, которыми A и B можно разделить на слова по одинаковым индексам.
3. Предположим, что задан массив чисел $A[1..n]$, числа могут быть положительными, отрицательными или нулями, а кроме того, не обязательно целыми.
- (a) Описать и проанализировать алгоритм, который находит максимальную сумму элементов в непрерывном подмассиве $A[i..j]$.
- (b) Описать и проанализировать алгоритм, который находит максимальное произведение элементов в непрерывном подмассиве $A[i..j]$.

Например, если в качестве входных данных задан массив $[-6, 12, -7, 0, 14, -7, 5]$, то первый алгоритм должен возвращать 19, а второй алгоритм – 504 (см. рис. 3.15).

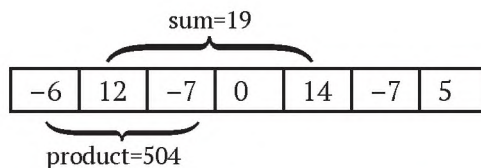


Рис. 3.15. Максимальная сумма и максимальное произведение в непрерывных подмассивах

При передаче одноэлементного массива $[-374]$ как входных данных первый алгоритм должен возвращать 0, а второй – 1. (Пустой интервал – это все равно интервал!) Для анализа предположить, что операции сравнения, сложения и умножения любой пары чисел выполняются за время $O(1)$.

[Совет: задача (a) была стандартным вопросом на интервью с кандидатом на должность в области ИТ как минимум с середины 1980-х гг. В интернете можно найти много корректных решений, этой задаче

посвящена даже отдельная страница в «Википедии». Но не позднее 2016 г. значительная часть решений, которые я находил в интернете для задачи (b) были более медленными, чем требуется, или действительно некорректными.]

4. В этом упражнении рассматриваются варианты задачи максимального подмассива (задача 3). Во всех вариантах входные данные состоят из массива действительных чисел $A[1..n]$ (числа могут быть положительными, отрицательными или нулями) и, возможно, дополнительного целого числа $X \geq 0$.
 - (a) *Перенос с вращением*: предположим, что A – кольцевой массив. При этом условии «непрерывным подмассивом» может быть интервал $A[i..j]$ или суффикс, за которым следует префикс $A[i..n] \cdot A[1..j]$. Описать и проанализировать алгоритм, находящий непрерывный подмассив A с максимальной суммой элементов.
 - (b) *Только длинные подмассивы*: описать и проанализировать алгоритм, находящий непрерывный подмассив A длиной не менее X с максимальной суммой элементов. (Предположить, что $X \leq n$.)
 - (c) *Только короткие подмассивы*: описать и проанализировать алгоритм, находящий непрерывный подмассив A длиной не более X с максимальной суммой элементов.
 - (d) *Цена точно известна*: описать и проанализировать алгоритм, находящий непрерывный подмассив A с максимальной суммой элементов, большей или равной X .
 - (e) Описать более быстрый алгоритм для задачи 4(d), если каждое число в массиве A неотрицательное.
5. В этом упражнении предлагается разработать эффективные алгоритмы для поиска оптимальных подпоследовательностей различных типов. Подпоследовательностью является все, что получается из некоторой последовательности при извлечении подмножества ее элементов, но с сохранением порядка следования этих элементов. Элементы в подпоследовательности не обязательно должны располагаться на соседних позициях в исходной последовательности. Например, все строки **C**, **DAMN**, **YATOAI** и **DYNAMICPROGRAMMING** представляют собой подпоследовательности строки **DYNAMICPROGRAMMING**.
 [Подсказка: только одна из предлагаемых задач может быть решена за время $O(n)$ с использованием жадного алгоритма.]
 - (a) Пусть $A[1..m]$ и $B[1..n]$ – два произвольных массива. Общей подпоследовательностью A и B является другая последовательность, являющаяся подпоследовательностью и A , и B . Описать эффективный алгоритм, вычисляющий длину максимальной общей подпоследовательности A и B .

- (b) Пусть $A[1..m]$ и $B[1..n]$ – два произвольных массива. Общей супер-последовательностью A и B является другая последовательность, содержащая A и B как подпоследовательности. Описать эффективный алгоритм, вычисляющий длину минимальной общей суперпоследовательности A и B .
- (c) Последовательность чисел $X[1..n]$ называется *битонической* (bitonic), если существует индекс i при $1 < i < n$, такой что префикс $X[1..i]$ является возрастающим, а суффикс $X[i..n]$ – убывающим. Описать эффективный алгоритм, вычисляющий длину максимальной битонической подпоследовательности произвольного массива целых чисел A .
- (d) Последовательность чисел $X[1..n]$ называется *колеблющейся* (oscillating), если $X[i] < X[i + 1]$ для всех четных i и $X[i] > X[i + 1]$ для всех нечетных i . Описать эффективный алгоритм, вычисляющий длину максимальной колеблющейся подпоследовательности произвольного массива целых чисел A .
- (e) Описать эффективный алгоритм, вычисляющий длину минимальной колеблющейся подпоследовательности произвольного массива целых чисел A .
- (f) Последовательность чисел $X[1..n]$ называется *выпуклой* (convex), если $2 \cdot X[i] < X[i - 1] + X[i + 1]$ для всех i . Описать эффективный алгоритм, вычисляющий длину максимальной выпуклой подпоследовательности произвольного массива целых чисел A .
- (g) Последовательность чисел $X[1..n]$ называется *слабо возрастающей* (weakly increasing), если каждый ее элемент больше среднего арифметического двух предыдущих элементов, т. е. $2 \cdot X[i] > X[i - 1] + X[i - 2]$ для всех $i > 2$. Описать эффективный алгоритм, вычисляющий длину максимальной слабо возрастающей подпоследовательности произвольного массива целых чисел A .
- (h) Последовательность чисел $X[1..n]$ называется *двукратно возрастающей* (double-increasing), если $X[i] > X[i - 2]$ для всех $i > 2$. (Другими словами, двукратно возрастающая последовательность – это идеальное перемешивание двух возрастающих последовательностей.) Описать эффективный алгоритм, вычисляющий длину максимальной двукратно возрастающей подпоследовательности произвольного массива целых чисел A .
- (i) Напомню, что последовательность чисел $X[i]$ является возрастающей, если $X[i] < X[i + 1]$ для всех i . Описать эффективный алгоритм, вычисляющий длину максимальной общей возрастающей подпоследовательности двух заданных массивов целых чисел. Например, $\{1, 4, 5, 6, 7, 9\}$ – максимальная общая возрастающая подпоследовательность последовательностей $\{3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3\}$ и $\{1, 4, 1, 4, 2, 1, 3, 5, 6, 2, 3, 7, 3, 0, 9, 5\}$.

6. Перемешивание (shuffle) двух строк X и Y создается посредством чередования символов (или групп символов) с объединением в новую строку, при этом сохраняется порядок символов в строках X и Y . Например, строка **BANANAANAS** представляет собой перемешивание строк **BANANA** и **ANAS** несколькими различными способами.

BANANA**ANAS**

BAN**ANA****ANA****NAS**

BAN**AN****A****NA****NAS**

Также строки **PRODGYRNAMMMIING** и **DYPRONGARMAMMicing** являются результатом перемешивания строк **DYNAMIC** и **PROGRAMMING**:

PRO**D****G****Y****R****N****A****M****M****I****I****N****G**

DY**P****R****O****N****G****A****R****M****A****M****M****I****C****I****N****G**

- (a) Заданы три строки $A[1..m]$, $B[1..n]$ и $C[1..m+n]$, описать и проанализировать алгоритм, определяющий, является ли C результатом перемешивания строк A и B .
- (b) Гладким перемешиванием (smooth shuffle) строк X и Y является такое перемешивание X и Y , в котором никогда не используется более двух соседних символов каждой строки. Например:

- **PR****D****O****Y****G****N****A****R****A****M****M****I****C****I****N****G** – гладкое перемешивание строк **DYNAMIC** и **PROGRAMMING**;
- **DY****P****R****O****N****G****A****R****M****A****M****M****I****C****I****N****G** – перемешивание строк **DYNAMIC** и **PROGRAMMING**, но это не гладкое перемешивание (из-за подстрок **OGR** и **ING**);
- **XX****X****X****X****X****XX****XX****XX****XX****XX** – гладкое перемешивание строк **XXXXXX** и **XXXXXXXXXX**;
- для строк **XXXX** и **XXXXXXXXXXXX** не существует какого-либо варианта гладкого перемешивания.

Описать и проанализировать алгоритм, который определяет для заданных трех строк X , Y и Z , является ли строка Z гладким перемешиванием строк X и Y .

7. Для каждой приведенной ниже задачи входные данные состоят из двух массивов $X[1..k]$ и $Y[1..n]$ при $k \leq n$.
- (a) Описать и проанализировать алгоритм, который определяет, является ли X подпоследовательностью Y . Например, строка **PPAP** является подпоследовательностью строки **PENPINEAPPLEAPPLEPEN**.
- (b) Описать и проанализировать алгоритм поиска минимального количества символов, которые можно удалить из строки Y , чтобы строка X не являлась ее подпоследовательностью. Кроме того, алгоритм должен находить максимальную подпоследовательность строки Y , которая не является суперпоследовательностью строки X . Например, после удаления двух символов из строки

PENPINEAPPLEAPPLEPEN строка PPAR перестает быть ее подпоследовательностью.

- ♥(с) Описать и проанализировать алгоритм, который определяет, встречается ли строка X как две непересекающиеся (не имеющие общих символов) подпоследовательности строки Y . Например, строка PPAR встречается как две непересекающиеся подпоследовательности в строке PENPINEAPPLEAPPLEPEN.
- (d) Предположим, что во входные данные также включен третий массив $C[1..n]$, состоящий из чисел, которые могут быть положительными, отрицательными или нулями, где элемент $C[i]$ – стоимость символа $Y[i]$. Описать и проанализировать алгоритм, вычисляющий вхождение строки X с минимальной стоимостью как подпоследовательности строки Y . То есть необходимо найти массив $L[1..k]$, такой что $L[j] < L[j + 1]$ и $X[L[j]] = Y[j]$ для каждого индекса j , а общая стоимость $\sum_{j=1}^k C[j]$ должна быть минимальной из всех возможных.
- (e) Описать и проанализировать алгоритм, вычисляющий общее количество (возможно, перекрывающихся) вхождений строки X как подпоследовательности строки Y . Для анализа предположить, что можно выполнить сложение двух произвольных целых чисел за время $O(1)$. Например, строка PPAR встречается ровно 23 раза как подпоследовательность строки PENPINEAPPLEAPPLEPEN. Если все символы в строках X и Y одинаковы, то алгоритм должен возвращать $\binom{n}{k}$.
- (f) Каково время выполнения алгоритма из задачи (d), если сложение двух ℓ -битовых целых чисел требует времени $O(\ell)$?
8. Описать и проанализировать алгоритм поиска длины максимальной непрерывной подстроки, которая встречается как с прямым, так и с обратным порядком символов во входной строке $T[1..n]$. Подстроки с прямым и обратным порядком символов не должны перекрывать друг друга. Ниже приведено несколько примеров:
- если задана входная строка ALGORITHM, то алгоритм должен возвращать 0;
 - если задана входная строка RECURSION, то алгоритм должен возвращать 1 для подстроки R;
 - если задана входная строка REDIVIDE, то алгоритм должен возвращать 3 для подстроки EDI. (Подстроки с прямым и обратным порядком символов не должны перекрывать друг друга.);
 - если задана входная строка DYNAMICPROGRAMMINGMANYTIMES, то алгоритм должен возвращать 4 для подстроки YNAM. (В данном конкретном случае алгоритм не должен возвращать 6 для подпоследовательности YNAMIR.)

9. Палиндром – это любая строка, которая читается абсолютно одинаково в прямом и обратном направлении, например **I**, или **DEED**, или **RACECAR**, или **AMANAPLANACATACANALPANAMA**.

- (a) Описать и проанализировать алгоритм поиска длины максимальной подпоследовательности заданной строки, при этом подпоследовательность должна быть палиндромом.

Например, максимальной подпоследовательностью-палиндромом строки **MANDYNAMICPROGRAMZLETMESHOWYOUTHEM** является **MHYMRORMYHM**, т. е. если эта строка задана как входные данные, то алгоритм должен вернуть 11.

- (b) Описать и проанализировать алгоритм поиска длины минимальной суперпоследовательности заданной строки, при этом суперпоследовательность должна быть палиндромом. Например, минимальной суперпоследовательностью-палиндромом строки **TWENTYONE** является **TWENTYOYTNEWT**, поэтому если строка **TWENTYONE** задана как входные данные, то алгоритм должен вернуть 13.

- (c) Любую строку можно разделить на последовательность палиндромов. Например, строку **BUBBASEESABANANA** («Bubba sees a banana») можно разделить на палиндромы показанными ниже способами (существует еще 65 различных способов):

BUB • BASEESAB • ANANA
B • U • BB • ASEESA • B • ANANA
BUB • B • A • SEES • ABA • N • ANA
B • U • BB • A • S • EE • S • A • B • A • NAN • A
B • U • B • B • A • S • E • E • S • A • B • A • N • A • N • A

Описать и проанализировать эффективный алгоритм поиска минимального количества палиндромов, из которых составлена заданная входная строка. Например, если задана входная строка **BUBBASEESABANANA**, то алгоритм должен вернуть 3.

- (d) Описать и проанализировать эффективный алгоритм поиска максимального целого числа k , такого что заданную строку можно было бы разделить на палиндромы длиной не менее k . Например:

- если задана строка **PALINDROME**, то алгоритм должен вернуть 1;
- если задана строка **BUBBASEESABANANA**, то алгоритм должен вернуть 3 для варианта разделения **BUB • BASEESAB • ANANA**;
- если задана строка из n одинаковых символов, то алгоритм должен вернуть n .

- (e) Описать и проанализировать эффективный алгоритм поиска количества различных способов разделения строки на палиндромы. Например:

- если задана строка **PALINDROME**, то алгоритм должен вернуть 1;
- если задана строка **BUBBASEESABANANA**, то алгоритм должен вернуть 70;
- если задана строка из n одинаковых символов, то алгоритм должен вернуть 2^{n-1} .

♥(f) Метапалиндром (metapalindrome) – это разделение строки на последовательность палиндромов, таких что последовательность длин этих палиндромов сама является палиндромом. Например:

BOB • S • MAM • ASEESA • UKU • L • ELE

Это метапалиндром строки **BOBSMAMASEESAUKULELE**, последовательность длин палиндромов в которой также является палиндромом (3, 1, 3, 6, 3, 1, 3). Описать и проанализировать эффективный алгоритм поиска длины минимального метапалиндрома для заданной строки. Например, если задана строка **BOBSMAMASEESAUKULELE**, то алгоритм должен вернуть 11.

10. Предположим, что задан массив положительных целых чисел $A[1..n]$. Возрастающая то вперед, то назад подпоследовательность – это последовательность индексов $I[1..t]$ со следующими свойствами:

- $1 \leq I[j] \leq n$ для всех j ;
- $A[I[j]] < A[I[j+1]]$ для всех $j < t$;
- если индекс $I[j]$ четный, то $I[j+1] > I[j]$;
- если индекс $I[j]$ нечетный, то $I[j+1] < I[j]$.

Менее формально: предположим, что задан массив из n квадратов, и каждый квадрат содержит положительное целое число. Предположим, что мы помещаем маркер в один из этих квадратов, а затем многократно перемещаем маркер влево (если он находится в квадрате с нечетным индексом) или вправо (если он находится в квадрате с четным индексом), т. е. перемещение всегда происходит от меньшего числа к большему. Тогда последовательность перемещений маркера является возрастающей то вперед, то назад подпоследовательностью.

Описать алгоритм, вычисляющий длину максимальной возрастающей то вперед, то назад подпоследовательности заданного массива из n целых чисел. Например, если задан входной массив, показанный на рис. 3.16,

1	1	8	7	5	6	3	6	4	4	8	3	9	1	2	2	3	9	4	0
1<	2>	3<	4>	5<	6>	7<	8>	9<	10>	11<	12>	13<	14>	15<	16>	17<	18>	19<	20>

Рис. 3.16. Исходный массив целых чисел

то алгоритм должен вернуть целое число 9, которое представляет длину возрастающей то вперед, то назад подпоследовательности, показанной на рис. 3.17.

0	1	2	3	4	6	7	8	9
20>	1<	15<	18>	10>	6>	4>	3<	13<

Рис. 3.17. Возрастающая то вперед, то назад подпоследовательность

11. Предположим, что необходимо напечатать абзац текста на листе бумаги (или, если вы так настаиваете, на экране компьютера). Текст состоит из последовательности n слов, при этом i -е слово имеет длину $\ell[i]$. Необходимо разделить абзац на несколько строк с общей длиной ровно L . Например, в соответствии с характеристиками $\text{T}_\text{E}\text{X}$ программы, используемой для печати этой книги, абзац, который вы читаете прямо сейчас, имеет ширину, приблизительно равную $11.94794 \text{ см} \approx 4.7055 \text{ дюйма}$.

В зависимости от того, как абзац разделяется на строки текста, необходимо обязательно вставлять различное количество пробелов между словами. Абзац должен быть полностью выровнен по ширине, т. е. первый символ каждой строки располагается на левой границе абзаца, а последний символ каждой строки, за исключением самой последней строки, располагается на правой границе абзаца. Между любыми двумя словами на одной строке обязательно должен быть размещен как минимум один символ пробела. Посмотрите на абзац, который вы читаете прямо сейчас. Именно таким должен быть выведенный на бумагу (экран) абзац.

Коэффициент удлинения (slop) макета абзаца определяется как сумма по всем строкам, за исключением последней, кубов числа дополнительных пробелов в каждой строке, не считая одного требуемого символа пробела между каждой парой соседних слов. Точнее говоря, если строка содержит слова с индексами от i до j , то коэффициент удлинения этой строки определяется по формуле $(L - j + i - \sum_{k=i}^j \ell[k])^3$. Описать алгоритм динамического программирования для печати (вывода) абзаца с минимальным коэффициентом удлинения.

12. Вы и ваш восьмилетний племянник Элмо решили поиграть в простую карточную игру. В начале игры карты раскладываются лицом (картинкой) вверх в длинный ряд. Каждой карте соответствует определенное (различное) число очков. После того как все карты разложены в ряд, вы и Элмо делаете ходы, удаляя либо левую крайнюю, либо правую крайнюю карту из ряда до тех пор, пока не будут удалены все карты. При каждом ходе вы можете решить, какую из двух (крайних) карт взять. Победителем игры становится тот, кто набрал наибольшее число очков после окончания игры.

Элмо, который никогда не слушал курсы по алгоритмам, придерживается очевидной жадной стратегии – при своем ходе он всегда берет карту с максимальным числом очков. Ваша задача – найти стратегию, которая позволяет одержать победу над Элмо, когда это возможно. (Возможно, победа над таким маленьким мальчиком выглядит некрасиво, но Элмо очень не любит, когда взрослые нарочно поддаются, чтобы он выиграл.)

- (a) Доказать, что вы не должны тоже использовать жадную стратегию. То есть показать, что существует игра, которую можно выиграть, но только если не использовать ту же жадную стратегию, что и Элмо.
 - (b) Описать и проанализировать алгоритм, определяющий при заданной начальной последовательности карт максимальное число очков, которые вы можете набрать, играя против Элмо.
 - ♣(c) Когда Элмо было четыре года, он применял еще более простую стратегию – при своем ходе он всегда выбирал следующую карту абсолютно случайно. То есть если при его ходе оставалось более одной карты, Элмо должен был взять крайнюю слева карту с вероятностью $1/2$ или крайнюю справа карту с вероятностью $1/2$. Описать алгоритм, определяющий при заданной начальной последовательности карт максимальное ожидаемое число очков, которое вы можете набрать, играя против четырехлетнего Элмо.
 - (d) Пять лет спустя тринадцатилетний Элмо стал гораздо более сильным игроком. Описать и проанализировать алгоритм, определяющий при заданной начальной последовательности карт максимальное число очков, которое вы можете набрать, играя против идеального соперника.
13. Скоро вы должны будете продемонстрировать свое мастерство в танцах. Завтра состоится большой танцевальный конкурс, к которому вы готовились всю жизнь, кроме того лета, когда со своим дядей на Аляске охотились на россомах. Вы заранее получили копию списка из n песен, которые арбитры будут воспроизводить во время конкурса, в хронологическом порядке. Повезло!

Вам известны все песни, все арбитры, и ваши танцевальные способности чрезвычайно хороши. Для каждого целого числа k вам известно, что если вы танцуете под k -ю песню в программе конкурса, то получите ровно $Score[k]$ баллов, но затем вы физически не сможете станцевать под следующие $Wait[k]$ песен (т. е. в не можете танцевать под песни с номерами от $k + 1$ до $k + Wait[k]$). Танцор, набравший максимальное суммарное число баллов к концу вечера, побеждает в конкурсе, поэтому желательно, чтобы ваша общая сумма баллов была настолько высокой, насколько это возможно.

Описать и проанализировать эффективный алгоритм вычисления максимальной общей суммы баллов, которую вы можете набрать. Входными данными для этого алгоритма является пара массивов $Score[1..n]$ и $Wait[1..n]$.

14. В новой игре-головоломке Candy Swap Saga XIII представлено n очаровательных зверей, пронумерованных от 1 до n . Каждая зверушка держит лакомство одного из трех типов: конфеты-маршмеллоу в форме орехов (circus peanuts), шоколадные плитки с начинкой Heath и итальянские шоколадные трюфели Cioccolateria Gardini. У вас в руке уже есть конфета – в начале игры вы получаете конфету-маршмеллоу в форме орехов (circus peanuts).

Чтобы набирать очки, вы подходите к каждому животному в порядке нумерации от 1 до n . При каждом ходе вы можете либо оставить свою конфету в руке, либо обменять ее на конфету, которую держит животное.

- Если вы обмениваете свою конфету на конфету того же типа, то получаете одно очко.
- Если вы обмениваете свою конфету на конфету другого типа, то теряете одно очко (да, число очков может быть отрицательным).
- Если вы подходите к животному и решаете не обменивать конфеты, то ваша сумма очков не изменяется.

Вы обязательно должны подходить к животным в заданном порядке, и после посещения любого животного вы не можете вернуться к нему снова.

Описать и проанализировать эффективный алгоритм вычисления максимального возможного числа набранных очков. Входные данные представлены массивом $C[1..n]$, где $C[i]$ – тип конфеты, которую держит i -е животное.

15. Ленни Рутенбар (Lenny Rutenbar), руководитель и основатель нового колледжа информационных технологий Максимилиана Р. Левчина (Maksymilian R. Levchin College of Computer Science), заказал создание последовательности рампы (небольших трамплинов) из снега на южном склоне трассы в Орчард Даунс (Orchard Downs)²⁰ и послал вызов Биллу Кудеки (Bill Kudeki), главе отдела электрической и компьютерной инженерии, для участия в скоростном спуске. Билл и Ленни будут спускаться с горы, пытаясь максимизировать время нахождения в воздухе. Победитель получает возможность расширить свое учреждение/колледж до Центра информационных технологий Зибеля (Siebel Center) и одновременно до нового здания ECE, а проигравший должен переместить свое учреждение/колледж в подземный коллектор Boneyard около лаборатории Лумиса (Loomis Lab).

²⁰ Северный склон круче и быстрее, но слишком короткий для настоящего соревнования.

Когда Ленни и Билл достигают трамплина, пока еще находясь на земле, они могут либо использовать этот трамплин для прыжка по воздуху, возможно, перелетая один или несколько следующих трамплинов, либо обойти этот трамплин и остаться на земле. Очевидно, что если Ленни или Билл перелетают через трамплин, то не могут использовать его для увеличения дальности прыжка.

(а) Предположим, что задана пара массивов $Ramp[1..n]$ и $Length[1..n]$, при этом $Ramp[i]$ – это расстояние от вершины горы до i -го трамплина, а $Length[i]$ – это расстояние, которое при использовании i -го трамплина каждый участник пролетит по воздуху. Описать и проанализировать алгоритм, определяющий максимальное общее расстояние, которое Ленни или Билл могут пройти по воздуху.

(б) Университетские юристы прослышали о небольшой ставке Ленни и Билла и сразу же выдвинули возражения против нее. Для защиты университета от судебных исков или от резкого роста страховых тарифов юристы установили верхнюю границу для количества прыжков, которые могут выполнить оба участника. Описать и проанализировать алгоритм, определяющий максимальное общее расстояние, которое Ленни или Билл могут пролететь по воздуху с учетом не более k разрешенных прыжков при заданных исходных массивах $Ramp[1..n]$ и $Length[1..n]$ и целого числа k как входных данных.

♥(с) Когда юристы поняли, что введение объявленных ими ограничение не привело к немедленной отмене состязания, они добавили новое ограничение: ни один трамплин нельзя использовать более одного раза. Уступая юридическому вмешательству, Ленни и Билл отказались от своей ставки и решили объединиться, чтобы создать эффектное шоу для зрителей. Описать и проанализировать алгоритм, определяющий максимальное общее расстояние, которое Ленни или Билл могут пролететь по воздуху, если каждый из них выполняет не более k прыжков (т. е. в сумме не более $2k$ прыжков) и если каждый трамплин используется не более одного раза.

16. Фермеры Боггис, Бунс и Бин построили полосу препятствий для Мистера Лиса. Полоса препятствий состоит из длинного ряда будок, при этом каждая будка имеет номер, нарисованный яркой красной краской на ее фасаде. Формально для Мистера Лиса задан массив $A[1..n]$, где $A[i]$ – номер, нарисованный на фасаде i -й будки. Каждый номер $A[i]$ может быть положительным, отрицательным или нулем. Все участники согласны со следующими правилами:

- в каждой будке Мистер Лис обязательно должен сказать «Ring!» или «Ding!»;

- если Мистер Лис говорит «Ring!» в i -й будке, то он зарабатывает вознаграждение в размере $A[i]$ цыплят. (Если $A[i] < 0$, то Мистер Лис платит штраф в размере $-A[i]$ цыплят.);
- если Мистер Лис говорит «Ding!» в i -й будке, то он платит штраф в размере $A[i]$ цыплят. (Если $A[i] < 0$, то Мистер Лис получает вознаграждение в размере $-A[i]$ цыплят.);
- Мистеру Лису запрещено говорить одно и то же слово более трех раз подряд. Например, если он сказал «Ring!» в будках 6, 7 и 8, то в будке 9 обязан сказать «Ding!»;
- все числовые итоги подводятся в конце, после того как Мистер Лис посетил все будки, и судья-посредник объявляет: «Hot box!» В действительности Мистер Лис не должен иметь при себе цыплят при прохождении полосы препятствий;
- наконец, если Мистер Лис нарушает одно из правил или завершает прохождение полосы препятствий, имея при себе фермерских цыплят, то фермеры застрелят его.

Описать и проанализировать алгоритм, вычисляющий максимальное количество цыплят, которое Мистер Лис может получить при прохождении полосы препятствий, если в качестве входных данных задан массив чисел $A[1..n]$. [Подсказка: берегитесь горячей сосновой шишки!]

17. Dance Dance Revolution – это танцевальная видеоигра, впервые представленная в Японии компанией Konami в 1998 г. Игроки стоят на платформе с изображением четырех стрелок, направленных вперед, назад, влево и вправо и размещенных в форме креста. В процессе игры проигрывается песня (мелодия), и последовательность из n стрелок ($\leftarrow$, $\uparrow$, $\downarrow$ или $\rightarrow$) прокручивается снизу вверх на экране. Точно в тот момент, когда очередная стрелка касается верхней границы экрана, игрок обязательно должен наступить на соответствующую стрелку на танцевальной платформе. (Движение стрелок по экрану синхронизировано со звуком, поэтому игрок обязательно должен наступать на стрелки в ритме проигрываемой мелодии.)

Вы играете в другой вариант этой игры под названием Vogue Vogue Revolution, целью которой является идеальное исполнение, но с минимальными по возможности перемещениями. Когда стрелка касается верхней границы экрана, если одна из ваших ног уже находится на правильной стрелке, то вы получаете один премиальный балл «стиля» за сохранение текущей позы. Если ни одна из ног не находится на правильной стрелке, вы должны переместить одну (и только одну) ногу из текущего положения на требуемую стрелку на платформе. Если вы вдруг наступили на неправильную стрелку, или промахнулись при шаге на правильную стрелку, или перемести-

ли сразу обе ноги одновременно, или сдвинули любую ногу, которая уже стоит на правильной стрелке, то все ваши очки стиля сгорают, и вы проигрываете.

Как вы должны перемещать ноги, чтобы набрать максимальное число очков за стиль? Для уточнения условий этой задачи предположим, что вы всегда начинаете с левой ноги, находящейся на стрелке $\leftarrow$, а правая нога находится на стрелке $\rightarrow$, и вы запомнили всю последовательность стрелок полностью. Например, если последовательность стрелок выглядит так: $\uparrow\uparrow\downarrow\downarrow\leftarrow\rightarrow\leftarrow\rightarrow$, то вы можете заработать 5 очков за стиль, перемещая ноги, как показано на рис. 3.18.

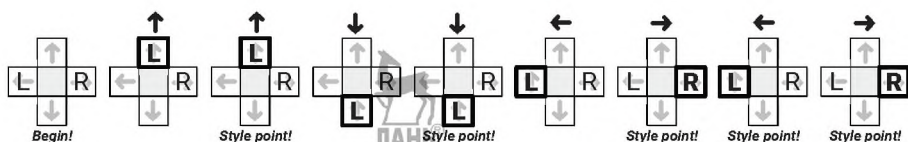


Рис. 3.18. Схема перемещения ног при игре в Vogue Vogue Revolution

- (а) Доказать, что при любой последовательности n стрелок можно заработать как минимум $n/4 - 1$ очков за стиль.
 - (б) Описать эффективный алгоритм поиска максимального числа очков за стиль, которое можно заработать во время заданной VVR-«программы». Входные данные для этого алгоритма представлены массивом $Arrow[1..n]$, содержащим последовательность стрелок.
18. Рассмотрим следующую форму игры с одним участником – вариант игры Scrabble (в русскоязычной версии – «Эрудит»). Мы начинаем со строго определенного конечного числа фишек, на каждой фишке изображена буква и числовое значение. В начале игры мы выбираем первые семь фишек из заданной последовательности и держим их в руке. При каждом ходе мы собираем слово на английском языке из нескольких или всех фишек, которые держим в руке, размещая фишки на игровом поле, и получаем сумму всех числовых значений на выложенных фишках как (выигрышные) очки. (Если слово на английском языке невозможно собрать из фишек, находящихся в руке, то игра сразу же завершается.) Затем мы поочередно берем следующую фишку из начала исходной последовательности до тех пор, пока (а) не наберем семь фишек в руке или (б) исходная последовательность окажется пустой. (Извините, но в этой игре не предусмотрены удваивания/утраивания очков за слова/буквы, приятные сюрпризы, пустые поля и возможность пропуска хода, как в оригинальной игре.) Наша цель – получить как можно больше очков.
- Например, рассмотрим последовательность из 20 фишек, показанную на рис. 3.19.



Рис. 3.19. Начальная последовательность из 20 фишек при игре в вариант Scrabble для одного игрока

Приняв эту последовательность в начале игры, мы можем заработать 68 очков, как показано ниже:

- сначала берем фишки $I_2, N_2, X_8, A_1, N_2, A_1, D_3$;
- собираем на игровом поле слово N_2, A_1, I_2, A_1, D_3 за 9 очков, в руке остаются фишки N_2, X_8 ;
- берем следующие пять фишек U_5, D_3, I_2, D_3, K_8 ;
- собираем слово U_5, N_2, D_3, I_2, D_3 за 15 очков, в руке остаются фишки K_8, X_8 ;
- берем следующие пять фишек U_5, B_4, L_2, A_1, K_8 ;
- собираем слово B_4, U_5, L_2, K_8 за 19 очков, в руке остаются фишки K_8, X_8 ;
- берем три последние фишки H_5, A_1, N_2 ;
- собираем слово A_1, N_2, K_8, H_5 за 16 очков, в руке остаются фишки X_8, A_1 ;
- собираем слово A_1, X_8 за 9 очков, в руке больше нет фишек, игра завершается.

- (а) Предположим, что последовательность фишек представлена двумя массивами: $Letter[1..n]$, содержащим последовательность букв от A до Z, и $Value[A..Z]$, где $Value[l]$ – числовое значение любой фишки с буквой l . Спроектировать и проанализировать эффективный алгоритм вычисления максимального числа очков, которое можно набрать при заданной последовательности фишек.
- (б) Теперь предположим, что две фишки с одинаковой буквой могут иметь различные числовые значения. В этом случае последовательность фишек представлена двумя массивами $Letter[1..n]$ и $Value[1..n]$, где $Value[i]$ – числовое значение i -й фишки. Спроектировать и проанализировать эффективный алгоритм вычисления максимального числа очков, которое можно набрать при заданной последовательности фишек.

В обеих задачах выводимым результатом является единственное число: максимальная возможная сумма набранных очков. Предположим (потому что это действительно так), что вы можете найти все слова на английском языке, которые можно составить из любого набора, содержащего не более семи фишек, и соответствующие этим словам суммы очков за время $O(1)$.

19. Предположим, что задано множество L , состоящее из n отрезков прямой на плоскости, при этом каждый отрезок содержит одну конечную точку, расположенную на прямой $y = 0$, и одну конечную точку, расположенную на прямой $y = 1$, и все $2n$ конечных точек различны.
- (a) Описать и проанализировать алгоритм вычисления максимального подмножества L , в котором нет пар пересекающихся отрезков.
 - (b) Описать и проанализировать алгоритм вычисления максимального подмножества L , в котором каждая пара отрезков пересекается.

Теперь предположим, что задано множество L , состоящее из n отрезков прямой на плоскости, при этом обе конечные точки каждого отрезка лежат на единичной окружности $x^2 + y^2 = 1$, и все $2n$ конечных точек различны.

- (c) Описать и проанализировать алгоритм вычисления максимального подмножества L , в котором нет пар пересекающихся отрезков.
 - (d) Описать и проанализировать алгоритм вычисления максимального подмножества L , в котором каждая пара отрезков пересекается.
20. Пусть P – множество n точек, равномерно распределенных на единичной окружности, и пусть S – множество m отрезков прямой с конечными точками из множества P . Конечные точки m отрезков не обязательно должны быть различными, поэтому n может быть значительно меньше $2m$.
- (a) Описать алгоритм поиска размера максимального подмножества отрезков из множества S , такого что каждая пара отрезков не имеет общих точек. Два отрезка не имеют общих точек, если они не пересекаются даже в конечных точках.
 - (b) Описать алгоритм поиска размера максимального подмножества отрезков из множества S , такого что каждая пара отрезков не имеет только внутренних общих точек. Два отрезка не имеют внутренних общих точек, если их пересечение либо пусто, либо представлено только конечной точкой обоих отрезков.
 - (c) Описать алгоритм поиска размера максимального подмножества отрезков из множества S , такого что каждая пара отрезков пересекается (в любых точках).
 - (d) Описать алгоритм поиска размера максимального подмножества отрезков из множества S , такого что каждая пара отрезков скрещивается. Два отрезка скрещиваются, если они пересекаются, но не в конечных точках.

Для получения максимальной оценки все четыре алгоритма должны выполняться за время $O(mn)$.

21. Вы водитель автобуса на скоростной трассе. Автобус заполнен беспокойными, гиперактивными, постоянно испытывающими жажду школьниками, кроме того, в автобусе есть стойка с газированной водой (сатуратор). В каждую минуту нахождения в автобусе школьник выпивает одну унцию газировки. Ваша цель – быстро развезти школьников по домам, чтобы общий объем выпитой всеми школьниками газировки был как можно меньшим.

Вам известно, сколько школьников выходят из автобуса на каждой остановке. Автобус начинает движение где-то в середине скоростной трассы (возможно, ни на одном из ее конечных пунктов) и едет с постоянной скоростью 37.4 мили в час. Вы должны вести автобус строго по скоростной трассе, но можете проехать вперед к одной остановке, затем вернуться назад к предыдущей остановке, меняя направление столько раз, сколько захотите. (Можно остановить автобус, высадить школьников и сразу же развернуться для движения в обратном направлении.)

Описать эффективный алгоритм высадки школьников в таком порядке, чтобы они успели выпить как можно меньше газировки. Входные данные состоят из маршрута (списка остановок и времени движения между последовательными остановками), количества школьников, которых вы высаживаете на каждой остановке, и текущее местоположение автобуса (которое можно считать одной из остановок).

22. Определим сумму (или агрегацию; *summary*) двух строк A и B как объединение (конкатенацию) подстрок в следующей форме:

- $\blacktriangle SNA$ обозначает подстроку SNA только первой строки A ;
- $\blacklozenge F00$ обозначает общую подстроку $F00$ обеих строк;
- $\blacktriangledown BAR$ обозначает подстроку BAR только второй строки B .

Сумма является корректной (*valid*), если можно восстановить исходные строки A и B , объединяя соответствующие подстроки этой суммы с сохранением порядка и отбрасывая разделители $\blacktriangle$, $\blacklozenge$ и $\blacktriangledown$. Каждый обычный символ имеет длину 1, а каждый разделитель $\blacktriangle$, $\blacklozenge$ или $\blacktriangledown$ имеет некоторую фиксированную неотрицательную длину Δ . Длиной суммы строк является сумма длин ее символов.

Например, каждая из приведенных ниже строк является корректной суммой строк $KITTEN$ и $KNITTING$:

- $\blacklozenge K \blacktriangledown N \blacklozenge ITT \blacktriangle E \blacktriangledown I \blacklozenge N \blacktriangledown G$ имеет длину $9 + 7\Delta$;
- $\blacklozenge K \blacktriangledown N \blacklozenge ITT \blacktriangle EN \blacktriangledown ING$ имеет длину $10 + 5\Delta$;
- $\blacklozenge K \blacktriangle ITTEN \blacktriangledown NITTING$ имеет длину $13 + 3\Delta$;
- $\blacktriangle KITTEN \blacktriangledown KNITTING$ имеет длину $14 + 2\Delta$.

Описать эффективный алгоритм, вычисляющий длину минимальной суммы двух заданных строк $A[1..m]$ и $B[1..n]$. Длина разделителя Δ также является частью входных данных для этого алгоритма. Например:

- если заданы строки **KITTEN** и **KNITTING** и $\Delta = 0$, то алгоритм должен вернуть 9;
- если заданы строки **KITTEN** и **KNITTING** и $\Delta = 1$, то алгоритм должен вернуть 15;
- если заданы строки **KITTEN** и **KNITTING** и $\Delta = 2$, то алгоритм должен вернуть 18.

23. Vankin's Mile – это американская игра для одного игрока на квадратной доске размером $n \times n$. В начале игрок помещает фишку на любое поле доски. Затем при каждом ходе игрок перемещает фишку на одно поле вправо или на одно поле вниз. Игра заканчивается, когда игрок выводит фишку за границу доски. Каждое поле доски имеет числовое значение – положительное, отрицательное или ноль. Игрок начинает с нулевого счета, при перемещении фишки на следующее поле значение этого поля прибавляется к текущему счету. Цель игры – набрать как можно больше очков.

Например, если задано поле, показанное на рис. 3.20, то игрок может набрать $8 - 6 + 7 - 3 + 4 = 10$ очков, поместив в начале игры фишку в поле 8 во втором ряду, а затем выполняя ходы: вниз, вниз, вправо, вниз, вниз. (Это не самый лучший возможный результат для этой таблицы чисел.)

-1	7	-8	10	-5
-4	-9	8	-6	0
5	-2	-6	-6	7
-7	4	7	-3	-3
7	1	-6	4	-9

Рис. 3.20. Сеанс игры в Vankin's Mile

- (а) Описать и проанализировать эффективный алгоритм вычисления максимального возможного счета игры Vankin's Mile, если в качестве входных данных задан массив числовых значений $n \times n$.
- (б) В европейской версии этой игры под названием Vankin's Kilometer игрок может перемещать фишку на одно поле вниз, вправо или влево при каждом ходе. Но, чтобы предотвратить бесконечный

набор очков, фишку запрещено передвигать на какое-либо поле более одного раза. Описать и проанализировать эффективный алгоритм вычисления максимального возможного счета игры Vankin's Kilometer, если в качестве входных данных задан массив числовых значений $n \times n$.²¹

24. Предположим, что задана битовая карта $m \times n$ как массив нулей и единиц $M[1..m, 1..n]$. Сплошной блок (solid block) в массиве M – это подмассив формы $M[i..i', j..j']$, в котором все биты равны. Сплошной блок является квадратным, если содержит равное количество строк и столбцов.
- (a) Описать алгоритм поиска максимальной площади сплошного квадратного блока в массиве M за время $O(n^3)$.
 - (b) Описать алгоритм поиска максимальной площади сплошного блока в массиве M за время $O(n^3)$.
 - (c) Описать алгоритм поиска максимальной площади сплошного блока в массиве M за время $O(n^2 \log n)$. [Совет: разделяй и властвуй.]
 - ♥(d) Описать алгоритм поиска максимальной площади сплошного блока в массиве M за время $O(n^2)$.
25. Предположим, что задан массив чисел $M[1..n, 1..n]$, числа могут быть положительными, отрицательными и нулями и не обязательно должны быть целыми. Описать алгоритм поиска максимальной суммы элементов в любом прямоугольном подмассиве формы $M[i..i', j..j']$. Для получения максимальной оценки алгоритм должен выполняться за время $O(n^3)$. [Подсказка: см. задачу 3.]
26. Описать и проанализировать алгоритм поиска прямоугольного образца максимальной площади, который встречается более одного раза в заданной битовой карте. Более точно: задан двумерный массив битов $M[1..n, 1..n]$ в качестве входных данных. Алгоритм должен выводить значение площади максимального повторяющегося прямоугольного образца в массиве M . Например, если задана битовая карта, показанная на рис. 3.21 слева, то алгоритм должен вернуть целое число 195, соответствующее площади 15×13 прямоугольного изображения песика. (Две копии повторяющегося образца могут перекрываться, хотя этого не происходит в рассматриваемом здесь примере.)

²¹ Если разрешить еще и ходы вверх, то получившаяся игра (Vankin's Fathom?) должна стать NP-трудной.

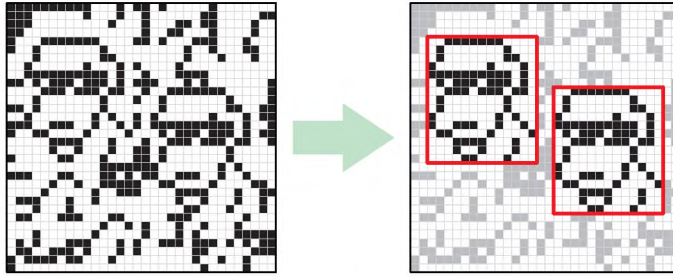


Рис. 3.21. Заданная битовая карта (слева) и найденные в ней повторяющиеся образцы (справа)

- (а) Для получения максимальной оценки описать алгоритм, выполняющийся за время $O(n^5)$.
 - ♥(b) Для получения экстремальной оценки описать алгоритм, выполняющийся за время $O(n^4)$.
 - ♣♥(c) Для получения экстра-экстремальной оценки описать алгоритм, выполняющийся за время $O(n^3 \text{ polylog } n)$.
27. Пусть P – множество точек на плоскости, при этом расположение точек является выпуклым (convex). Интуитивно понятно, что если эластичную резиновую ленту обернуть вокруг этих точек, то каждая точка должна касаться этой резиновой ленты. Более формально: для любой точки p из множества P существует прямая, отделяющая p от всех прочих точек множества P . Кроме того, предположим, что точки проиндексированы $P[1], P[2], \dots, P[n]$ по направлению против часовой стрелки вдоль этой воображаемой «резиновой ленты», начиная с крайней левой точки $P[1]$.
- В этой задаче предлагается решить особый случай задачи коммивояжера, в котором коммивояжер обязан посетить каждую точку множества P , а стоимостью перехода из одной точки $p \in P$ в другую точку $q \in P$ является евклидово расстояние $|pq|$.
- (а) Описать простой алгоритм вычисления кратчайшего циклического пути прохода по множеству P .
 - (b) Простой путь обхода – это путь, никогда не пересекающий самого себя. Доказать, что кратчайший путь обхода P обязательно должен быть простым.
 - (c) Описать и проанализировать эффективный алгоритм вычисления кратчайшего пути обхода P , начинающегося в левой крайней точке $P[1]$ и заканчивающегося в правой крайней точке $P[r]$.
 - (d) Описать и проанализировать эффективный алгоритм вычисления кратчайшего пути обхода P без ограничений по конечным точкам.

♥28. Описать и проанализировать алгоритм для решения задачи коммивояжера за время $O(2^n \text{ poly}(n))$. Задан не направленный граф G с n вершинами и с взвешенными ребрами. Алгоритм должен возвращать общий вес самого «легкого» цикла в графе G , который посещает каждую вершину ровно один раз, или ∞ , если граф G не содержит таких циклов. [Подсказка: тривиальный рекурсивный алгоритм поиска с возвратом требует время выполнения $O(n!)$.]

29. Пусть $W = \{w_1, w_2, \dots, w_n\}$ – конечное множество строк на некотором фиксированном алфавите Σ . Редакторский центр (edit center) для множества W – это строка $C \in \Sigma^*$, такая что максимальное расстояние редактирования от строки C до любой строки из множества W является минимальным, насколько это возможно. Редакторский радиус (edit radius) множества W – это максимальное расстояние редактирования от центра редактирования до любой строки из множества W . Множество строк может иметь несколько центров редактирования, радиус редактирования такого множества является единственным.

$$\text{EditRadius}(W) := \min_{C \in \Sigma^*} \max_{w \in W} \text{Edit}(w, C);$$

$$\text{EditCenter}(W) := \arg \min_{C \in \Sigma^*} \max_{w \in W} \text{Edit}(w, C).$$

(а) Описать и проанализировать эффективный алгоритм вычисления редакторского радиуса трех заданных строк.

♣♥(b) Описать и проанализировать эффективный алгоритм приближительного вычисления редакторского радиуса произвольного множества строк с коэффициентом 2. (Вычисление точного редакторского радиуса является NP-трудной задачей, если не задано строго определенное количество строк.)

♥30. Пусть $D[1..n]$ – массив цифр, каждая из которых является целым числом от 0 до 9. Цифровая подпоследовательность массива D – это последовательность положительных целых цифр, составленная обычным способом из непересекающихся подстрок массива D . Например, последовательность 3, 4, 5, 6, 8, 9, 32, 38, 46, 64, 83, 279 – это цифровая подпоследовательность последовательности первых цифр числа π :

3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 6, 4, 3, 3, 8, 3, 2, 7, 9

Длиной цифровой подпоследовательности является количество целых чисел, составляющих ее, но не количество цифр. В приведенном выше примере длина цифровой подпоследовательности равна 12. Как принято, цифровая подпоследовательность возрастает, если каждое ее число больше предыдущего.

Описать и проанализировать эффективный алгоритм вычисления максимальной возрастающей цифровой подпоследовательности массива D . [Подсказка: внимательно отнеситесь к предположениям, касающимся вычислений. Сколько времени займет сравнение двух k -разрядных чисел?]

Для получения наивысшей оценки алгоритм должен выполняться за время $O(n^4)$, более быстрые алгоритмы заслуживают сверхвысокой оценки. Самый быстрый известный мне алгоритм решения этой задачи выполняется за время $O(n^{3/2} \log n)$, для достижения этой границы требуется несколько особых приемов как при проектировании алгоритма, так и при его анализе, но ничего такого, что выходило бы за пределы текущего курса.²²

- ♥31. Рассмотрим следующий вариант классической задачи «Ханойская башня». Как обычно, предлагается n дисков различных размеров, размещенных на одном из трех стержней, пронумерованных 0, 1 и 2. Изначально все n дисков находятся на стержне 0, и они отсортированы по размеру: самый маленький наверху, самый большой внизу. Наша цель – переместить все диски на стержень 2. При каждом отдельном ходе можно переместить самый верхний диск с любого стержня на другой при условии соблюдения двух ограничений. Во-первых, нельзя помещать диск большего размера поверх диска меньшего размера. Во-вторых – и это нестандартное ограничение, – нельзя перемещать диск напрямую со стержня 0 на стержень 2.

Описать и проанализировать алгоритм вычисления точного числа ходов, требуемых для перемещения всех n дисков со стержня 0 на стержень 2 с учетом приведенных выше ограничений. Для получения наивысшей оценки алгоритм должен использовать только $O(\log n)$ арифметических операций в наихудшем случае. Для анализа предположим, что операция сложения или умножения двух k -разрядных чисел требует времени $O(k)$. [Подсказка: матрицы.]

Разделение последовательностей/массивов

32. Простое арифметическое выражение состоит из символов из множества $\{1, +, \times\}$ и скобок. Почти каждое целое число можно представить более чем одним простым арифметическим выражением. Например, все приведенные ниже простые арифметические выражения представляют число 14:

$$\begin{aligned} &1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\ &((1 + 1) \times (1 + 1 + 1 + 1 + 1)) + ((1 + 1) \times (1 + 1)) \\ &(1 + 1) \times (1 + 1 + 1 + 1 + 1 + 1 + 1) \\ &(1 + 1) \times (((1 + 1 + 1) \times (1 + 1)) + 1). \end{aligned}$$

²² Я уверен, что с применением более продвинутых методик время выполнения этого алгоритма можно сократить до $O(n^{3/2} \log \log n)$, но все подробности я пока еще не проработал.

Описать и проанализировать алгоритм вычисления при заданном входном целом числе n минимального количества единиц в простом арифметическом выражении, значение которого равно n . Количество скобок не имеет значения, учитывается только количество единиц. Например, при $n = 14$ алгоритм должен вернуть 8 для последнего варианта выражения, приведенного выше. Время выполнения алгоритма должно быть ограничено небольшой полиномиальной функцией от n .

33. Предположим, что задана последовательность целых чисел, разделенных знаками $+$ и $-$, например:

$$1 + 3 - 2 - 5 + 1 - 6 + 7.$$

Можно изменять значение этого выражения, добавляя скобки в различные позиции. Например:

$$\begin{aligned} 1 + 3 - 2 - 5 + 1 - 6 + 7 &= -1 \\ (1 + 3 - (2 - 5)) + (1 - 6) + 7 &= 9 \\ (1 + (3 - 2)) - (5 + 1) - (6 + 7) &= -17. \end{aligned}$$

Описать и проанализировать алгоритм вычисления для заданного входного списка целых чисел, разделенных знаками $+$ и $-$, максимального возможного значения выражения, которое можно получить, добавляя скобки. Скобки должны использоваться только для группирования операций сложения и вычитания, нельзя использовать скобки для создания неявной операции умножения, как показано здесь:

$$1 + 3(-2)(-5) + 1 - 6 + 7 = 33.$$

34. Предположим, что задана последовательность целых чисел, разделенных знаками $+$ и $\times$, например:

$$1 + 3 \times 2 \times 0 + 1 \times 6 + 7.$$

Можно изменять значение этого выражения, добавляя скобки в различные позиции. Например:

$$\begin{aligned} (1 + (3 \times 2)) \times 0 + (1 \times 6) + 7 &= 13 \\ ((1 + (3 \times 2 \times 0) + 1) \times 6) + 7 &= 19 \\ (1 + 3) \times 2 \times (0 + 1) \times (6 + 7) &= 104. \end{aligned}$$

- (a) Описать и проанализировать алгоритм вычисления максимального возможного значения заданного выражения при возможности добавления скобок, полагая, что все целые числа во входном выражении положительны. [Подсказка: это простая задача.]
- (b) Описать и проанализировать алгоритм вычисления максимального возможного значения заданного выражения при возможнос-

ти добавления скобок, полагая, что все целые числа во входном выражении неотрицательны.

- (с) Описать и проанализировать алгоритм вычисления максимального возможного значения заданного выражения при возможности добавления скобок без каких-либо ограничений для чисел во входном выражении.

Предполагается, что все арифметические операции выполняются за время $O(1)$.

35. После окончания университета Шам-Пубанана (Sham-Poobanana University) вы решили интервьюироваться на должность в банке Long Live Boole (Да здравствует Буль) на Уолл-стрит. Управляющий директор банка Луб Ждрожд (Eloob Egroeg – Джордж Буль наоборот) предложил каждому новому сотруднику задание класса «реши или умри», которое непременно должно быть решено в течение 24 ч. Те, кто не смогут решить задачу, будут уволены немедленно.

Войдя в банк в первый раз, вы заметили, что сотрудники офисов организованы в строгом линейном порядке в соответствии с большими буквами T или F , написанными на двери каждого офиса. Кроме того, между каждой соседней парой офисов расположена табличка, помеченная одним из символов $\wedge$, $\vee$ или $\oplus$. Когда вы спросили о смысле этих таинственных символов, Луб подтвердил, что T и F обозначают логические значения True и False, а символы на табличках представляют стандартные логические операторы AND, OR и XOR соответственно. Луб также объяснил, что эти буквы и символы описывают, могут ли успешно работать вместе определенные конкретные сочетания сотрудников. В начале каждого нового проекта Луб в иерархической форме объединяет в кластеры своих подчиненных, добавляя скобки в последовательность символов, чтобы получить недвусмысленное логическое выражение. Проект считается успешным, если результатом вычисления, полученного с помощью скобок логического выражения, является T (True).

Например, если в банке работают три сотрудника, а последовательность символов на их дверях и между ними выглядит как $T \wedge F \oplus T$, то существует ровно одна схема расстановки скобок: $(T \wedge (F \oplus T))$. Но при другом списке символов $F \wedge T \oplus F$ не существует способа добавления скобок, чтобы проект стал успешным.

В конце собеседования Луб предложил вам вопрос-задачу класса «реши или умри»: описать алгоритм принятия решения – можно ли расставить скобки в заданной последовательности символов так, чтобы полученное в результате логическое выражение при вычислении было равным T . Входные данные: массив $S[0..2n]$, где $S[i] \in \{T, F\}$, если i – четное число, и $S[i] \in \{\vee, \wedge, \oplus\}$, если i – нечетное число.

36. Каждый год как часть традиционной встречи Антарктические любители улиток Верхнего Глетчервила (Antarctican Snail Lovers of Upper Glacierville (SLUG – слизень)) устраивают состязание спаривания на круглом столе (Round Table Mating Race). Несколько улиток с наилучшими характеристиками для размножения размещается на краю круглого стола. Улитки пронумерованы по порядку вдоль края стола от 1 до n . Во время состязания каждая улитка блуждает по столу, оставляя за собой след слизи. Улитки специально обучены, чтобы не падать с края стола и не пересекать след слизи, даже собственный. При встрече двух улиток они объявляются размножающейся парой, удаляются со стола и скрываются в романтическую норку в земле для создания потомства. Следует отметить, что некоторые улитки могут никогда не найти пару, даже если состязание продолжается бесконечно.

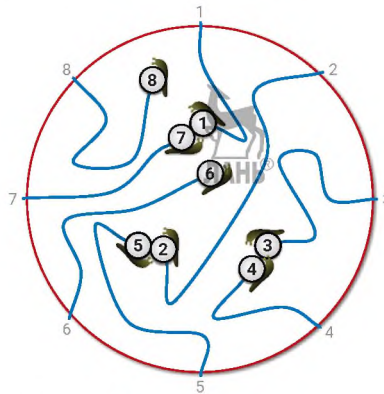


Рис. 3.22. Конечное состояние обычного состязания антарктического общества SLUG. Улитки 6 и 8 никогда не найдут пару. Организаторы состязания должны выдать премиальные в размере $M[3, 4] + M[2, 5] + M[1, 7]$

Для каждой пары улиток организаторы состязания антарктического общества SLUG объявили денежное вознаграждение, выплачиваемое владельцам улиток, если улитки встретились и образовали пару во время состязания спаривания. Точнее говоря, существует двумерный массив $M[1..n, 1..n]$, изображенный на стене рядом с круглым столом, при этом $M[i, j] = M[j, i]$ – вознаграждение, выплачиваемое, если улитки i и j встретились.

Описать и проанализировать алгоритм вычисления максимальной общей суммы вознаграждения, которое должны будут выплатить организаторы состязания, если входные данные представлены массивом M .

37. Вы откопали большую мраморную плиту в карьере. Для упрощения предположим, что плита имеет прямоугольную форму с высотой n дюймов и шириной m дюймов. Необходимо разрезать эту плиту на прямоугольные плитки меньшего размера, но размеры

должны быть различными – некоторые плитки для кухонной стойки, некоторые для крупных скульптурных проектов, некоторые для мемориальных надгробий. В вашем распоряжении есть станок для резки мрамора, который может выполнять горизонтальные или вертикальные разрезы любой прямоугольной плиты. В любой момент можно запросить текущую цену (спот-цену) $P[x, y]$ прямоугольной мраморной плитки размером x на y дюймов при любых положительных значениях x и y . Цены зависят от потребностей заказчиков, и люди, покупающие мраморную кухонную стойку, встречаются редко, поэтому не следует делать каких-либо предположений об их намерениях. Следует особо отметить, что прямоугольные плиты большего размера могут иметь значительно более низкие текущие цены. Входные данные представлены массивом текущих цен и целыми числами m и n . Описать алгоритм, определяющий, как распилить мраморную плиту размером $n \times m$, чтобы получить максимальную выгоду.

38. В этой задаче предлагается спроектировать эффективные алгоритмы создания оптимальных двоичных деревьев поиска, для которых соблюдаются дополнительные ограничения по балансу. Входные данные состоят из отсортированного массива ключей поиска $A[1..n]$ и массива счетчиков частоты $f[1..n]$, где $f[i]$ – количество операций поиска для $A[i]$. Функция стоимости в точности та же самая, что описана в разделе 3.9. Но теперь ваша задача – вычислить оптимальное дерево с соблюдением некоторых дополнительных ограничений.

- (а) АВЛ-деревья были самыми ранними самобалансирующимися двоичными деревьями поиска, впервые описанными в 1962 г. Георгием Адельсон-Вельским и Евгением Ландисом. АВЛ-дерево – это двоичное дерево поиска, в котором для каждого узла v высота левого и правого поддеревьев узла v отличается не более чем на единицу.

Описать и проанализировать алгоритм создания оптимального АВЛ-дерева для заданного множества ключей поиска и счетчиков частот.

- (б) Симметричные двоичные В-деревья – это другой тип самобалансирующихся двоичных деревьев, впервые описанных Рудольфом Байером (Rudolf Bayer) в 1972 г., они более широко известны как красно-черные деревья (red-black trees) после несколько упрощенного альтернативного описания Леонидасом Гимбасом (Leonidas Guibas) и Робертом Седжвиком (Robert Sedgwick) в 1978 г. Красно-черное дерево – это двоичное дерево поиска со следующими дополнительными ограничениями:

- каждый узел должен быть красным или черным;
- каждый красный узел имеет черного родителя;

- каждый путь от корня к листу содержит одинаковое число черных узлов.
- (с) АА-деревья были предложены Арне Андерссоном (Arne Andersson) в 1993 г. и немного упрощены (и поименованы) Марком Алленом Вайсом (Mark Allen Weiss) в 2000 г. АА-деревья также известны под названием «склоненные влево красно-черные деревья» (left-leaning red-black trees) после симметричного преобразования описания (с помощью различных алгоритмов ребалансирования) Робертом Седжвиком в 2006 г. АА-дерево – это красно-черное дерево с единственным дополнительным ограничением:
- ни один из левых узлов не должен быть красным²³.

Описать и проанализировать алгоритм создания оптимального АА-дерева для заданного множества ключей поиска и счетчиков частот.

39. Предположим, что задана битовая карта $m \times n$ как массив $M[1..m, 1..n]$, состоящий из нулей и единиц. Сплошной блок в массиве M – это подмассив формы $M[i..i', j..j']$, в котором все биты равны. Предположим, что необходимо разделить массив M на несколько непересекающихся блоков по возможности в наибольшем количестве.

Одна из естественных рекурсивных стратегий деления называется алгоритмом гильотинного подразбиения (guillotine subdivision). Если вся битовая карта M является сплошным блоком, то ничего делать не нужно. Иначе M разделяется на две битовые карты меньшего размера по горизонтали или по вертикали, затем рекурсивно выполняется разделение этих двух меньших битовых карт на сплошные блоки.

Любую процедуру гильотинного подразбиения можно представить как двоичное дерево, в котором каждый внутренний узел хранит позицию и ориентацию (направление) разреза, а в каждом листе хранится единственный бит 0 или 1, обозначающий содержимое соответствующего блока. Размер гильотинного подразбиения – это число листьев в соответствующем двоичном дереве (т. е. итоговое число сплошных блоков), а глубина гильотинного подразбиения – это глубина соответствующего двоичного дерева.

- (а) Описать и проанализировать алгоритм вычисления гильотинного подразбиения массива M минимального возможного размера.

²³ В описании Седжвика требование изменено: ни один из правых узлов не должен быть красным. Никакой разницы. Странно, что Андерссон и Седжвик умалчивают о том, следует ли измерять углы по часовой стрелке или против часовой стрелки, является ли Плутон планетой, соответствует ли выражение «более низкий ранг» свойству «лучше» или «хуже», что лучше – драться с сотней лошадей размером с утку или с одной уткой размером с лошадь.

- (b) Показать, что алгоритм гильотинного подразбиения не всегда приводит к разделению на наименьшее число сплошных блоков.
- (c) Описать и проанализировать алгоритм вычисления гильотинного подразбиения массива M с минимальной возможной глубиной.
- (d) Описать и проанализировать алгоритм, определяющий пиксел $M[i, j]$ при заданном дереве, представляющем гильотинное подразбиение массива M , и двух индексах i и j .

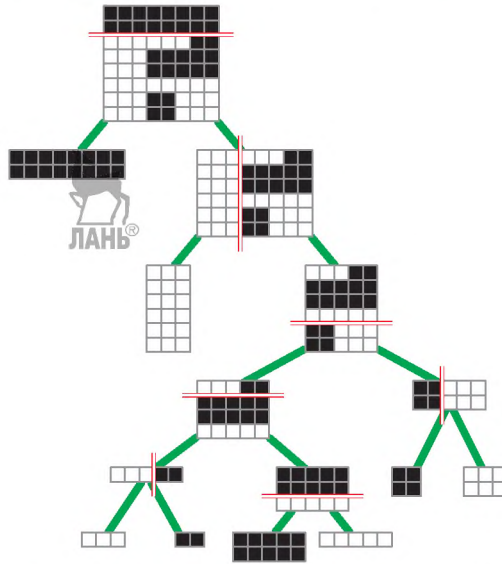


Рис. 3.23. Гильотинное подразбиение с размером 8 и глубиной 5

- (e) Глубина пиксела $M[i, j]$ в гильотинном подразбиении определяется как глубина листа, содержащего этот пиксел. Описать и проанализировать алгоритм вычисления гильотинного подразбиения массива M , такой что сумма глубин пикселей будет минимальной из возможных.
 - (f) Описать и проанализировать алгоритм вычисления гильотинного преобразования массива M , такой что сумма глубин черных пикселей будет минимальной из возможных.
- ♣40. Поздравляем! Вы приняты на работу в огромный онлайн-книжный магазин DeNile («Это не только река в Египте») для оптимизации работы роботов на складе. Каждая продаваемая в DeNile книга имеет единственный в своем роде номер ISBN (International Standard Book Number), представленный обычным числовым значением. В каждом складе в DeNile находятся длинные ряды контейнеров, каждый из которых содержит множество экземпляров одной и той же книги. Контейнеры размещены в порядке, отсортированном по ISBN. Номер ISBN нанесен на переднюю часть контейнера в машинно-читаемой

форме. Книги извлекаются из контейнеров роботами, перемещающимися по рельсам, проложенным параллельно рядам контейнеров.

В DeNile не ведется список соответствия контейнеров и номеров ISBN, это было бы слишком просто. Вместо этого для извлечения требуемой книги робот сначала должен найти соответствующий контейнер, применяя метод двоичного поиска. Поскольку для поиска требуется физическое перемещение робота, мы уже не можем предположить, что каждый шаг двоичного поиска выполняется за время $O(1)$. Уточнения:

- робот всегда начинает поиск с «контейнера 0» (в котором книги упаковываются в коробки для доставки покупателям);
- перемещение робота от i -го контейнера к j -му контейнеру требует $\alpha|i - j|$ при некотором постоянном значении α ;
- робот обязательно должен находиться прямо перед лицевой стенкой контейнера, чтобы прочитать его ISBN. Для чтения ISBN требуется β с при некотором постоянном значении β ;
- для изменения направления движения робота (от возрастания к убыванию ISBN и наоборот) дополнительно требуется γ с при некотором постоянном значении γ ;
- когда робот находит требуемый контейнер, он извлекает из него один экземпляр книги и возвращается к «контейнеру 0».

Описать и проанализировать алгоритм вычисления двоичного дерева поиска по контейнерам, которое минимизирует общее время, затрачиваемое роботом на поиск книг. Входные данные: массив целых чисел $f[1..n]$, где $f[i]$ – число запросов к роботу для извлечения экземпляра книги из i -го контейнера, а также временные параметры α , β и γ .

- ♣41. Стандартным методом улучшения производительности кеша деревьев поиска является упаковка большего числа ключей поиска и поддеревьев в каждом узле. В-дерево – это дерево с корнем, в котором каждый внутренний узел хранит до B ключей и указатели на не менее $B + 1$ потомков, каждый из которых является корнем меньшего В-дерева. Точнее говоря, каждый узел v содержит три поля:

- положительное целое число $v.d \leq B$;
- отсортированный массив $v.key[1..v.d]$;
- массив указателей на потомков $v.child[0..v.d]$.

В частности, число указателей на потомков ровно на единицу больше числа ключей²⁴.

²⁴ Обычно для В-деревьев требуется соблюдение двух дополнительных ограничений, обеспечивающих стоимость поиска $O(\log_b n)$ в наихудшем случае: каждый лист непременно должен иметь одну и ту же глубину, а каждый узел, за исключением, возможно, корня, обязательно должен содержать не менее $B/2$ ключей. Но в этой задаче нас не интересует оптимизация стоимости поиска в наихудшем случае, требуется вычислить общую стоимость последовательности поиска, поэтому описанные выше дополнительные ограничения не накладываются.

Каждый указатель на потомка $v.child[i]$ представляет собой либо `Null`, либо указатель на корень В-дерева, все ключи которого больше $v.key[i]$ и меньше $v.key[i + 1]$. В частности, все ключи в левом крайнем поддереве $v.child[0]$ меньше $v.key[1]$, а все ключи в правом крайнем поддереве $v.child[v.d]$ больше $v.key[v.d]$.

По интуиции вы должны хорошо запомнить схему, изображенную на рис. 3.24.

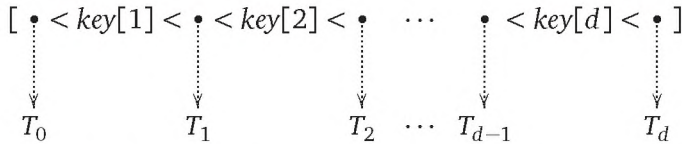


Рис. 3.24. Схема отношений узлов, ключей и указателей в дереве поиска

На рис. 3.24 T_i обозначает поддерево, на которое указывает потомок $child[i]$.

Стоимость (cost) поиска по ключу x в В-дереве – это число узлов в пути от корня до узла, содержащего x как один из своих ключей. 1-дерево – это просто стандартное двоичное дерево поиска.

Определим произвольное положительное целое число $B > 0$. (Предлагаю $B = 8$.) Предположим, что задан отсортированный массив ключей поиска $A[1, \dots, n]$ и соответствующий массив счетчиков частот $F[1, \dots, n]$, где $F[i]$ – число операций поиска по ключу $A[i]$. Ваша задача – описать и проанализировать эффективный алгоритм поиска В-дерева, которое минимизирует общую стоимость поиска по заданным ключам с заданными частотами.

- (a) Описать алгоритм с полиномиальным временем выполнения для особого случая $B = 2$.
 - (b) Описать алгоритм для произвольного значения B , выполняемый за время $O(n^{B+c})$ при некотором строго определенном целом значении c .
 - ♥(c) Описать алгоритм для произвольного значения B , выполняемый за время $O(n^c)$ при некотором строго определенном целом значении c , которое не зависит от B .
42. Строка w , состоящая из круглых $()$ и квадратных $[]$ скобок, является сбалансированной (balanced), если выполняется одно из следующих условий:
- w – пустая строка;
 - $w = (x)$ для некоторой сбалансированной строки x ;
 - $w = [x]$ для некоторой сбалансированной строки x ;
 - $w = xy$ для некоторых сбалансированных строк x и y .

Например, строка

$$w = ([()])([()])()$$

является сбалансированной, так как $w = xy$, где

$$x = ([()])() \text{ и } y = ([()])().$$

- (a) Описать и проанализировать алгоритм, определяющий, является ли заданная строка из круглых и квадратных скобок сбалансированной.
- (b) Описать и проанализировать алгоритм вычисления длины максимальной сбалансированной подпоследовательности заданной строки из круглых и квадратных скобок.
- (c) Описать и проанализировать алгоритм вычисления длины минимальной сбалансированной подпоследовательности заданной строки из круглых и квадратных скобок.
- (d) Описать и проанализировать алгоритм вычисления минимального редакторского расстояния от заданной строки из круглых и квадратных скобок до сбалансированной строки из круглых и квадратных скобок.
- ♥(e) Описать и проанализировать алгоритм вычисления максимальной общей сбалансированной подпоследовательности двух заданных строк из круглых и квадратных скобок.
- ♥(f) Описать и проанализировать алгоритм вычисления максимальной палиндромной сбалансированной подпоследовательности заданной строки из круглых и квадратных скобок.
- ♥(g) Описать и проанализировать алгоритм вычисления максимальной общей палиндромной сбалансированной подпоследовательности (ого!) двух заданных строк из круглых и квадратных скобок.

Для каждой задачи входными данными является массив $w[1..n]$, где $w[i] \in \{ (,), [,] \}$ для каждого индекса i . (Можно использовать другие символы вместо круглых и квадратных скобок, например L, R, l, r или $\triangleleft, \triangleright, \blacktriangleleft, \blacktriangleright$, но заранее предупредите преподавателя об используемых символах.)

- ♥43. Поздравляем! Вашей исследовательской группе только что был передан многолетний проект с бюджетом 50 млн долл., инициированный совместно DARPA, Google и McDonald's для создания DWIM: первого компилятора, читающего мысли программистов. В вашей заявке и во всех многочисленных пресс-релизах вы обещаете, что DWIM будет автоматически исправлять ошибки в любом заданном фрагменте кода по возможности с минимальными изменениями этого кода. К сожалению, настало время действительно заставить работать этот проклятый компилятор.

В качестве упражнения для разогрева вы решили взяться за следующую насущную подзадачу. Напомню, что редакторское расстояние между двумя строками – это минимальное количество операций вставки, удаления и замены одного символа, требуемое для преобразования одной строки в другую. Арифметическое выражение – это строка w , такая что:

- w – строка из одной или нескольких десятичных цифр;
- $w = (x)$ для некоторого арифметического выражения x ;
- $w = x \diamond u$ для некоторых арифметических выражений x и u и некоторого бинарного оператора $\diamond$.

Предположим, что задана строка символов из алфавита $\{\#, \diamond, (,)\}$, где $\#$ представляет десятичную цифру, а символ $\diamond$ – бинарный оператор. Описать и проанализировать алгоритм вычисления минимального редакторского расстояния от заданной строки до арифметического выражения.

44. Молекулы рибонуклеиновой кислоты (РНК) – это длинные цепочки нуклеотидов или оснований четырех различных типов: аденина (A), цитозина (C), гуанина (G) и урацила (U). Последовательность или цепочка молекулы РНК – это строка $b[1..n]$, где каждый символ $b[i] \in \{A, C, G, U\}$ соответствует основанию. В дополнение к химическим связям между соседними основаниями в цепочке могут формироваться водородные связи между определенными парами оснований. Множество связанных пар оснований называется вторичной структурой молекулы РНК.

Мы говорим, что две пары оснований (i, j) и (i', j') при $i < j$ и $i' < j'$ пересекаются (накладываются друг на друга – overlap), если $i < i' < j < j'$ или $i' < i < j' < j$. На практике большинство пар оснований не пересекается. Пересекающиеся пары оснований создают так называемые псевдоузлы (pseudoknots) во вторичной структуре, которые весьма важны для некоторых функций РНК, но более труднопредсказуемы.

Предположим, что необходимо спрогнозировать наилучшую из возможных вторичную структуру для заданной цепочки РНК. Мы примем существенно упрощенную модель вторичной структуры:

- каждой основание может быть связано не более чем с одним другим основанием;
- только пары оснований A–U и C–G могут быть связаны;
- пары формы $(i, i + 1)$ и $(i, i + 2)$ не могут быть связаны;
- связанные пары оснований не могут пересекаться.

Самое последнее (и самое нереалистичное) ограничение позволяет наглядно изобразить вторичную структуру РНК как один из видов утолщенного дерева (fat tree), показанного на рис. 3.25.

Описать и проанализировать эффективный алгоритм, который для заданной строки w и обобщенного регулярного выражения E определяет, действительно ли $w \in L(E)$.

В обеих задачах предполагается, что вам действительно задано дерево синтаксического разбора (парсинга) для (обобщенного) регулярного выражения, а не только строка.



Деревья и поддеревья

46. Вас только что назначили новым организатором первой ежегодной обязательной праздничной вечеринки в Giggle (дочерней компании Abugida). Сотрудники Giggle организованы в строгую иерархию – дерево с президентом компании в корне. Всеведущие прорицатели из отдела кадров присвоили каждому сотруднику действительное число, определяющее, насколько «веселым» он является. Для сохранения социальной структуры введено одно ограничение в списке приглашенных: сотрудникам не разрешено посещение вечеринки, если на ней присутствует их непосредственный руководитель. С другой стороны, президент компании обязательно должна присутствовать на вечеринке, даже если ее коэффициент веселости отрицательный, это все-таки ее компания. Создать алгоритм, составляющий список приглашенных на вечеринку, который максимизирует сумму коэффициентов «веселости» всех гостей.
47. Поскольку на прошлогоднюю праздничную вечеринку пришло очень мало людей, президент Giggle решила в этом году вместо вечеринки сделать подарок каждому сотруднику. Точнее говоря, каждый сотрудник должен получить один из трех подарков: (1) шестимесячный отпуск с оплатой всех расходов в любой точке мира, (2) завтрак на двоих «из всех блинов, которые вы сможете отсортировать» в заведении Jumping Jack Flash's Flapjack Stack Shack, или (3) крутой бумажный пакет, полный собачьих экскрементов. Корпоративные правила запрещают любому сотруднику получать в точности такой же подарок, что и его/ее непосредственный руководитель. Любой сотрудник, получивший более ценный подарок, чем его/ее непосредственный руководитель, почти наверняка будет уволен, чтобы избежать приступов мелочной зависти.

Поскольку вы официальный руководитель и координатор вечеринок Giggle, ваша работа состоит в принятии решения о том, какой подарок получит каждый сотрудник. Описать алгоритм распределения подарков таким способом, чтобы было уволено минимальное количество сотрудников. Да, вы можете отправить в подарок президенту крутой пакет с собачьими экскрементами.

Более формальное изложение задачи: задано дерево T с корнем, представляющее иерархию компании, необходимо пометить все узлы дерева T целыми числами 1, 2 или 3 так, чтобы каждый узел имел метку, отличающуюся от метки его родителя. Стоимость присваивания метки равна числу узлов с метками, значения которых меньше значений меток их родителей. Пример показан на рис. 3.9. Описать и проанализировать алгоритм вычисления процедуры разметки дерева T с минимальной стоимостью.

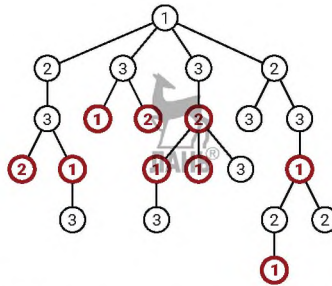


Рис. 3.26. Разметка дерева со стоимостью 9.

Девять узлов, выделенных утолщенными окружностями, имеют значения меток меньше, чем метки их родителей.

Это не оптимальный вариант разметки для показанного здесь дерева

48. После полного провала праздника с «крутыми собачьими экскрементами» вам пришлось срочно искать другую работу, и вы покинули компанию Gigggle и перешли в конкурирующую компанию Twitbook. К сожалению, новый президент Twitbook сразу решил подражать Gigggle, назначив собственную праздничную вечеринку, и, учитывая ваш предыдущий опыт, назначил вас официальным организатором праздника. Президент требует, чтобы вы пригласили ровно k сотрудников, включая самого президента, и от каждого приглашенного требуется обязательное присутствие. Да, это будет весело.

Как и в Gigggle, сотрудники Twitbook организованы в строгую иерархию: дерево с президентом компании в корне. Всеведущие прорицатели из отдела кадров присвоили каждому сотруднику действительное число, определяющее степень неловкости, испытываемой между приглашенными сотрудниками и их непосредственным руководителем. Отрицательное значение говорит о том, что сотрудники и их руководитель действительно нравятся друг другу. Ваша цель – выбрать подмножество ровно k приглашенных сотрудников, такое что суммарная степень неловкости на итоговой вечеринке была по возможности минимальной. Например, если в список приглашенных не включены сотрудники и их непосредственный руководитель, то общая степень неловкости равна нулю. Входные данные для этого алгоритма: дерево T , целое число k и значения степени неловкости для каждого узла дерева T .

- (а) Описать алгоритм вычисления общей степени неловкости для подмножества из k наименее стеснительных сотрудников, предполагая, что иерархия компании описывается двоичным деревом. То есть предположить, что каждый сотрудник является непосредственным руководителем не более чем двух других сотрудников.
- ♥(b) Описать алгоритм вычисления общей степени неловкости для подмножества из k наименее стеснительных сотрудников без каких-либо ограничений в иерархии компании.
49. Предположим, что необходимо распространить сообщение в широковещательном режиме по всем узлам дерева с корнем. Изначально это сообщение известно только корневому узлу. В каждом отдельном раунде любой узел, которому известно сообщение, может передать его не более чем одному из своих потомков. Пример см. на рис. 3.27.

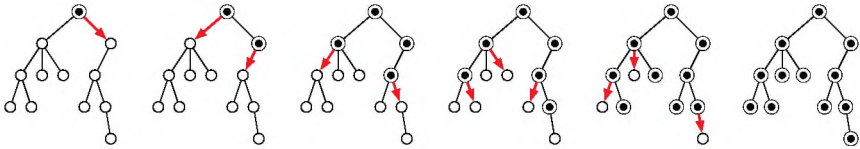


Рис. 3.27. Сообщение, распространяемое по дереву за пять раундов

- (а) Спроектировать алгоритм вычисления минимального числа раундов, требуемых для широковещательной передачи сообщения во все узлы двоичного дерева.
- ♦(b) Спроектировать алгоритм вычисления минимального числа раундов, требуемых для широковещательной передачи сообщения во все узлы произвольного дерева с корнем. [Совет: вы можете обнаружить, что методики, описанные в следующей главе, полезны для доказательства корректности этого алгоритма, даже если это не жадный алгоритм.]
50. Однажды Алекс надоело карабкаться по гимнастической стенке, и она решила собрать весьма большую группу друзей-скалолазов для упражнений на природе. В том месте, куда прибыла эта группа, находилась огромная широкая, но не слишком высокая каменная глыба с разнообразными зацепами для рук и ног. Алекс быстро определила «допустимое» множество перемещений, которые ее группа друзей может выполнить для перехода от одного зацепа к другому.
- Общую систему зацепов можно описать деревом T с корнем и с n вершинами, где каждая вершина соответствует одному зацепу, а каждое ребро – допустимому перемещению между зацепами. Пути подъема

сходятся, так как все они ведут на вершину этой глыбы, т. е. к особому зацепу в месте встречи, представленному корнем дерева T^{25} .

Алекс и ее друзья (все они превосходные скалолазы) решили сыграть в такую игру: максимальное возможное число скалолазов взбирается на глыбу, и от каждого требуется выполнить последовательность ровно k перемещений. Каждый скалолаз может выбрать произвольный зацеп для начала движения, но все перемещения обязательно должны начинаться с поверхности земли. Таким образом, каждый скалолаз последовательно проходит путь из k ребер в дереве T , при этом все пути направлены к корню. Но никаким двум скалолазам не разрешено одновременно касаться одного и того же зацепа, и пути, проходимые различными скалолазами, вообще не должны пересекаться.

Описать и проанализировать эффективный алгоритм вычисления максимального числа скалолазов, которые могут сыграть в эту игру. Более формальное описание: задано дерево T с корнем и целое число k . Необходимо найти максимальное возможное число непересекающихся путей в дереве T , при этом каждый путь должен иметь длину k . Здесь не предполагается, что T является двоичным деревом. Например, если в качестве входных данных задано дерево T , показанное ниже на рис. 3.28, и $k = 3$, то алгоритм должен вернуть целое число 8.

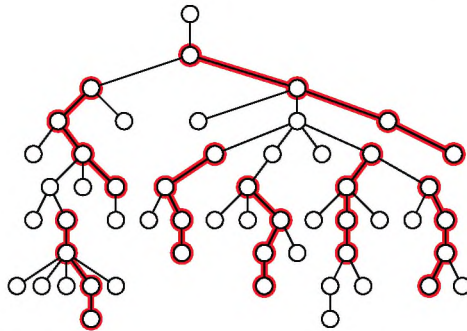


Рис. 3.28. Семь непересекающихся путей длиной $k = 3$.

Это не наибольшее множество таких путей в показанном здесь дереве

51. Пусть T – двоичное дерево с корнем и с n вершинами, и пусть $k \leq n$ является положительным целым числом. Желательно пометить k вершин в дереве T так, чтобы у каждой вершины был ближайший помеченный предок. Более формальное описание: мы определяем стоимость группировки (объединения в кластер) для любого подмножества вершин K как

$$\text{cost}(K) = \max_v \text{cost}(v, K),$$

²⁵ Вопрос: почему все преподаватели по информационным технологиям считают, что корни деревьев всегда находятся в самой верхней точке? Ответ: потому что они никогда не выходили на улицу!

- (а) Напомню, что двоичное дерево – это дерево с корнем, в котором каждый узел имеет (возможно, пустое) левое поддерево и (возможно, пустое) правое поддерево. Два двоичных дерева являются изоморфными, если и только если оба они пустые или их левые и правые поддеревья также являются изоморфными. Описать алгоритм поиска максимального общего двоичного поддерева двух заданных двоичных деревьев.

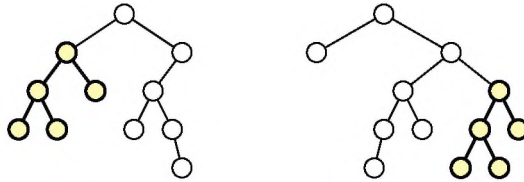


Рис. 3.30. Два двоичных дерева с выделенным в них максимальным общим двоичным поддеревом (с корнем)

- (б) В упорядоченном дереве с корнем каждый узел имеет последовательность потомков, которые являются корнями упорядоченных поддеревьев (разумеется, с корнями). Два упорядоченных дерева с корнями являются изоморфными, если оба они пустые или их i -е поддеревья также являются изоморфными для каждого индекса i . Описать алгоритм поиска максимального общего упорядоченного поддерева для двух упорядоченных деревьев T_1 и T_2 .
- ♦♥(с) В неупорядоченном дереве с корнем каждый узел имеет неупорядоченное множество потомков, которые являются корнями неупорядоченных поддеревьев. Два неупорядоченных дерева с корнями являются изоморфными, если оба они пустые или если поддеревья каждого корня можно упорядочить так, чтобы их i -е поддеревья являлись изоморфными для каждого индекса i . Описать алгоритм поиска максимального общего неупорядоченного поддерева для двух неупорядоченных деревьев T_1 и T_2 .
53. В этом задании предлагается найти эффективные алгоритмы для вычисления оптимальных поддеревьев в деревьях без корня, т. е. в связных ациклических не направленных графах. Поддеревом дерева без корня является любой связный подграф.
- (а) Предположим, что задано дерево без корня T с весами на его ребрах. Значения весов могут быть положительными, отрицательными или нулевыми. Описать алгоритм поиска пути в дереве T с максимальным суммарным весом.
- (б) Предположим, что задано дерево без корня T с весами в его вершинах. Значения весов могут быть положительными, отрицательными или нулевыми. Описать алгоритм поиска поддерева T

с максимальным суммарным весом. (В 2016 г. этот вопрос предлагался на собеседовании в Google.)

- (с) Пусть T_1 и T_2 – произвольные упорядоченные деревья без корня. Это значит, что соседи любого узла размещены в четко определенном циклическом порядке. Описать алгоритм поиска максимального общего упорядоченного поддерева для деревьев T_1 и T_2 .

- ♦♥(d) Пусть T_1 и T_2 – произвольные не упорядоченные деревья без корня. Описать алгоритм поиска максимального общего не упорядоченного поддерева для деревьев T_1 и T_2 .

54. Миноры с корнями (rooted minor) в деревьях с корнями – это естественное обобщение подпоследовательностей. Минор с корнем дерева с корнем T – это любое дерево, полученное посредством стягивания (contracting) одного или нескольких ребер. При стягивании ребра $u \rightarrow v$, где u – родитель v , потомок узла v становится новым потомком узла u , затем узел v удаляется. В частности, корень дерева T также является корнем каждого минора с корнем дерева T .

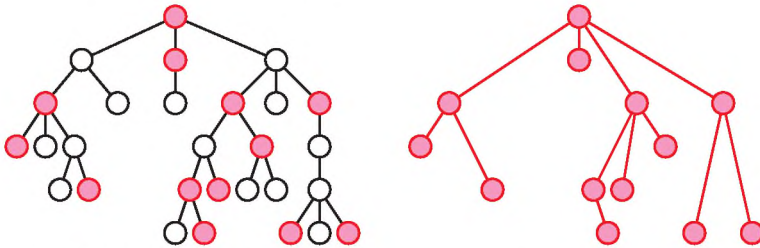


Рис. 3.31. Дерево с корнем и один из его миноров с корнем

- (a) Пусть T – дерево с корнем с помеченными узлами. Мы говорим, что дерево T скучное, если для каждого узла x все его потомки имеют одну и ту же метку. Потомки различных узлов могут иметь разные метки. Описать алгоритм поиска максимального скучного минора с корнем для заданного помеченного дерева с корнем.
- (b) Предположим, что задано дерево с корнем T , узлы которого помечены числами. Описать алгоритм поиска максимального пирамидально упорядоченного минора с корнем в дереве. То есть алгоритм должен возвращать максимальный минор с корнем M , такой что каждый узел в миноре M имеет метку, меньшую, чем метки его потомков в том же миноре M .
- (с) Предположим, что задано двоичное дерево T , узлы которого помечены числами. Описать алгоритм поиска максимального упорядоченного для двоичного поиска минора с корнем в дереве. То есть алгоритм должен возвращать максимальный минор с корнем M , такой что каждый узел в миноре M имеет не более двух

потомков, а обход с порядковой выборкой (упорядоченный обход) минора M представляет собой возрастающую подпоследовательность обхода с порядковой выборкой дерева T .

- (d) Напомню, что дерево с корнем является упорядоченным, если потомки каждого узла размещены в четко определенном порядке слева направо. Описать алгоритм поиска максимального упорядоченного для двоичного поиска минора произвольного упорядоченного дерева T , узлы которого помечены числами. При этом порядок слева направо узлов в миноре M должен быть согласован с их порядком в дереве T .
- ♥(e) Описать алгоритм поиска максимального общего упорядоченного минора с корнем для двух упорядоченных помеченных деревьев с корнем.
- ♥♦(f) Описать алгоритм поиска максимального общего неупорядоченного минора с корнем для двух неупорядоченных помеченных деревьев с корнями. [*Подсказка*: объедините метод динамического программирования с алгоритмом поиска максимального потока.]

Глава 4

Жадные алгоритмы

Дело в том, дамы и господа, что жадность — это хорошо. Жадность всегда работает, жадность всегда права. Жадность проясняет, проникает и улавливает сущность эволюционного духа. Жадность во всех ее формах, жажда жизни, денег, любви, знаний ознаменовала рост прогресса человечества. И жадность — запомните мои слова — спасет не только компанию Teldar Paper, но и другую неисправно работающую корпорацию под названием США.

— Гордон Гекко (Gordon Gekko) (роль Майкла Дугласа (Michael Douglas) в фильме «Уолл Стрит» («Wall Street»), 1987 г.)

У каждой человеческой проблемы всегда есть легкое решение — ясное, правдоподобное и неправильное.

— Генри Луис Менкен (Henry Louis Mencken), «Божественное откровение» («The Divine Afflatus»), газета New York Evening Mail от 16 ноября 1917 г.

4.1. Сохранение файлов на магнитной ленте

Предположим, что имеется набор из n файлов, которые необходимо сохранить на магнитной ленте¹. В будущем пользователи пожелают прочитать эти файлы с ленты. Считывание файла с ленты не похоже на считывание файла с диска — сначала необходимо выполнить перемотку вперед, чтобы пропустить все предшествующие файлы, — для этого требуется довольно-таки длительное время. Пусть $L[1..n]$ — массив, содержащий список длин каждого файла, т. е. файл i имеет длину $L[i]$. Если файлы сохраняются в порядке от 1 до n , то стоимость доступа к k -му файлу равна

$$\text{cost}(k) = \sum_{i=1}^k L[i].$$

¹ Читателей, склонных к возражениям, что магнитная лента устарела аж несколько десятилетий назад, советуем радушно пригласить на экскурсию в ближайший вычислительный центр, использующий суперкомпьютеры. Там попросите показать автоматизированную библиотеку магнитных лент (tape robot). Или в качестве альтернативы рассмотрите процесс расстановки последовательности книг на библиотечной полке. А вам известно, что эти странные объекты, похожие на кирпичи, сделаны из мертвых деревьев и типографской краски?

Стоимость отражает тот факт, что перед считыванием файла k мы обязательно должны сначала последовательно прочитать все предыдущие файлы на ленте. Если предположить, что вероятность доступа к каждому файлу одинакова, то ожидаемая стоимость поиска случайного файла равна

$$E[\text{cost}] = \sum_{k=1}^n \left(\frac{\text{cost}(k)}{n} \right) = \frac{1}{n} \sum_{k=1}^n \sum_{i=1}^k L[i].$$

Если изменить порядок хранения файлов на ленте, то изменится и стоимость доступа к файлам: считывание некоторых файлов станет более дорогим, но считывание других файлов станет дешевле. Различные варианты порядка файлов, вероятнее всего, приведут в итоге к различным значениям ожидаемой стоимости. В частности, пусть $\pi(i)$ обозначает индекс файла, сохраненного в позиции i на ленте. Тогда ожидаемая стоимость перестановки π равна

$$E[\text{cost}(\pi)] = \frac{1}{n} \sum_{k=1}^n \sum_{i=1}^k L[\pi(i)].$$

Какой порядок мы должны использовать, если требуется, чтобы эта ожидаемая стоимость была минимальной из всех возможных? Ответ выглядит интуитивно ясным: отсортировать файлы по увеличению длины. Но интуиция – коварный зверь. Единственный способ полностью удостовериться в том, что этот порядок верный, – «взлететь и с орбиты разнести здесь всё ядерным ударом» (с) к/ф «Чужие» это доказать, что он верный.

Лемма 4.1. $E[\text{cost}(\pi)]$ минимизирована, если $L[\pi(i)] \leq L[\pi(i+1)]$ для всех i .

Доказательство. Предположим, что $L[\pi(i)] \leq L[\pi(i+1)]$ для некоторого индекса i . Для упрощения записи пусть $a = \pi(i)$ и $b = \pi(i+1)$. Если поменять местами файлы a и b , то стоимость доступа к файлу a возрастает на $L[b]$, а стоимость доступа к файлу b уменьшается на $L[a]$. В целом обмен местами изменяет ожидаемую стоимость на $(L[b] - L[a])/n$. Но такое изменение является улучшением, так как $L[b] < L[a]$. Следовательно, если файлы расположены в неверном порядке, то можно снизить ожидаемую стоимость, поменяв местами некоторую неверно упорядоченную пару файлов. $\square$

Это первый рассмотренный нами пример корректного *жадного алгоритма* (greedy algorithm). Для минимизации общей ожидаемой стоимости доступа к файлам файл с самым дешевым доступом размещается первым, затем рекурсивно записываются все остальные файлы, без поиска с возвратом, без динамического программирования, только наилучший локальный вариант выбора и слепое движение вперед. Если воспользоваться эффективным алгоритмом сортировки, то время выполнения явно будет равным $O(n \log n)$ плюс время, требуемое на физическую запись файлов.

Чтобы показать, что жадный алгоритм действительно корректен, мы доказали, что результат работы любого другого алгоритма можно улучшить с помощью некоторого типа обмена местами файлов.

Обобщим эту идею в еще большей степени. Предположим, что также задан массив частот доступа к каждому файлу $F[1..n]$, т. е. к файлу i будет выполняться доступ ровно $F[i]$ раз за все время существования этой магнитной ленты. Тогда общая стоимость доступа ко всем файлам на ленте равна

$$\sum cost(\pi) = \sum_{k=1}^n \left(F[\pi(k)] \cdot \sum_{i=1}^k L[\pi(i)] \right) = \sum_{k=1}^n \sum_{i=1}^k \left(F[\pi(k)] \cdot L[\pi(i)] \right).$$

Как и ранее, изменение порядка файлов может изменить эту общую стоимость. Так какой вариант порядка мы должны использовать, если необходимо, чтобы общая стоимость была минимальной из всех возможных? (Этот вопрос по существу похож на задачу вычисления оптимального двоичного дерева поиска, но целевая структура данных и функция стоимости отличаются друг от друга, так что алгоритм также должен быть другим.)

Мы уже доказали, что если все частоты равны, то необходимо отсортировать файлы в порядке возрастания их размера. Если все частоты различны, а все длины файлов $L[i]$ равны, то по интуитивному предположению мы должны отсортировать файлы по уменьшающейся частоте доступа, чтобы файл с максимальной частотой доступа был первым. В действительности это не трудно доказать (это подсказка), изменив доказательство леммы 4.1. Но что если и размеры, и частоты доступа различны? В этом случае мы должны отсортировать файлы по отношению L/F .

Лемма 4.2. $\sum cost(\pi)$ минимизирована, если $\frac{L[\pi(i)]}{F[\pi(i)]} \leq \frac{L[\pi(i+1)]}{F[\pi(i+1)]}$ для всех i .

Доказательство. Предположим, что $L[\pi(i)] / F[\pi(i)] > L[\pi(i+1)] / F[\pi(i+1)]$ для некоторого индекса i . Для упрощения записи пусть $a = \pi(i)$ и $b = \pi(i+1)$. Если поменять местами файлы a и b , то стоимость доступа к файлу a возрастает на $L[b]$, а стоимость доступа к файлу b уменьшается на $L[a]$. В целом обмен местами изменяет общую стоимость на $L[b]F[a] - L[a]F[b]$. Но такое изменение является улучшением, так как

$$\frac{L[a]}{F[a]} > \frac{L[b]}{F[b]} \Leftrightarrow L[b]F[a] - L[a]F[b] < 0.$$

Следовательно, если любые два файла расположены в неверном порядке, то можно улучшить общую стоимость, поменяв местами эти файлы. $\square$

4.2. Планирование учебных курсов

Следующий пример немного сложнее. Предположим, что вы решили уйти из сферы информационных технологий и сменить свою специальность на Applied Chaos (применяемый (на практике) хаос). Кафедра Applied Chaos предлагает все свои учебные курсы в один и тот же день каждую неделю. Студенты прозвали этот день «Soberday» (День отрезвления, но, что любопытно, – не из-за преподавательского состава). Для каждого курса установлено отличающееся от прочих время начала и окончания: курс AC 101 («Ландшафтная архитектура из туалетной бумаги») начинается в 10:27 (в 22:27), а заканчивается в 11:51 (в 23:51); курс AC 666 («Материализация эсхатона (Конца времен; Апокалипсиса)») начинается в 4:18 (в 16:18), заканчивается в 4:22 (в 16:22) и т. д. Вы заинтересованы в получении диплома и степени как можно быстрее, поэтому хотите зарегистрироваться для изучения максимального возможного количества курсов. (Для курсов Applied Chaos не требуется какой-либо реальной работы.) Университетский регистрационный компьютер не позволяет вам зарегистрироваться в перекрывающихся по времени курсах, и никто на кафедре не знает, как обойти это его «свойство». На скольких курсах вы должны зарегистрироваться?

Более формальное описание задачи: пусть заданы два массива $S[1..n]$ и $F[1..n]$, содержащие списки времени начала и окончания каждого курса. Более конкретно можно предположить, что $0 \leq S[i] \leq F[i] \leq M$ для каждого i при некотором значении M (например, M – количество пикосекунд (пс) в Дне отрезвления). Ваша задача – выбрать максимальное возможное подмножество $X \in \{1, 2, \dots, n\}$, такое что для любой пары $i, j \in X$ выполняется либо $S[i] > F[j]$, либо $S[j] > F[i]$. Эту задачу можно наглядно представить в графическом виде, изобразив каждый курс как прямоугольник, левая и правая координата x которого соответствует времени начала и окончания курса. Цель – найти максимальное подмножество прямоугольников, которые не перекрывают друг друга по вертикали (см. рис. 4.1).

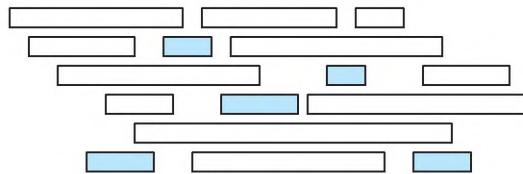


Рис. 4.1. Максимальное бесконфликтное планирование множества курсов

Эта задача имеет достаточно простое рекурсивное решение, основанное на наблюдении, сообщаящем, что вы либо можете выбрать курс 1, либо отказаться от него. Пусть B обозначает множество курсов, которые завершаются раньше, чем начинается курс 1, и пусть A обозначает множество курсов, которые начинаются после завершения курса 1:

$$B := \{i \mid 2 \leq i \leq n \text{ и } F[i] < S[1]\}$$

$$A := \{i \mid 2 \leq i \leq n \text{ и } S[i] > F[1]\}.$$

Если курс 1 находится в оптимальном расписании, то существуют также оптимальные расписания для множеств B и A , которые можно найти рекурсивно. Если расписание курса 1 не оптимально, то можно найти вариант оптимального расписания для $\{2, 3, \dots, n\}$ рекурсивно. Поэтому мы должны проверить оба варианта выбора и принять тот, который выдает более эффективное расписание. Вычисление этого рекурсивного алгоритма снизу вверх дает нам алгоритм динамического программирования, выполняемый за время $O(n^3)$. Я не буду вдаваться в подробности, потому что можно решить эту задачу лучше².

По интуитивному предположению нам должно подойти окончание первого курса как можно раньше, потому что при этом остается возможность использования максимального количества остальных курсов. Интуиция предлагает следующий простой жадный алгоритм. Просматриваем все уроки в порядке убывания их времени окончания, и, когда встречается урок, не конфликтующий с самым последним выбранным до этого момента уроком, принимаем его. На рис. 4.2 показано визуальное представление итогового жадного планирования.

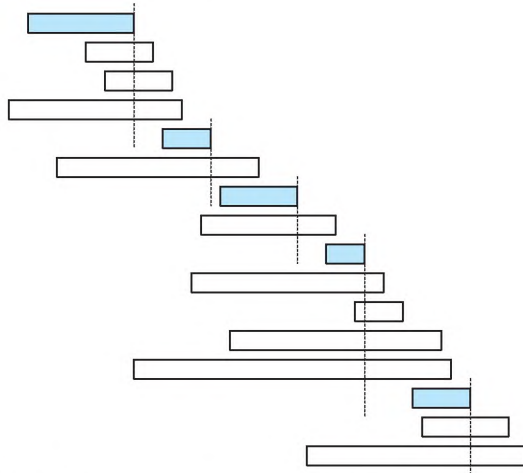


Рис. 4.2. Те же уроки (что и на рис. 4.1), отсортированные по времени завершения, и результат жадного построения расписания

Записать этот жадный алгоритм можно в несколько более формальном виде, как показано на рис. 4.3. (Надеюсь, что первая строка псевдокода понятна.) После начальной сортировки алгоритм представляет собой простой

² Но вы должны самостоятельно проработать эти подробности. Алгоритм динамического программирования можно использовать для поиска «наилучшего» варианта расписания для нескольких различных определений термина «наилучший», но жадный алгоритм, описываемый здесь, работает, только если «наилучший» означает «наибольший». Кроме того, вы можете улучшить время выполнения до $O(n^2)$, используя другое рекуррентное выражение.

цикл с линейным временем выполнения, поэтому весь алгоритм в целом выполняется за время $O(n \log n)$.

```

GreedySchedule(S[1..n], F[1..n]):
    sort F and permute S to match
    count ← 1
    X[count] ← 1
    for i ← 2 to n
        if S[i] > F[X[count]]
            count ← count + 1
            X[count] ← i
    return X[1 .. count]

```

Рис. 4.3. Жадный алгоритм поиска максимального множества не перекрывающихся по времени уроков

Чтобы доказать, что алгоритм GreedySchedule действительно вычисляет максимальный бесконфликтный план курсов, используется перестановка (обмен местами) аргументов, подобная той, что применялась для сортировки файлов на магнитной ленте. Мы не утверждаем, что результат жадного построения расписания является единственным оптимальным расписанием, могут существовать и другие варианты (см. рис. 4.1 и 4.2). Все, что мы можем утверждать, – как минимум один из вариантов оптимального плана вычисляется с помощью этого жадного алгоритма.

Лемма 4.3. Как минимум один оптимальный бесконфликтный вариант планирования включает урок, который завершается первым.

Доказательство. Пусть f – урок, который завершается первым. Предположим, что имеется оптимальный бесконфликтный вариант расписания X , который не включает f . Пусть g – урок в X , который завершается первым. Так как f завершается раньше g , урок f не может конфликтовать с любым уроком в множестве $X \setminus \{g\}$. Следовательно, вариант расписания $X' = X \cup f \setminus \{g\}$ также является бесконфликтным. Поскольку X' имеет тот же размер, что и X , X' также является оптимальным вариантом расписания. □

Чтобы завершить доказательство, снова обратимся к нашему старому другу – индукции.

Теорема 4.4. Жадное построение расписания является оптимальным способом планирования.

Доказательство. Пусть f – урок, который завершается первым, и пусть A – подмножество уроков, которые начинаются после завершения f . Предыдущая лемма 4.3 определяет, что некоторый оптимальный план содержит f , поэтому наилучший вариант расписания, содержащий f , является оптимальным расписанием. Наилучший вариант расписания, включающий f ,

обязательно должен содержать оптимальное расписание для уроков, которые не конфликтуют с f , т. е. оптимальное расписание для подмножества A . Жадный алгоритм выбирает f , а затем по индуктивному предположению вычисляет оптимальное расписание уроков из подмножества A . □

Это доказательство можно сделать более простым для понимания, если немного подробнее описать процесс индукции.

Доказательство. Пусть $\langle g_1, g_2, \dots, g_k \rangle$ – последовательность уроков, выбранная жадным алгоритмом, отсортированная по времени начала. Предположим, что имеется максимальный бесконфликтный вариант расписания

$$S = \langle g_1, g_2, \dots, g_{j-1}, c_j, c_{j+1}, \dots, c_m \rangle,$$

также отсортированный по времени начала, где c_j отличается от урока g_j , выбранного жадным алгоритмом. (Мы могли бы принять $j = 1$, в этом случае рассматриваемый здесь план начинается с не жадного выбора c_1 .) По построению j -й жадный выбор не конфликтует ни с одним предыдущим уроком $g_1, g_2, \dots, g_{j-1}$, и поэтому рассматриваемый здесь план S является бесконфликтным и не конфликтует также с c_j . Кроме того, g_j имеет самое раннее время завершения среди всех уроков, которые не конфликтуют с более ранними уроками, в частности g_j завершается раньше c_j . Из этого следует, что g_j не конфликтует с любым более поздним уроком $c_{j+1}, \dots, c_m$. Таким образом, измененное расписание

$$S' = \langle g_1, g_2, \dots, g_{j-1}, g_j, c_{j+1}, \dots, c_m \rangle$$

также является бесконфликтным. (Это утверждение является прямым обобщением леммы 4.3, при котором рассматривается случай $j = 1$.)

Теперь по принципу индукции из этого следует, что существует оптимальное расписание $\langle g_1, g_2, \dots, g_k, c_{k+1}, \dots, c_m \rangle$, включающее каждый урок, выбранный жадным алгоритмом. Но это невозможно без того, чтобы $k = n$. Если некоторый урок c_{k+1} не конфликтует ни с одним из первых k уроков, выбранных жадным алгоритмом, то жадный алгоритм должен будет выбрать более k уроков. □

4.3. Общий шаблон

Основная структура этого доказательства корректности в точности та же самая, что и для задачи сортировки файлов на магнитной ленте: индуктивный обмен местами аргументов.

- Предположим, что существует оптимальное решение, отличающееся от решения с использованием жадного алгоритма.
- Найти «первое» различие между этими двумя решениями.
- Доказать, что можно заменить оптимальный выбор на жадный выбор без ухудшения решения (хотя эта замена, возможно, не улучшит решение).

По индукции это утверждение полагает, что некоторое оптимальное решение содержит все жадное решение, следовательно, равнозначно этому жадному решению. Иногда, как в рассматриваемой выше задаче планирования, требуется дополнительный шаг для демонстрации того, что ни одно оптимальное решение в строгом смысле не улучшает решение с использованием жадного алгоритма.

4.4. Коды Хаффмана

Двоичный код присваивает строку из нулей и единиц каждому символу исходного алфавита. Двоичный код является префиксным кодом (prefix-free code), если ни одна из его комбинаций не является префиксом другой комбинации того же кода. (По непонятной причине в английском языке prefix-free-коды весьма часто называют prefix-кодами. В русском языке название одно – префиксный код.) Кодировки ASCII 7 бит и Unicode UTF-8 являются префиксными двоичными кодами. Код Морзе – двоичный код с символами • и —, но он не является префиксным кодом, потому что код буквы **Е** (••) является префиксом кодов для букв **І** (•••) и **Н** (••••).³

Любой беспрефиксный двоичный код можно представить визуально как двоичное дерево, в листьях которого хранятся закодированные символы. Кодовое слово для любого символа задается путем от корня до соответствующего листа: 0 для левой ветви, 1 для правой ветви. Таким образом, длина кодового слова любого символа равна глубине соответствующего листа в кодовом дереве. Двоичные кодовые деревья не являются двоичными деревьями поиска, хотя внешне они похожи, но в кодовом дереве порядок расположения символов в листьях абсолютно не важен.

Предположим, что необходимо закодировать сообщение, написанное с использованием n -символьного алфавита, так, чтобы закодированное сообщение было по возможности наикратчайшим. Более точная формулировка: задан массив счетчиков частот $f[1..n]$, необходимо вычислить беспрефиксный двоичный код, минимизирующий общую длину закодированного сообщения:

$$\sum_{i=1}^n f[i] \cdot depth(i).$$

Это в точности та же самая функция стоимости, которая рассматривалась для оптимизации двоичных деревьев поиска, но оптимизация – это

[illegible]

совсем другая задача, потому что в кодовых деревьях не требуется хранение ключей в каком-либо определенном порядке.

В 1951 г. студент Массачусетского технологического института (MIT) Дэвид Хаффман (David Huffman) разработал следующий жадный алгоритм для вычисления такого типа оптимального кода⁴:

Huffman: Объединить две наименее часто встречающиеся буквы и выполнить рекурсию.

Алгоритм Хаффмана наилучшим образом иллюстрирует весь пример от начала до конца. Предположим, что необходимо закодировать приведенное ниже ценное предложение, которое описывает само себя, предложенное Ли Саллоузом (Lee Sallows)⁵:

This sentence contains three a's, three c's, two d's, twenty-six e's, five f's, three g's, eight h's, thirteen i's, two l's, sixteen n's, nine o's, six r's, twenty-seven s's, twenty-two t's, two u's, five v's, eight w's, four x's, five y's, and only one z.

(Это предложение содержит три a, три c, два d, двадцать шесть e, пять f, три g, восемь h, тринадцать i, два l, шестнадцать n, девять o, шесть r, двадцать семь s, двадцать два t, два u, пять v, восемь w, четыре x, пять y и только один z.)

Для упрощения не будем учитывать сорок четыре пробела, девятнадцать апострофов, девятнадцать запятых, три дефиса и единственную точку. Закодируем только буквы, как если бы это сообщение было записано в формате непрерывной записи (*scriptio continua*):

⁴ Хаффман был студентом курса теории информации, который читал Роберт Фано (Robert Fano), близкий коллега Клода Шеннона (Claude Shannon), отца теории информации. Фано и Шеннон несколько ранее разработали другой жадный алгоритм для создания префиксных кодов – разделение массива частот на два подмассива по возможности равного размера, затем рекурсивное создание кода для каждого подмассива, – но эти коды Фано–Шеннона, как выяснилось, не были оптимальными. Фано предложил задачу поиска оптимального префиксного кода в своем учебном курсе. Хаффман хотел решить эту задачу в виде курсового проекта, а не на завершающем экзамене по курсу, не зная, что задача была не решенной (открытой) и что Фано и Шеннон уже пытались решить ее и не смогли. После нескольких месяцев бесплодных усилий Хаффман в итоге сдался и решил отложить ее до завершающего экзамена. Как только он выбросил свои записи в мусорную корзину, решение вдруг пришло само. Впоследствии Хаффман должен был описать это озарение как «абсолютно молниеносное внезапное осознание».

⁵ Это предложение впервые было опубликовано Александром Дьюдни (Alexander Dewdney) в своей колонке «Математические развлечения» в октябрьском номере журнала *Scientific American* 1984 г. Сам Саллоуз опубликовал занимательную историю своего открытия в 1985 г. вместе с несколькими другими предложениями, описывающими самих себя. Работы Саллоуза можно найти на его личном веб-сайте. Разочарованный слишком медленным развитием своего кода, выполняющегося на мини-компьютере VAX 11/780, Саллоуз спроектировал и создал специализированное аппаратное оборудование для выполнения поиска прямым перебором (грубой силой) предложений, описывающих самих себя, с различными характерными особенностями текста («This pangram has...» – «В этой панграмме...»; «This sentence contains exactly...» – «Это предложение содержит ровно...» и т. д.). Тщательный теоретический анализ ограничил пространство поиска всего лишь немного более чем 6 млрд возможных вариантов, которые компьютер Саллоуза Pangram Machine с тактовой частотой 1 МГц перечислял в течение двух часов.

THISSENTENCECONTAINSTHREEASTHREECSTWODSTWENTYSIXESFIVEFST
HREEGSEIGHTHSTHIRTEENISTWOLSSIXTEENNSNINEOSSIXRSTWENTYSEV
ENSSTWENTYTWOTSTWOUSFIVEVSEIGHTWSFOURXS FIVEYSANDONLYONEZ⁶

В табл. 4.1 приведены частоты букв в предложении Саллоуза.

Таблица 4.1

A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

Алгоритм Хаффмана выбирает две наименее часто встречающиеся буквы, а при равенстве частот выбирает букву случайным образом – в рассматриваемом здесь примере это буквы **Z** и **D** – и объединяет их в один новый символ **IX** с частотой 3. Этот новый символ становится внутренним узлом в создаваемом кодовом дереве, а **Z** и **D** – потомками этого узла, при этом неважен порядок расположения этих потомков. Затем алгоритм рекурсивно создает код Хаффмана для новой таблицы частот, показанной в табл. 4.2.

Таблица 4.2

A	C	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	IX
3	3	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	3

После 19 слияний все 20 букв объединены. Запись процедур слияний дает нам кодовое дерево. Алгоритм выполняет некоторое количество произвольных выборов, в результате чего получается действительно несколько различных кодов Хаффмана. Один из этих кодов Хаффмана показан на рис. 4.4, где числа в узлах, которые не являются листьями, представляют частоты объединенных символов. Например, код буквы **A** 101000, а код буквы **S** 111.

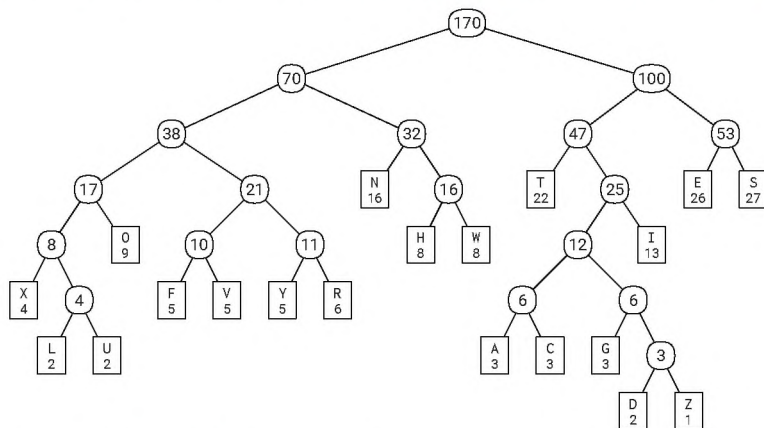


Рис. 4.4. Код Хаффмана в виде двоичного кодового дерева

⁶ ...и он говорил сорок пять минут, и никто не понял ни слова из сказанного, поэтому мы развлекались, заполняя бланки и играя с карандашами на скамье. (Строки из очень длинной песни Арло Гатри (Arlo Guthrie) из альбома Alice's Restaurant, 1967 г. — Прим. перев.)

Закодированное с помощью этого конкретного кода Хаффмана предложение Саллоуза становится строкой битов, начало которой показано на рис. 4.5.

$$\begin{array}{cccccccccccccccccccccccc} 100 & 0110 & 1011 & 111 & 111 & 110 & 010 & 100 & 110 & 010 & 101001 & 110 & 101001 & 0001 & 010 & 100 & \dots \\ \hline T & H & I & S & S & E & N & T & E & N & C & E & C & O & N & T & \dots \end{array}$$

Рис. 4.5. Начало строки Саллоуза в кодировке Хаффмана

В табл. 4.3 приведен список стоимостей кодирования каждого символа в предложении Саллоуза вместе с вкладом этих символов в общую длину закодированного предложения.

Таблица 4.3

буква	A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
частота	3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1
глубина	6	6	7	3	5	6	4	4	6	3	4	5	3	3	6	5	4	5	5	7
общая хар-ка	18	18	14	78	25	18	32	52	12	48	36	30	81	66	12	25	32	20	25	7

В сумме закодированное сообщение имеет длину 649 бит. Различные варианты кода Хаффмана кодируют одни и те же символы по-разному, возможно, кодовыми словами разной длины, но общая длина закодированного сообщения всегда остается одинаковой для каждого варианта кода Хаффмана: 649 бит.

С учетом простой структуры алгоритма Хаффмана довольно неожиданно, что он создает оптимальный беспрефиксный двоичный код.⁷ Кодирование предложения Саллоуза с использованием любого беспрефиксного кода требует не менее 649 бит. К счастью, рекурсивная структура позволяет легко доказать это утверждение, используя обмен местами аргументов, подобно тому, как это делалось в наших предыдущих доказательствах. Начнем с доказательства того, что выбранный самым первым вариант этого алгоритма корректен.

Лемма 4.5. Пусть x и y – два символа с наименьшей частотой (при одинаковой частоте двух и более символов выполняется произвольный выбор). Существует оптимальное кодовое дерево, в котором x и y являются одноуровневыми элементами («братьями»).

Доказательство. В действительности я буду доказывать более строгое утверждение: существует оптимальный код, в котором x и y являются одноуровневыми элементами и имеют максимальную глубину любого листа.

Пусть T – оптимальное кодовое дерево. Предположим, что это дерево имеет глубину d . Поскольку T является полным двоичным деревом, в нем

⁷ Это стало весьма неожиданным и для Хаффмана, и для Фано.

существуют по меньшей мере два листа на глубине d , которые являются одноуровневыми элементами. (Подтвердите это утверждение методом индукции.) Предположим, что этими двумя листьями являются не x и y , а некоторые другие символы a и b .

Пусть T' – кодовое дерево, полученное в результате обмена местами x и a , и пусть $\Delta = d - \text{depth}_T(x)$. Этот обмен местами увеличивает глубину x на Δ и уменьшает глубину a на Δ , следовательно:

$$\text{cost}(T') = \text{cost}(T) + \Delta \cdot (f[x] - f[a]).$$

Наше предположение о том, что x , но не a является одним из двух символов с наименьшей частотой, подразумевает $f[x] \leq f[a]$, а наше предположение о том, что a имеет максимальную глубину, подразумевает $\Delta \geq 0$. Из этого следует, что $\text{cost}(T') \leq \text{cost}(T)$. С другой стороны, T – оптимальное кодовое дерево, поэтому мы также непременно имеем $\text{cost}(T') \geq \text{cost}(T)$. Таким образом, мы приходим к заключению: T' также является оптимальным кодовым деревом.

Аналогично обмен местами y и b обязательно дает в результате еще одно оптимальное кодовое дерево. В этом итоговом оптимальном кодовом дереве x и y являются одноуровневыми элементами с максимальной глубиной, как требуется по условию. $\square$

Теперь оптимальный вариант обеспечивается нашим лучшим другом – Феей Рекурсией. Наш рекурсивный аргумент основан на следующем нестандартном рекурсивном определении: полное двоичное дерево является либо единственным узлом, либо полным двоичным деревом, в котором некоторый лист заменен внутренним узлом с двумя листьями-потомками.

Теорема 4.6. Каждый код Хаффмана является оптимальным префиксным двоичным кодом.

Доказательство. Если в сообщении содержится только один или два различных символа, то теорема тривиальна, поэтому рассмотрим другой случай.

Пусть $f[1..n]$ – входной массив изначально заданных частот символов, и предположим без ущерба для обобщения, что $f[1]$ и $f[2]$ – наименьшие два значения частоты. Для формулирования рекурсивной подзадачи определим $f[n+1] = f[1] + f[2]$. Применяемый выше метод обмена местами аргументов подразумевает, что 1 и 2 – (самые глубокие) одноуровневые элементы в некотором оптимальном коде для $f[1..n]$.

Пусть T' – дерево Хаффмана для $[3..n+1]$, тогда индуктивное предположение подразумевает, что T' – оптимальное кодовое дерево для меньшего множества частот. Для получения итогового кодового дерева T заменим лист с меткой $n+1$ внутренним узлом с двумя потомками, помеченными 1 и 2. Я утверждаю, что дерево T является оптимальным для исходного массива частот $f[1..n]$.

Для доказательства этого утверждения можно выразить стоимость дерева T через стоимость дерева T' , как показано ниже. (В приведенных ниже равенствах $depth(i)$ обозначает глубину листа с меткой i в дереве T или T' , при этом каждый узел, существующий в обоих деревьях T и T' , имеет в них одинаковую глубину.)

$$\begin{aligned}
 cost(T) &= \sum_{i=1}^n f[i] \cdot depth(i) = \\
 &= \sum_{i=3}^{n+1} f[i] \cdot depth(i) + f[1] \cdot depth(1) + f[2] \cdot depth(2) - f[n+1] \cdot depth(n+1) = \\
 &= cost(T') + (f[1] + f[2]) \cdot depth(T) - f[n+1] \cdot (depth(T) - 1) = \\
 &= cost(T') + f[1] + f[2] + (f[1] + f[2] - f[n+1]) \cdot (depth(T) - 1) = \\
 &= cost(T') + f[1] + f[2].
 \end{aligned}$$

Это равенство подразумевает, что минимизация стоимости дерева T равнозначна минимизации стоимости дерева T' , в частности добавление листьев, помеченных 1 и 2, к листу в дереве T' , помеченному $n + 1$, создает оптимальное кодовое дерево для исходного массива частот. $\square$

```

BuildHuffman(f[1..n]):
  for i ← 1 to n
    L[i] ← 0; R[i] ← 0
    Insert(i, f[i])
  for i ← n to 2n - 1
    x ← ExtractMin()      <<find two rarest symbols>>
    y ← ExtractMin()
    f[i] ← f[x] + f[y]    <<merge into a new symbol>>
    Insert(i, f[i])
    L[i] ← x; P[x] ← i    <<update tree pointers>>
    R[i] ← y; P[y] ← i
  P[2n - 1] ← 0
  
```

Рис. 4.6. Создание кода Хаффмана

Для эффективного создания кода Хаффмана мы сохраняем символы в очереди с приоритетами, используя в качестве приоритетов частоты символов. Можно представить кодовое дерево как три массива индексов, перечисляющих левых (Left) и правых (Right) потомков и родителя (Parent) каждого узла. Листья итогового кодового дерева – это узлы с индексами от 1 до n , а корень – узел с индексом $2n - 1$. Псевдокод для этого алгоритма показан на рис. 4.6. Алгоритм BuildHuffman выполняет $O(n)$ операций очереди с приоритетами: ровно $2n - 1$ операций вставки Insert и $2n - 2$ операций

ExtractMins. Если реализовать очередь с приоритетами как стандартный двоичный динамический массив («кучу»), то каждая из этих операций требует времени $O(\log n)$, следовательно, весь алгоритм в целом выполняется за время $O(n \log n)$.

Наконец, простые алгоритмы кодирования и декодирования сообщений с использованием четко заданного кода Хаффмана показаны на рис. 4.7. Оба алгоритма выполняются за время $O(m)$, где m – длина закодированного сообщения.

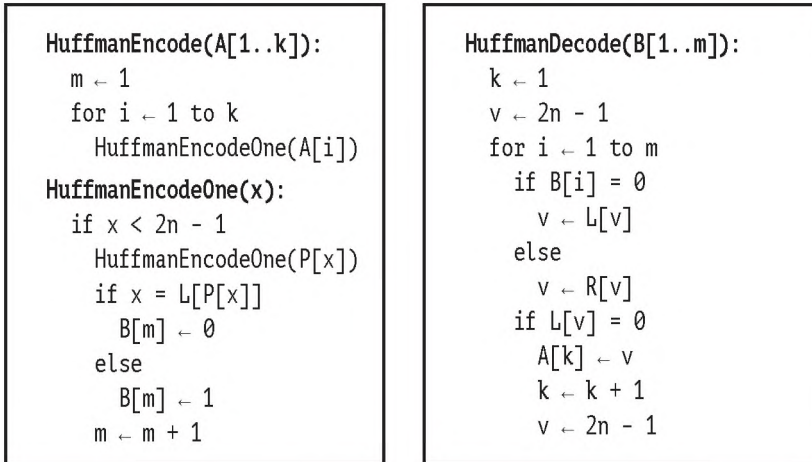


Рис. 4.7. Алгоритмы кодирования и декодирования с использованием кодов Хаффмана

4.5. Задача о стабильных браках

Каждый год тысячи новых врачей обязательно должны получить интернатуру (стажировку) в медицинских (лечебных) учреждениях США. В течение первой половины XX в. конкуренция между медицинскими учреждениями за привлечение самых лучших врачей привела ко все более ранним («упреждающим») предложениям интернатуры, иногда даже во время второго года обучения в медицинском университете (на медицинском факультете) в сочетании с более сжатыми сроками принятия решения. В 1940-х гг. медицинские университеты (факультеты) пришли к общему соглашению о неразглашении информации до общепринятой даты, назначенной в течение четвертого года обучения своих студентов. В ответ медицинские учреждения начали требовать более быстрого принятия решений. К 1950 г. медицинские учреждения должны были регулярно обращаться с требованиями о направлении к ним врачей, предоставлять им интернатуру и запрашивать немедленные ответы. Интерны были вынуждены рисковать, если медицинское учреждение третьего выбора обратилось первым – принять предложение с риском потерять более выгодную возможность позже

или отвергнуть предложение с риском вообще не получить никакого рабочего места.⁸

Наконец, центр координации и обмена информацией о назначениях интернатуры, который сейчас называется National Resident Matching Program (NRMP), был создан в начале 1950-х гг. Каждый год врачи-выпускники получают рейтинговый список всех медицинских учреждений, в которые они могли бы быть приняты как интерны. Затем NRMP вычисляет паросочетания между врачами-выпускниками и медицинскими учреждениями; в этих паросочетаниях должно выполняться следующее требование стабильности. Паросочетание является нестабильным (unstable), если существует врач-выпускник α и медицинское учреждение B , которые в большей степени довольны друг другом, чем их текущее соответствие, т. е.:

- для α существует паросочетание с некоторым другим медицинским учреждением A , даже если α предпочитает B ;
- для B существует паросочетание с некоторым другим врачом-выпускником β , даже если B предпочитает α .

В этом случае мы называем (α, B) нестабильной парой для данного паросочетания. Целью NRMP является определение стабильного соответствия (stable matching), которое представляет собой соответствие без нестабильных пар.

Для упрощения с этого момента я предполагаю, что существует абсолютно одинаковое количество врачей-выпускников и медицинских учреждений, каждое медицинское учреждение предлагает ровно одну вакансию интерна, для каждого врача-выпускника предоставлен рейтинг всех медицинских учреждений и наоборот, наконец, не существует никаких связей в рейтинговых списках врачей или медицинских учреждений.⁹

Некоторые неудачные идеи

На первый взгляд даже не очевидно, что стабильное соответствие существует всегда. Разумеется, не каждое соответствие врачей и медицинских учреждений является стабильным. Предположим, что существует три врача (доктора Куинси (Dr. Quincy), Ротванг (Dr. Rotwang), Шепард (Dr. Shep-

⁸ Образовательный (академический) рынок труда в США подразумевает такие же риски, по крайней мере, в области информационных технологий. Некоторые организации начинают рассылать предложения в феврале с двухнедельным сроком принятия решения. Другие организации даже не начинают собеседования до марта. Массачусетский технологический институт (MIT), как известно, ждет мая, когда все собеседования MIT завершаются, прежде чем делать какие-либо предложения по преподавательским должностям. Само собой, путаница с датами предложений и сроками принятия решений приводит к весьма сильным стрессам как для кандидатов, так и для самих организаций. По схожим причинам с 1965 г. большинство американских университетов пришло к общему соглашению о том, что 15 апреля следует считать конечным сроком для приема будущими выпускниками предложений о финансовой поддержке (и дополнительных предложений о возможности приема на работу).

⁹ В действительности большинство медицинских учреждений предлагают несколько вакансий интернов, каждый врач-выпускник оценивает рейтинг только некоторого подмножества медицинских учреждений и наоборот. Кроме того, обычно существует больше вакансий интернов, чем заинтересованных в них врачей-выпускников. Из-за этого задача начинает усложняться.

hard), представленные буквами нижнего регистра) и три больницы (Arkham Asylum, Bethlem Royal Hospital и County General Hospital, представленные буквами верхнего регистра), рейтинги которых по отношению друг к другу приведены в табл. 4.4 и 4.5.

Таблица 4.4

<i>q</i>	<i>r</i>	<i>s</i>
<i>A</i>	<i>C</i>	<i>A</i>
<i>C</i>	<i>A</i>	<i>B</i>
<i>B</i>	<i>B</i>	<i>C</i>

Таблица 4.5

<i>A</i>	<i>B</i>	<i>C</i>
<i>r</i>	<i>s</i>	<i>q</i>
<i>q</i>	<i>q</i>	<i>r</i>
<i>s</i>	<i>r</i>	<i>s</i>

Соответствие $\{Aq, Br, Cs\}$ является нестабильным, потому что Arkham предпочитает нанять д-ра Ротванга, а не д-ра Куинси и сам д-р Ротванг предпочитает работать в Arkham, а не в Bethlem. В этом совпадении (A, r) является нестабильной парой.

Можно предположить применение инкрементального алгоритма, начинающего работу с произвольного соответствия, а затем жадно выполняющего перестановки для разрешения нестабильностей. К сожалению, разрешение одной нестабильности может прождать новые, и в действительности такое инкрементальное «улучшение» может привести к бесконечному циклу. Например, если начать с приведенного выше нестабильного соответствия $\{Aq, Br, Cs\}$, то каждая из следующих операций перестановки устраняет одну нестабильную пару (указанную над стрелкой), но последовательность таких перестановок приводит опять к исходному соответствию¹⁰:

$$\{Aq, Br, Cs\} \xrightarrow{Ar} \{Ar, Bq, Cs\} \xrightarrow{Cr} \{As, Bq, Cr\} \xrightarrow{Cq} \{As, Br, Cq\} \xrightarrow{Aq} \{Aq, Br, Cs\}.$$

При другом подходе можно попробовать применить следующий многоэтапный жадный протокол. На каждом этапе любая больница без установленного соответствия делает предложение наиболее предпочитаемому врачу, для которого также не установлено соответствие, затем каждый врач без пары, но с предложениями принимает наиболее предпочтительное для него предложение. Нетрудно доказать, что как минимум для одной новой пары врач–больница на каждом этапе будет установлено соответствие, поэтому алгоритм всегда завершается с установлением всех соответствий. Для входных данных рассмотренного выше примера стабильное соответствие $\{Ar, Bs, Cq\}$ получается уже в конце первого этапа. Но давайте рассмотрим другие входные данные, приведенные в табл. 4.6 и 4.7.

Таблица 4.6

<i>q</i>	<i>r</i>	<i>s</i>
<i>C</i>	<i>A</i>	<i>A</i>
<i>B</i>	<i>C</i>	<i>B</i>
<i>A</i>	<i>B</i>	<i>C</i>

Таблица 4.7

<i>A</i>	<i>B</i>	<i>C</i>
<i>q</i>	<i>q</i>	<i>s</i>
<i>s</i>	<i>r</i>	<i>r</i>
<i>r</i>	<i>s</i>	<i>q</i>

¹⁰ Этот пример придумал Дональд Кнут (Donald Knuth).

На первом этапе д-р Шепард принимает предложение от County, а д-р Куинси принимает предложение от Bethlem (отвергая предложение от Arkham), оставляя только д-ра Ротванга и больницу Arkham без совпадений. Таким образом, этот протокол завершается с соответствием $\{Ar, Bq, Cs\}$ после двух этапов. К сожалению, это соответствие нестабильно, так как и больница Arkham, и д-р Ротванг предпочитают другие соответствия.

Алгоритмы Boston Pool и Гэйла–Шепли

В 1952 г. координационный центр NRMP принял алгоритм Boston Pool для распределения интернов, названный так потому, что ранее он использовался региональным координационным центром в районе Бостона. Десять лет спустя Дэвид Гэйл (David Gale) и Ллойд Шепли (Lloyd Shapley) описали и формально проанализировали обобщение алгоритма Boston Pool и доказали, что этот алгоритм вычисляет стабильное соответствие. Гэйл и Шепли использовали метафору приема в колледж. По существу тот же алгоритм был независимо разработан Эллиоттом Перансоном (Elliott Peranson) в 1972 г. и использовался для зачисления в медицинские учебные заведения. Подобные алгоритмы несколько позже были адаптированы для многих других практических задач подбора соответствий, включая подбор преподавательских кадров во Франции, найм выпускников с дипломами доктора философии по экономике в США, зачисление в университеты в Германии, прием в бесплатные средние школы в Нью-Йорке и Бостоне, размещение личного состава ВМФ США и программы подбора паросочетаний по складу характера.

В 2012 г. Шепли получил Нобелевскую премию в области экономики за исследования стабильных соответствий вместе с Элвином Ротом (Alvin Roth), который существенно расширил работу Шепли и использовал ее для разработки нескольких практических методов финансовых расчетов («теория стабильного распределения и практика устройства рынков»). (Гэйл не был включен в число Нобелевских лауреатов, так как умер в 2008 г.)

Как и рассматриваемый в предыдущем разделе неправильный жадный алгоритм, алгоритм Гэйла–Шепли выполняется поэтапно до тех пор, пока не будет распределена каждая вакансия. Каждый этап состоит из двух стадий.

1. Произвольно выбранная больница без пары A предлагает свою вакансию самому лучшему врачу α (в соответствии со списком предпочтений A), который пока еще не отверг это предложение.
2. Если врач α не имеет пары, то он/она (предположительно) принимает предложение A . Если у α уже есть пара, но он/она предпочитает A , то α отменяет текущее паросочетание и (предположительно) принимает новое предложение от A . Иначе α отвергает новое предложение.

Каждый врач в итоге принимает наилучшее предложение из всех полученных в соответствии с его/ее списком предпочтений.¹¹ Если говорить кратко, больницы делают предложения жадным способом и врачи принимают предложения в жадном режиме. Возможность врача отказаться от текущей пары в пользу более выгодного предложения является ключом к успешной работе этой взаимно жадной стратегии.

Например, предположим, что существует четыре врача (доктора Куинси (Dr. Quincy), Ротванг (Dr. Rotwang), Шепард (Dr. Shephard) и Там (Dr. Tam)) и четыре больницы (Arkham Asylum, Bethlem Royal Hospital, County General Hospital и The Dharma Initiative), которые назначают рейтинги друг для друга, как показано в табл. 4.8 и 4.9.

Таблица 4.8

q	r	s
A	A	B
B	D	A
C	C	C
D	B	D

Таблица 4.9

A	B	C
t	r	t
s	t	r
r	q	s
q	s	q

Принимая эти списки предпочтений как входные данные, алгоритм Гэйла–Шепли может продолжить работу, как показано ниже.

1. Arkham делает предложение д-ру Таму.
2. Bethlem делает предложение д-ру Ротвангу.
3. County делает предложение д-ру Таму, который отказывается от предыдущего предложения от Arkham.
4. Dharma делает предложение д-ру Шепарду. (С этого момента остается только одна больница без пары, поэтому у алгоритма больше нет вариантов выбора.)
5. Arkham делает предложение д-ру Шепарду, который отказывается от предыдущего предложения от Dharma.
6. Dharma делает предложение д-ру Ротвангу, который отказывается от предыдущего предложения от Bethlem.
7. Bethlem делает предложение д-ру Таму, который отказывается от предыдущего предложения от County.
8. County делает предложение д-ру Ротвангу, который отвергает его.
9. County делает предложение д-ру Шепарду, который отвергает его.
10. County делает предложение д-ру Куинси.

¹¹ В 1952 г. алгоритм Boston Pool представлял собой особый случай алгоритма Гэйла–Шепли, в котором предложения обрабатываются в некотором установленном порядке. Грубо говоря, каждое предложение выполняется больницей X , а наиболее предпочтительный для нее врач (из тех, которые пока еще не отвергли предложение X) присваивал X наивысший рейтинг. Поскольку порядок предложений зависит от полного множества списков предпочтений, этот алгоритм обязательно должен выполняться центральным руководящим органом. В противоположность ему алгоритм Гэйла–Шепли не требует даже того, чтобы каждый участник заранее знал свои предпочтения, пока они ведут себя согласованно с некоторым постоянным списком рейтингов.

После десятого этапа все рассматриваемые предложения приняты, и алгоритм возвращает паросочетание $\{As, Bt, Cq, Dr\}$. Вы можете (и должны) проверить методом прямого перебора, что это паросочетание является стабильным, даже если нет врачей, принятых в предпочитаемую больницу, и нет больниц, принявших предпочитаемого врача. В действительности больница County завершила процедуру приема, выбрав наименее предпочитаемого врача. Это не единственное стабильное паросочетание для приведенных выше списков предпочтений, паросочетание $\{Ar, Bs, Cq, Dt\}$ также является стабильным.

Время выполнения

Анализ количества предложений, выполненных алгоритмом Гэйла–Шепли, относительно прост (именно поэтому мы рассматриваем его в первую очередь). Каждая больница делает предложение каждому врачу не более одного раза, поэтому алгоритм выполняет не более n^2 предложений.

Но для анализа истинного времени выполнения требуется более подробная спецификация этого алгоритма. Как содержимое списков предпочтений передается в алгоритм? Как алгоритм определяет, остается ли какая-либо больница без пары, и если так, то как алгоритм находит больницу без пары? Как алгоритм хранит предполагаемые пары в паросочетании? Как алгоритм определяет, предпочитает ли врач новое предложение и отказывается от текущей пары? По существу дела (в максимально формализованном виде): как в алгоритме в действительности представлены врачи и больницы?

Один из возможных вариантов: представление каждого врача и каждой больницы неповторяющимся целым числом от 1 до n , а для предпочтений формируются два массива $Dpref[1..n, 1..n]$ и $Hpref[1..n, 1..n]$, где $Dpref[i, r]$ представляет r -ю больницу в списке предпочтений врача i , а $Hpref[j, r]$ представляет r -го врача в списке предпочтений больницы j . При входных данных в такой форме алгоритм Boston Pool может выполнить каждое предложение за постоянное время после некоторой начальной предварительной обработки, а полная реализация выполняется за время $O(n^2)$.

Немного более трудное задание – доказать, что существуют входные данные (и варианты выбора: кто и когда делает предложения), при которых алгоритм вынужден сделать $\Omega(n^2)$ предложений, прежде чем завершит работу. Таким образом, верхняя граница времени выполнения $O(n^2)$ в наихудшем случае является жестко фиксированной.

Корректность

Но почему этот алгоритм вообще корректен? Откуда нам известно, что он всегда вычисляет стабильное соответствие или любое полное соответствие, если уж на то пошло?

Как только врач получает предложение, он/она включается как минимум в предполагаемую пару на все остальное время. А если какой-либо врач не

имеет пары, то ни одна из больниц не предложила ему работу, т. е. подразумевается, что больницы не исчерпали свои списки предпочтений. Из этого следует, что, когда алгоритм завершается (после не более n^2 этапов), для каждого врача определена пара, следовательно, каждая вакансия занята. Другими словами, этот алгоритм всегда вычисляет идеальное соответствие между врачами и больницами. (Ого!) Остается только доказать, что итоговое соответствие является стабильным.

Оптимальность

Удивительно, но корректность алгоритма Гэйла–Шепли не зависит от того, какая больница делает предложение на каждом этапе. Действительно, вне зависимости от того, какая больница без пары делает предложение на каждом этапе, алгоритм всегда вычисляет одно и то же соответствие. Будем считать, что α – подходящий врач для больницы A , если существует стабильное соответствие, распределяющее врача α в больницу A .

Лемма 4.7. При выполнении алгоритма Гэйла–Шепли предложение каждой больницы A отвергается только теми врачами, которые не подходят для больницы A .

Доказательство. Докажем эту лемму методом индукции по количеству этапов. Рассмотрим произвольный этап алгоритма, на котором врач α отказывается от предложения больницы A в пользу другой больницы B . Отказ предполагает, что α предпочитает больницу B больнице A . Каждый врач, располагающийся выше α в списке предпочтений больницы B , уже отказался от предложения B на предыдущем этапе, следовательно, по индуктивному предположению все отказавшиеся врачи не подходят для B .

Теперь рассмотрим произвольное соответствие (включающее тех же врачей и те же больницы), которое распределяет α в больницу A . Мы уже установили, что врач α предпочитает больницу B больнице A . Если больница B предпочитает врача α его коллеге, то соответствие является нестабильным. С другой стороны, если больница B предпочитает принять коллегу вместо врача α , то (по нашему предыдущему утверждению) этот коллега не подходит, и снова соответствие является нестабильным. Мы приходим к следующему заключению: не существует стабильного соответствия, которое распределяет врача α в больницу A . $\square$

Теперь пусть $best(A)$ обозначает подходящего врача с наивысшим рейтингом в списке предпочтений больницы A . Лемма 4.7 подразумевает, что каждый врач, которого предпочитает больница A в своей итоговой паре, является не подходящим для A . С другой стороны, итоговая пара является стабильной, поэтому врач, распределенный в больницу A , непременно должен быть подходящим для A . Тогда сразу же получаем следующий результат.

Следствие 4.8. Алгоритм Гэйла–Шепли формирует пару $best(A)$ с A для каждой больницы A .

Другими словами, алгоритм Гэйла–Шепли вычисляет наилучшее возможное стабильное соответствие с точки зрения больниц. Оказывается, что это соответствие также является наихудшим из возможных, с точки зрения врачей. Пусть $worst(\alpha)$ обозначает подходящую больницу с самым низким рейтингом в списке предпочтений врача α .

Следствие 4.9. Алгоритм Гэйла–Шепли формирует пару α с $worst(\alpha)$ для каждого врача α .

Доказательство. Предположим, что алгоритм Гэйла–Шепли распределил врача α в больницу A , и необходимо показать, что $A = worst(\alpha)$. Рассмотрим произвольное стабильное соответствие, в котором больница A находится не в паре с α , а в паре с другим врачом β . Предыдущее следствие 4.8 подразумевает, что A предпочитает принять врача $\alpha = best(A)$, а не врача β . Поскольку это соответствие стабильно, врач α обязательно должен отдавать предпочтение больнице, в которую он распределен, а не больнице A . Это положение выполняется для любого стабильного соответствия, поэтому α предпочитает любую другую подходящую пару, но не A . Другими словами, $A = worst(\alpha)$. $\square$

Остроумный вывод из этих двух следствий, сформулированный Лестером Дабинсом (Lester Dubins) и Дэвидом Фридманом (David Freedman) в 1981 г., состоит в том, что врач может потенциально улучшить свою пару, солгав о своих предпочтениях, но больница не имеет такой возможности. (Тем не менее множество больниц может вступить в тайный сговор, чтобы улучшить некоторые из пар.) Отчасти по этой причине координационный центр NRMP в 1998 г. изменил на обратное действие алгоритма поиска соответствий, чтобы потенциальные кандидаты предлагали свои услуги больницам в соответствии с оформленными официально собственными предпочтениями, при этом каждая больница принимает наилучшее для себя предложение. Но на практике такое обратное действие изменило менее 1 % пар с кандидатами. Насколько мне известно, точное определение степени воздействия этого изменения на пациентов остается нерешенной задачей.

Упражнения

Предупреждение преподавателю: некоторые из предлагаемых здесь упражнений невозможно решить с использованием жадных алгоритмов.

После описания и анализа жадного алгоритма необходимо обязательно добавить еще и доказательство корректности этого алгоритма. Это доказательство обычно принимает форму обмена местами (перестановки) аргументов. Такие доказательства особенно важны на занятиях в аудитории (как на моих занятиях), на которых обычно не требуются доказательства корректности.

1. Алгоритм GreedySchedule, описанный для класса задач планирования, – это не единственная жадная стратегия, которую мы могли бы попробовать. Для каждой из описанных ниже альтернативных жадных стратегий требуется либо доказать, что итоговый алгоритм всегда создает оптимальный план, либо описать небольшой пример входных данных, для которых алгоритм не создает оптимальный план. Предположить, что все алгоритмы разрешают неоднозначность произвольно (т. е. способом, которым вы не можете управлять абсолютно). [Подсказка: три алгоритма из описанных ниже действительно корректны.]
 - (a) Выбрать курс x , который завершается последним, отбросить все курсы, конфликтующие с x , и применить рекурсию.
 - (b) Выбрать курс x , который начинается первым, отбросить все курсы, конфликтующие с x , и применить рекурсию.
 - (c) Выбрать курс x , который начинается последним, отбросить все курсы, конфликтующие с x , и применить рекурсию.
 - (d) Выбрать курс x с самой короткой продолжительностью, отбросить все курсы, конфликтующие с x , и применить рекурсию.
 - (e) Выбрать курс x , который конфликтует с наименьшим числом других курсов, отбросить все курсы, конфликтующие с x , и применить рекурсию.
 - (f) Если нет конфликтующих курсов, то выбрать все курсы. Иначе отбросить курс с наибольшей продолжительностью и применить рекурсию.
 - (g) Если нет конфликтующих курсов, то выбрать все курсы. Иначе отбросить курс, конфликтующий с наибольшим числом других курсов и применить рекурсию.
 - (h) Пусть x – курс с самым ранним временем начала, и пусть y – курс с вторым ранним временем начала.
 - Если x и y не пересекаются, то выбрать x и применить рекурсию для всех курсов, кроме x .
 - Если x полностью содержит (перекрывает) y , то отбросить y и выполнить рекурсию.
 - Иначе отбросить y и выполнить рекурсию.
 - (i) Если любой курс x полностью содержит (перекрывает) другой курс, то отбросить x и выполнить рекурсию. Иначе выбрать курс y , который завершается последним, отбросить все курсы, конфликтующие с y , и выполнить рекурсию.
2. Теперь рассмотрим версию задачи планирования учебных курсов с весами, в которой различные курсы предлагают разное количество

зачетных часов (абсолютно не связанных с продолжительностью лекций по курсам). Теперь ваша цель – выбрать множество неконфликтующих курсов, которые обеспечат максимальное возможное число зачетных часов. Входные данные: массивы времени начала и окончания курсов и соответствующее число зачетных часов.

- (a) Доказать, что жадный алгоритм, описанный в начале этой главы, – выбор курса, завершающегося первым, и применение рекурсии, – не всегда возвращает оптимальный план.
 - (b) Доказать, что ни один из жадных алгоритмов, описанных в упражнении 1, никогда не возвращает оптимальный план. [Подсказка: сначала необходимо решить упражнение 1, так как алгоритмы, не работающие там, не работают и здесь.]
 - (c) Описать и проанализировать алгоритм, который всегда вычисляет оптимальный план. [Подсказка: этот алгоритм не должен быть жадным.]
3. Пусть X – множество n отрезков на оси. Мы говорим, что подмножество отрезков $Y \subseteq X$ покрывает X , если объединение всех отрезков в подмножестве Y равно объединению всех отрезков в множестве X . Размер покрытия – это просто число отрезков.
- Описать и проанализировать эффективный алгоритм вычисления минимального покрытия X . Предположить, что входные данные состоят из двух массивов $L[1..n]$ и $R[1..n]$, представляющих левые и правые конечные точки отрезков в множестве X . Если вы используете жадный алгоритм, то обязательно должны доказать его корректность.

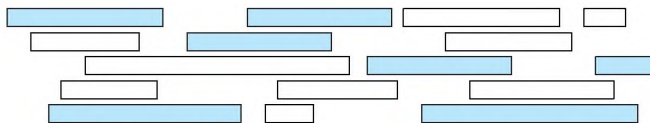


Рис. 4.8. Множество отрезков с покрытием (закрашенным голубым цветом) размером 7

4. Пусть X – множество n отрезков на оси. Мы говорим, что множество точек P пронизывает (прокалывает) X , если каждый отрезок в X содержит хотя бы одну точку из множества P . Описать и проанализировать эффективный алгоритм вычисления минимального множества точек, пронизывающего X . Предположить, что входные данные состоят из двух массивов $L[1..n]$ и $R[1..n]$, представляющих левые и правые конечные точки отрезков в множестве X . Если вы используете жадный алгоритм, то обязательно должны доказать его корректность.

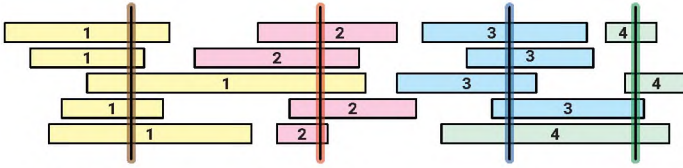


Рис. 4.9. Множество отрезков, пронизанных (проколотых) четырьмя точками (показанными в виде вертикальных прямых)

5. Пусть X – множество n отрезков на оси. Процедура правильной раскраски X присваивает цвет каждому отрезку так, чтобы любые два пересекающихся отрезка имели различные цвета. Описать и проанализировать эффективный алгоритм вычисления минимального числа цветов, необходимых для правильной раскраски X . Предположить, что входные данные состоят из двух массивов $L[1..n]$ и $R[1..n]$, представляющих левые и правые конечные точки отрезков в множестве X . И как обычно, если вы используете жадный алгоритм, то обязательно должны доказать его корректность.

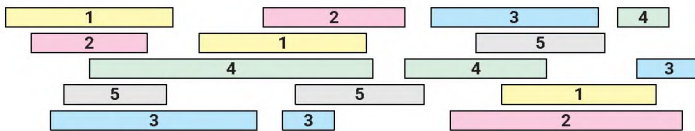


Рис. 4.10. Правильная раскраска множества отрезков с использованием пяти цветов

6. (a) Для каждого целого числа n найти массив частот $f[1..n]$, дерево кода Хаффмана которого имеет глубину $n - 1$, при этом наибольшая частота должна быть минимальной из возможных.
- (b) Предположим, что общая длина N незакодированного сообщения связана полиномиальной зависимостью с алфавитом размером n . Доказать, что любое дерево кода Хаффмана для частот $f[1..n]$ имеет глубину $O(\log n)$.

- ♥7. Массив частот $f[1..n]$ называют α -сложным (α -heavy), если он соответствует двум условиям:

- $f[1] > f[i]$ для всех $i > 1$, т. е. 1 – неповторяющийся наиболее часто встречающийся символ;
- $f[1] \geq \alpha \sum_{i=1}^n f[i]$, т. е. как минимум α символов из всех являются единицами.

Найти максимальное действительное число α , такое что в каждом коде Хаффмана для каждого α -сложного массива частот символ 1 представлен одним битом. [Подсказка: сначала необходимо доказать, что $1/3 \leq \alpha \leq 1/2$.]

8. Описать и проанализировать алгоритм, вычисляющий оптимальный троичный префиксный код для заданного массива частот $f[1..n]$. Не забудьте доказать, что полученный алгоритм корректен для всех n .
9. Подробно описать реализацию алгоритма Гэйла–Шепли поиска стабильных паросочетаний, такую что в наихудшем случае время выполнения равно $O(n^2)$, как утверждалось выше в этой главе.
10. (a) Доказать, что для алгоритма Гэйла–Шепли возможно выполнение $\Omega(n^2)$ предложений до его завершения. (Необходимо описать и подходящие входные данные, и последовательность из $\Omega(n^2)$ допустимых предложений.)
 (b) Для любого целого числа n описать множество предпочтительных вариантов для n врачей и n больниц, которое вынуждает алгоритм Гэйла–Шепли выполнить $\Omega(n^2)$ этапов независимо от того, какое допустимое предложение делается на каждом этапе. [Подсказка: решение части (b) подразумевает использование решения части (a).]
11. Описать и проанализировать эффективный алгоритм, определяющий, возможно ли по заданному множеству предпочтений больниц и врачей получить строго определенное стабильное соответствие.
12. Рассмотреть обобщение задачи поиска стабильного соответствия, в которой некоторые врачи не присваивают рейтинг всем больницам, а некоторые больницы не присваивают рейтинг всем врачам. Врач может быть распределен в больницу, только если они присутствуют в списках предпочтений друг друга. В этом случае существуют три дополнительные нестабильные ситуации:
 - больница в существующей паре предпочитает врача без пары уже распределенному в нее врачу;
 - врач в существующей паре предпочитает больницу без пары больнице, в которую он уже распределен;
 - нераспределенный врач и больница без пары присутствуют в списках предпочтений друг друга.

При таких условиях стабильное соответствие может оставить некоторых врачей и/или некоторые больницы без пары, даже если их списки предпочтений не пусты. Например, если каждый врач укажет Harvard как свой единственный приемлемый вариант выбора больницы, а каждая больница определит д-ра Хауса как единственного приемлемого интерна, то в пару будут объединены только Хаус и Harvard.

Описать и проанализировать эффективный алгоритм вычисления стабильного соответствия при описанных выше обобщенных усло-

виях. [Подсказка: необходимо свести эту задачу к случаю, в котором каждый врач присваивает рейтинг каждой больнице, затем вызвать алгоритм Гэйла–Шепли.]

13. Скандинавская мебельная компания F rni наняла n водителей для доставки n одинаковых заказов по n различным адресам в Уилмингтоне (шт. Делавар). У каждого водителя есть собственный четко определенный маршрут доставки в Уилмингтоне с посещением всех n адресов. Предположим, что водители, как всегда, следуют по своим маршрутам, и два водителя никогда не посещают один адрес одновременно.

Теоретически каждый из n водителей может доставить мебель по любому из n адресов, но при этом возникает некоторая сложность. Один из водителей тайно подключил бесконтактные датчики обнаружения и взрывчатку к диванам Johannshamn (с зеленым рисунком в полоску Strinne). Если два дивана вдруг окажутся по одному адресу одновременно, то оба взорвутся, уничтожив и грузовик, и здание по этому адресу. Это может случиться, только если водитель доставит заказ по некоторому адресу, а немного позже другой водитель посетит тот же адрес, при этом мебель пока еще остается в их грузовиках.

Ваша задача как диспетчера компании F rni – назначить каждому водителю адреса доставки. Описать алгоритм назначения адресов для водителей, такой чтобы каждый из n адресов получил мебель по своему заказу без взрывов.

Например, предположим, что маршрут Джека определяет посещение адреса 537 Paper Street в 6 часов вечера и адреса 1888 Franklin Street в 8 часов вечера, а по маршруту Марлы запланировано посещение 537 Paper в 7 часов вечера и 1888 Franklin в 9 часов вечера. Следовательно, Джек должен доставить заказ по адресу 1888 Franklin, а Марла – по адресу 537 Paper, иначе в 8 часов вечера по адресу 1888 Franklin произойдет взрыв. (Слушайте The Pixies.) [Подсказка: Джек и Марла немного непунктуальны.]

14. Предположим, что вы простой владелец магазина, проживающий в стране с n различными типами монет с номиналами $1 = c[1] < c[2] < \dots < c[n]$. (Например, в США $n = 6$, а номиналы: 1, 5, 10, 25, 50 и 100 центов.) Ваш любимый и великодушный диктатор El Generalissimo издал указ, по которому при возврате сдачи покупателю вы обязаны использовать наименьшее возможное число монет, чтобы не затереть портрет El Generalissimo, тщательно выгравированный на каждой монете сотрудниками Королевского монетного двора.

(а) В США существует простой жадный алгоритм, который всегда возвращает минимальное число монет: вычитание монеты с наибольшим номиналом и рекурсивный возврат сдачи для остатка.

El Generalissimo не одобряет жадность американских капиталистов. Показать, что существует множество номиналов монет, для которых жадный алгоритм не всегда возвращает минимальное возможное число монет.

- (b) Теперь предположим, что El Generalissimo решил ввести систему денежного обращения, в которой достоинствами монет являются последовательные степени $b^0, b^1, b^2, \dots, b^k$ некоторого целого числа $b \geq 2$. Доказать, что, несмотря на неодобрение El Generalissimo, жадный алгоритм, описанный в части (a), набирает оптимальный вариант сдачи в этой системе денежного обращения.
- (c) Описать и проанализировать эффективный алгоритм, определяющий при заданной целевой сумме T и отсортированном массиве номиналов монет $c[1..n]$ минимальное число монет, необходимое для того, чтобы набрать T центов сдачи. Предположить, что $c[1] = 1$, чтобы существовала возможность набрать сдачу, равную любой сумме T .
15. Предположим, что задан массив целых чисел $A[1..n]$, и каждое из этих чисел может быть положительным, отрицательным или нулем. Непрерывный подмассив $A[i..j]$ называется положительным интервалом (positive interval), если сумма его элементов больше нуля. Описать и проанализировать алгоритм, вычисляющий минимальное число положительных интервалов, охватывающих каждый положительный элемент в массиве A . Например, если задан массив, показанный на рис. 4.11, то алгоритм должен вернуть 3. Если все элементы входного массива отрицательные, то алгоритм должен вернуть 0.

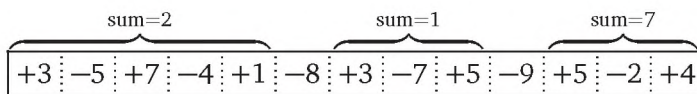


Рис. 4.11. Пример минимального числа положительных интервалов в массиве

16. Рассмотрим следующий процесс. В любой момент времени вы получаете одно положительное целое число x , которое изначально равно 1. На каждом шаге мы можете либо увеличить x на единицу (инкрементировать), либо удвоить x . Ваша цель – получить целевое значение n . Например, можно получить целое число 10 за четыре шага, как показано ниже:

$$1 \xrightarrow{+1} 2 \xrightarrow{\times 2} 4 \xrightarrow{+1} 5 \xrightarrow{\times 2} 10.$$

Разумеется, вы можете получить любое целое число n , используя ровно $n - 1$ операций инкрементирования, но почти для всех значений n это чрезвычайно неэффективно. Описать и проанализировать

алгоритм, вычисляющий минимальное число шагов, требуемых для получения любого заданного целого числа n .

17. Предположим, что имеется n лыжников с значениями их роста, заданными в массиве $P[1..n]$ и n пар лыж с длинами, заданными в массиве $S[1..n]$. Описать эффективный алгоритм распределения пар лыж для каждого лыжника так, чтобы средняя разность между ростом лыжника и длиной выданных ему лыж была по возможности минимальной. Алгоритм должен вычислять перестановку σ , такую что выражение

$$\frac{1}{n} \sum_{i=1}^n |P[i] - S[\sigma(i)]|$$

должно иметь настолько малое значение, насколько это возможно.

18. Алиса хочет устроить вечеринку и пытается решить, кого пригласить. Алиса должна выбрать из n человек, и ей известно, какие пары знают друг друга. Алиса хочет пригласить как можно больше людей с учетом двух ограничений:

- для каждого гостя должно существовать не менее пяти других гостей, которых он уже знает;
- для каждого гостя должно существовать не менее пяти других гостей, которых он пока еще не знает.

Описать и проанализировать алгоритм, который вычисляет максимальное возможное число гостей, которое Алиса может пригласить из заданного списка n людей и с учетом списка пар, знающих друг друга.

19. Предположим, что заданы два массива положительных целых чисел $C[1..n]$ и $R[1..n]$. Матрица $n \times n$ из нулей и единиц согласована с массивами R и C , если для каждого индекса i соответствующая i -я строка содержит $R[i]$ единиц, а i -й столбец содержит $C[i]$ единиц. Описать и проанализировать алгоритм, который либо формирует матрицу, согласованную с массивами R и C , либо корректно сообщает, что такой матрицы не существует.

20. Вы только что получили работу у Илона Маска (Elon Musk) – доставка бурритос из Сан-Франциско в Нью-Йорк Сити. Вы будете управлять специальным транспортным средством Burrrito-Delivery Vehicle в новом построенном Илоном трансконтинентальном тоннеле Transcontinental Underground Burrrito-Delivery Tube, который представляет прямую линию между этими двумя городами¹².

¹² ...и который был точно смоделирован после вымышленного «Alameda-Weehawken Burrrito Tunnel» Мацея Цегловского (Maciej Cegłowski).

Ваше транспортное средство Burrito-Delivery Vehicle работает на одноразовых батареях, которые обязательно необходимо заменять после каждых 100 миль (не более). Настоящее горючее почти бесплатно, но батареи дороги и хрупки, поэтому непременно должны устанавливаться только официальными сотрудниками Transcontinental Underground Burrito-Delivery Vehicle Battery-Replacement Technicians' Union¹⁵. Таким образом, даже если вы заменяете батарею раньше окончания ее срока службы, все равно придется платить полную цену за каждую новую устанавливаемую батарею. Кроме того, ваше транспортное средство слишком мало, чтобы нести более одной батареи одновременно.

В тоннеле есть несколько заправочных станций, каждая станция требует различную оплату установки новой батареи. Перед началом рейса вы предусмотрительно распечатали страницу «Википедии» со списком мест расположения и цен каждой заправочной станции в Тоннеле. Располагая этой информацией, как вы определите наилучшие пункты остановок для дозаправки?

Более формальное описание: предположим, что заданы два массива $D[1..n]$ и $C[1..n]$, где $D[i]$ – расстояние от начальной точки Тоннеля до i -й заправочной станции, а $C[i]$ – стоимость замены батареи на i -й станции. Предположим, что маршрут начинается и заканчивается на заправочных станциях (т. е. $D[1] = 0$, а $D[n]$ равно общей длине маршрута) и что ваше транспортное средство начинает рейс с полностью разряженной батареей (т. е. обязательно необходимо установить новую батарею на заправочной станции 1).

- (a) Описать и проанализировать жадный алгоритм поиска минимального числа остановок для дозаправки, необходимых для завершения вашего рейса. Не забудьте доказать корректность этого алгоритма.
 - (b) Но в действительности необходимо минимизировать общую стоимость рейса. Показать, что жадный алгоритм, разработанный в части (a), не позволяет получить оптимальное решение при заданном здесь расширении условий.
 - (c) Описать эффективный алгоритм, вычисляющий места расположения заправочных станций, на которых вы должны останавливаться, чтобы минимизировать общую стоимость рейса.
21. Вас приняли на работу для обслуживания последовательности n книг, размещенных на полках в библиотеке. Порядок расположения книг строго определен и записан в системе каталогизации и не может быть изменен, т. е. на каждой полке обязательно должна находиться непрерывная часть заданной последовательности книг. Заданы два

¹⁵ ...или, как они сами себя называют в Германии, Die Transkontinentaluntergrundburritolieferfahrzeugbatteriewechseltechnikervereinigung.

массива $H[1..n]$ и $T[1..n]$, где $H[i]$ и $T[i]$ – соответственно высота и толщина i -й книги в последовательности. Все полки в библиотеке имеют одинаковую длину L , так что суммарная толщина всех книг на любой полке не может превышать L .

- (а) Предположим, что все книги имеют одинаковую высоту h , а полки – высоту больше h , поэтому каждую книгу можно поместить на любую полку. Описать и проанализировать жадный алгоритм размещения книг на минимальном из всех возможных количестве полок. [*Подсказка*: алгоритм очевиден, но почему он является корректным?]
- (б) Это была неплохая разминка, но теперь нужно решить реальную задачу. В действительности книги имеют различную высоту, но вы можете регулировать высоту каждой полки, подгоняя ее под самую высокую книгу. (В частности, вы можете уменьшить высоту любой пустой полки до нуля.) Теперь ваша задача – разместить книги так, чтобы сумма высот всех полок была наименьшей из возможных. Показать, что жадный алгоритм, разработанный в части (а), не всегда позволяет получить наилучшее решение этой задачи.
- (с) Описать и проанализировать алгоритм поиска наилучшего соответствия между книгами и полками при условиях, описанных в части (б).

22. Строка w , состоящая из круглых скобок (и), является сбалансированной (balanced), если выполняется одно из следующих условий:

- w – пустая строка;
- $w = (x)$ для некоторой сбалансированной строки x ;
- $w = xy$ для некоторых сбалансированных строк x и y .

Например, строка

$w = ((()())(())())$

является сбалансированной, так как $w = xy$, где

$x = ((()())())$ и $y = (())()$.

- (а) Описать и проанализировать алгоритм, определяющий, является ли заданная строка из круглых скобок сбалансированной.
- (б) Описать и проанализировать алгоритм вычисления длины максимальной сбалансированной подпоследовательности заданной строки из круглых скобок. Как обычно, не забудьте доказать, что предложенный вами алгоритм является корректным.

Для обеих задач входные данные представлены массивом $w[1..n]$, где для каждого i либо $w[i] = ($, либо $w[i] =)$. Оба алгоритма должны выполняться за время $O(n)$.

23. Однажды Алекс надоело карабкаться по гимнастической стенке, и она решила собрать весьма большую группу друзей-скалолазов для упражнений на природе. В том месте, куда прибыла эта группа, находилась огромная широкая, но не слишком высокая каменная глыба с разнообразными зацепами для рук и ног. Алекс быстро определила «допустимое» множество перемещений, которые ее группа друзей может выполнить для перехода от одного зацепа к другому.

Общую систему зацепов можно описать деревом T с корнем и с n вершинами, где каждая вершина соответствует одному зацепу, а каждое ребро – допустимому перемещению между зацепами. Пути подъема сходятся, так как все они ведут на вершину этой глыбы, т. е. к особому зацепу в месте встречи, представленному корнем дерева T .

Алекс и ее друзья (все они превосходные скалолазы) решили сыграть в такую игру: максимальное возможное число скалолазов взбирается на глыбу, и от каждого требуется выполнить последовательность ровно k перемещений. Каждый скалолаз может выбрать произвольный зацеп для начала движения, но все перемещения обязательно должны начинаться с поверхности земли. Таким образом, каждый скалолаз последовательно проходит путь из k ребер в дереве T , при этом все пути направлены к корню. Но никаким двум скалолазам не разрешено одновременно касаться одного и того же зацепа, и пути, проходимые различными скалолазами, вообще не должны пересекаться.

- (а) Описать и проанализировать эффективный алгоритм вычисления максимального числа скалолазов, которые могут сыграть в эту игру. Входные данные для этого алгоритма: дерево T с корнем и целое число k . Алгоритм должен вычислить максимальное возможное число непересекающихся путей в дереве T , при этом каждый путь имеет длину k . Здесь не предполагается, что T является двоичным деревом. Например, если задано исходное дерево, показанное на рис. 4.12, то алгоритм должен вернуть целое число 8.

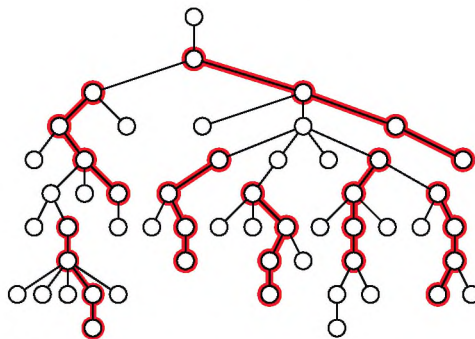


Рис. 4.12. Семь непересекающихся путей длиной $k = 3$.

Это не самое максимальное множество таких путей в данном дереве

- (b) Теперь предположим, что каждая вершина в дереве T имеет связанное с ней вознаграждение, и ваша цель – получить максимальное суммарное вознаграждение из вершин путей, а не общее число путей. Показать, что разработанный жадный алгоритм не всегда возвращает оптимальное вознаграждение.
- (c) Описать эффективный алгоритм, вычисляющий максимальное возможное вознаграждение при условиях, описанных в части (b).

24. Поздравляем! Вы успешно завоевали Камелот, преобразовав бывшую потрепанную передаваемую по наследству монархию в анархо-синдикалистскую коммуну, граждане которой поочередно исполняют обязанности ответственного руководителя в течение недели, но все принимаемые им решения должны утверждаться на особом собрании, проходящем два раза в неделю, простым большинством, если это исключительно внутренние дела, но в более серьезных вопросах требуется большинство в размере двух третей голосов...

Как завершающее знаменательное деяние вы приказали разделить Круглый стол (удивительно, но стол действительно круглый) на клинья (как при разрезании пиццы), которые распределяются между гражданами Камелота как награды. Каждый гражданин подал заявку на клинообразную часть стола, определяемую двумя углами, например сэр Робин Отважный может потребовать клин от 17.23° до 42° , а сэр Ланселот Непорочный может попросить клин в 2° от 359° до 1° . Каждый гражданин будет счастлив, если и только если получит в точности тот клин, который он запросил. К сожалению, некоторые запрашиваемые диапазоны углов перекрываются, поэтому удовлетворить запросы всех граждан просто невозможно. Добро пожаловать в политику.

Описать и проанализировать алгоритм поиска максимального числа запросов, которые можно удовлетворить. [Подсказка: вывод созданного алгоритма не должен изменяться при вращении стола. Здесь не предполагается, что значения углов являются целыми числами.]

25. Предположим, что вы стоите на поле, окруженном несколькими большими воздушными шарами. Вы хотите воспользоваться своей новейшей зажигалкой Acme Brand Zap-O-Matic™, чтобы прострелить все шары, не сходя с места. Зажигалка Zap-O-Matic™ испускает мощный лазерный луч, взрывающий все шары, в которые он попадает. Поскольку каждый выстрел требует расхода энергии, достаточного для энергоснабжения небольшой страны в течение года, вы должны сделать минимальное возможное количество выстрелов.

Задачу о минимальном количестве выстрелов можно описать более формально, как показано ниже. Задано множество S из n кругов на плоскости. Каждый круг определяется радиусом и координата-

ми (x, y) центра. Вычислить минимальное число лучей, исходящих из начала координат и пересекающих каждый круг из множества S . Ваша цель – найти эффективный алгоритм решения этой задачи.

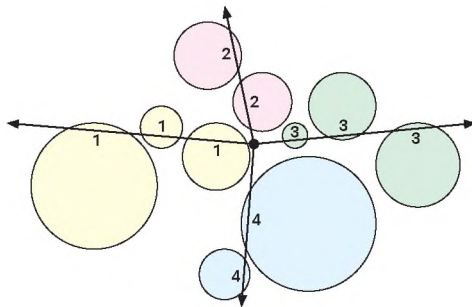


Рис. 4.13. Девять воздушных шаров, пораженных четырьмя выстрелами из Zap-O-Matic™

- (a) Предположим, что возможно выпустить луч, который не пересекает ни один из шаров. Описать и проанализировать жадный алгоритм, который решает задачу о минимальном количестве выстрелов в этом особом случае. [Подсказка: см. упражнение 4.]
- (b) Описать и проанализировать жадный алгоритм, вывод которого отклоняется не более чем на 1 от оптимального. То есть если m – минимальное число лучей, требуемое для поражения каждого шара, то созданный жадный алгоритм должен выводить либо m , либо $m + 1$. (Разумеется, вы должны доказать этот факт.)
- (c) Описать алгоритм, решающий задачу о минимальном количестве выстрелов за время $O(n^2)$.
- ♥(d) Описать алгоритм, решающий задачу о минимальном количестве выстрелов за время $O(n \log n)$.

Предположим, что у вас есть подпрограмма, сообщающая диапазон углов, под которыми лучи пересекают некоторый произвольный круг s . Подпрограмма выполняется за время $O(1)$. Такую подпрограмму не трудно написать, но это не самая интересная часть предложенной здесь задачи.

Глава 5

.....

Основные графовые алгоритмы

Распространение и передача знаний не похожи на несколько прямых, пересекающихся под некоторым углом в одной точке, но подобны ветвям дерева, которые объединяются в ствол с определенным размером и степенью целостности и непрерывности, прежде чем снова разъединиться, образуя ответвления и сучья.

— Фрэнсис Бэкон (Francis Bacon), «О достоинстве и приумножении наук»
(*The Advancement of Learning*) (1605 г.)

Таким образом, вы видите, благороднейший господин, что этот тип решения имеет мало отношения к математике, и я не понимаю, почему вы ожидаете, что математик даст его, а не кто-либо другой.

— Леонард Эйлер (Leonhard Euler), описание задачи о кенигсбергских мостах в письме Карлу Леонарду Готлибу Элеру (Carl Leonhard Gottlieb Ehler) (3 апреля 1736 г.)

Ну скажем так, поверните налево у пожарной станции в деревне и сверните на старую почтовую дорогу у водохранилища и ... нет, так не пойдет.

Лучше всего ехать прямо по асфальтированной дороге, пока не доедете до школы, а затем поверните налево на дорогу к озеру Беннетт до... нет, это тоже не годится.

Ист-Миллинокет, говорите? Если хорошо подумать, то отсюда вы не сможете добраться туда.

— Роберт Брайан (Robert Bryan) и Маршалл Додж (Marshall Dodge), юмористический цикл «Берт и я, и другие истории с Нижнего Восточного побережья (шт. Мэн)»
(*Bert and I and Other Stories from Down East*) (1961 г.)

5.1. Введение и историческая справка

Граф (graph) – это набор пар: пар целых чисел, людей, городов, звезд, стран, научных работ, веб-страниц, игровых позиций, рекурсивных подзадач и даже графов. В соответствии с наиболее распространенным методом визуализации графов базовые сопоставляемые объекты обычно называют

вершинами (vertices) или узлами (nodes), а сами пары называются ребрами (edges) или дугами (arcs), но в действительности объектами и парами может быть все, что угодно.

Одними из самых ранних примеров графов являются дорожные сети (road networks) и их карты. Римские инженеры создали сеть общедоступных дорог протяженностью более 400 000 км по всей Европе, а также в Западной и Центральной Азии и Северной Африке в период расцвета Римской империи. Путешественники по этой дорожной сети могли пользоваться итинерариями (itinerarium – описание путешествия; путеводитель), которые представляли собой либо простые списки, либо более красочные описания ландшафтов и расстояния, проходимые по различным дорогам. Итинерарий «Tabula Peutingeriana», свиток XIII в., описывающий «общественный путь» (cursus publicus) Римской империи, по общему мнению, является средневековой копией редакции V в. дорожной карты (itinerarium pictum) I в., созданной в эпоху правления Октавиана Августа (Augustus Caesar). Пейтингерова таблица (Peutinger Table) не является географически точной картой – историки дискутируют по поводу того, возможно ли вообще считать ее «картой», – это, скорее, абстрактное представление дорожной сети, больше похожее на современные схемы метрополитена. Города вдоль дорог обозначены резкими перегибами на кривой, представляющей конкретную дорогу. Названия этих городов и длины участков дорог между ними также обозначены на этой карте. Таким образом, карта содержит достаточный объем информации для поиска наикратчайшего маршрута между любыми двумя городами в Римской империи V в. (см. рис. 5.1).



Рис. 5.1. Небольшой фрагмент воспроизведенной в 1872 г. Конрадом Миллером (Konrad Miller) Пейтингеровой таблицы (Tabula Peutingeriana), на которой показана римская дорога от современного Биртена (Veteribus, вверху слева) через Кёльн (Agripina) и Бонн (Bonnae) в Майнц (Mogontiacum, вверху справа) с ответвлениями на Трир (Avg Tresvirorum, в центре) и Метц (Matricorvm, в центре внизу) (Источник: Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Tabula_Peutingeriana_-_Miller.jpg)

Одно из самых старых классических практических приложений графов – и в особенности деревьев – представление генеалогических деревьев. Сложные семейные «деревья» использовались веками для решения юридических вопросов о браках, наследстве и передаче королевского престола. Гражданское право Римской империи, впоследствии адаптированное как церковное каноническое право ранней католической церкви, запре-

щало брак между двоюродными братьями и сестрами или более близкими родственниками. В начале IX в. католическая церковь изменила требуемую степень родства и методику вычисления. Если римский закон о вычислении (*computatio legalis*) требовал сумму степеней родства до ближайшего общего предка не менее четырех поколений, то по новому закону о вычислении *computatio canonica* требовалось не менее семи поколений по обоим степеням родства. В 1215 г. под воздействием реальных условий (и практического опыта) церковь уменьшила минимальную требуемую степень родства при вступлении в брак до четырех поколений.¹ На рис. 5.2 на левой схеме показан чрезвычайно запутанный случай: Тириус (Tirius) и Тебурга (Theburga) вступили в брак, и у них родился сын Гаиус (Gaius), после чего Тириус (Tirius) умер. Затем Тебурга (Theburga) вышла замуж за Лотара, родила ему сына и умерла, наконец, Лотар (Lothar) и Берта (Bertha) вступили в брак и у них родилась дочь Джемма (Gemma). Может ли сын Гаиуса законно жениться на дочери Джеммы?

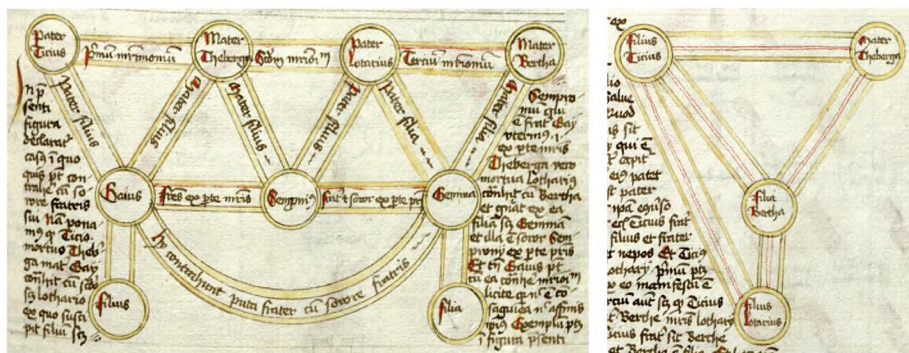


Рис. 5.2. Две схемы, описывающие сложный случай определения возможности вступления в брак, взятые из трактата неизвестного автора XV в., воспроизведенные по трактату Йоханнеса Андрэ (Johannes Andreae) «*Super arboribus consanguinitatis et affinitatis*» начала XIV в. о церковном каноническом праве (Источник: Галерея Legal Trees, опубликованная юридической библиотекой Йельской школы права Йельского университета (Yale Law Library) под защитой лицензии Creative Commons Licence; <https://www.flickr.com/photos/yalelawlibrary/albums/72157621954683764>)

В конце 1600-х гг. французский математик Пьер Вариньон (Pierre Varignon) разработал графический метод определения положения равновесия древовидной сети из веревок под растягивающими силами на основе более ранней работы Саймона Стевина (Simon Stevin), опубликованной веком ранее. Вариньон заметил, что если веревки находятся в равновесии,

¹ В XI и XII вв. это ограничение постепенно расширялось, включая до четырех родственных связей, сначала через брак, позже через внебрачные связи, обручения и даже через крестных родителей. Например, брак между мужчиной и сестрой мужа сестры мужа его сестры был формально запрещен, как и брак между вдовцом и вдовствующей матерью жены его сына. Эти требования к родственным связям были существенно снижены, но не отменены полностью в 1215 г., а принцип отслеживания степени родства при определении законности бракосочетания (*ex copula illicita*) был отменен церковью только в 1917 г.

то можно изобразить граф, ребрами которого являются отрезки, параллельные веревкам, с длинами, равными силам, действующими вдоль веревки так, что веревки, сходящиеся в одной точке (узле) сети, определяют замкнутый цикл в этом графе. Метод «графической статики» Вариньона не был опубликован со всеми подробностями до 1725 г., через два года после его смерти. Графы Вариньона теперь известны как диаграммы взаимодействующих сил или диаграммы Максвелла–Кремоны в честь Джеймса Клерка Максвелла (James Clerk Maxwell) и Луиджи Кремоны (Luigi Cremona), которые (вместе с Карлом Кульманом (Carl Culman) и др.) разработали подробнейшую теорию диаграмм взаимодействия в конце 1800-х гг.

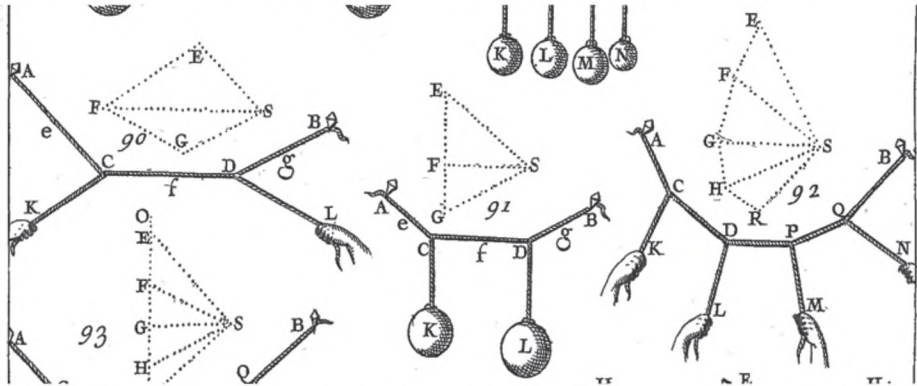


Рис. 5.3. Диаграммы взаимодействующих сил (изображены пунктирными линиями) из посмертно опубликованной работы Вариньона «Проект новой механики или статики 1687 г.» (*Nouvelle mécanique, ou statique, dont le projet fut donné en MDCLXXXVII*) (Источник: Internet Archive; <https://archive.org/details/A077240124/page/n261>)

Разумеется, существует множество других известных примеров использования графов, таких как настольные игры (относящиеся к периоду античности), вершины и ребра выпуклых многогранников (тщательно исследуемых не только древнегреческими учеными, но и намного ранее), визуальное представление созвездий (также разработанное в Восточной Азии к VII в. н. э.), рыцарские походы (описанные Аль-Адил, Рудрата, Аль-Сули (al-Adli, Rudrata, al-Suli) и др. в XI–X вв.), лабиринты (представленные в современной форме Джованни Фонтаной (Giovanni Fontana) около 1420 г.), геодезические триангуляции (которые впервые использовал Гемма Фризиус (Gemma Frisius) в 1533 г., а Виллеброрд Снелл (Willebrord Snellius) применил для вычисления длины окружности Земли в 1615 г., а в 1799 г. метод использовался для определения метра), широко известное частичное² решение Леонарда Эйлера (Leonhard Euler) задачи-головоломки

² Эйлер опустил завершающий шаг своего доказательства – в действительности это поиск пути обхода Эйлера в графе, в котором каждая вершина имеет четную степень – как очевидный. Эйлер также не заметил, что граф с путем обхода Эйлера обязательно должен быть связным. Первое полное доказательство того, что в графе существует путь обхода Эйлера, если и только если он является связным и каждая вершина имеет четную степень, было опубликовано Карлом Гиргольцером (Carl Hierholzer) в 1873 г.

о кенигсбергских мостах (1735 г.), телеграф и другие коммуникационные сети (впервые предложенные в 1753 г., разработанные Роналдсом (Ronalds), Шиллингом (Schilling), Гауссом (Gauss), Вебером (Weber) и др. в начале 1800-х гг.), электрические схемы (формализованные в начале 1800-х гг. Омом (Ohm), Максвеллом (Maxwell), Кирхгоффом (Kirchhoff) и др.), структурные формулы молекул (введенные независимо Августом Кекуле (August Kekulé) в 1857 г. и Арчибалдом Купером (Archibald Couper) в 1858 г.), социальные сети (впервые исследованные в середине 1930-х гг. социологом Джейкобом Морено (Jacob Moreno)), схемы цифровой электроники (предложенные Чарльзом Сандерсом Пирсом (Charles Sanders Peirce) еще в 1886 г. и приведенные к их современной форме Клодом Шенноном (Claude Shannon) в 1937 г.), ну и, если вы так настаиваете, – современный интернет.

Слово «граф» как абстрактный математический термин было придумано Джеймсом Сильвестром (James Sylvester) в 1878 г., который изменил термин Кекуле «chemicographs» (химиографы) для описания конкретных алгебраических инвариантов по предложению своего коллеги Уильяма Клиффорда (William Clifford). Слово «дерево» впервые было использовано для обозначения связных ациклических графов Артуром Кэйли (Arthur Cayley) в 1857 г., хотя абстрактная концепция деревьев уже применялась Густавом Кирхгоффом (Gustav Kirchhoff) и Карлом фон Штаудтом (Karl von Staudt) десятью годами ранее. Нулевая книга по теории графов была опубликована Андре Сен-Лагё (André Sainte-Laguë) в 1926 г., а десять лет спустя вышла первая книга по теории графов Денеша Кёнига (Dénes Kőnig).

5.2. Основные определения

Формально (простой) граф (graph) – это пара множеств (V, E) , где V – произвольное непустое конечное множество, элементы которого называются *вершинами* (vertices)³ или *узлами* (nodes), а E – множество пар элементов множества V , называемых *ребрами* (edges). В *ненаправленном* (неориентированном) графе (undirected graph) ребра являются неупорядоченными парами или просто множествами из двух элементов. Я обычно пишу uv вместо $\{u, v\}$ для обозначения ненаправленного ребра между вершинами u и v . В *направленном* (ориентированном) графе (directed graph) ребра являются упорядоченными парами вершин. Я обычно пишу $u \rightarrow v$ вместо (u, v) для обозначения направленного ребра от вершины u к вершине v .

Следуя стандартной (но по общему признанию запутанной) практике, я буду также использовать V для обозначения числа вершин в графе, а E для обозначения числа ребер. Таким образом, в любом ненаправленном графе имеем $0 \leq E \leq \binom{V}{2}$, а в любом направленном графе $0 \leq E \leq V(V-1)$.

³ Единственное число английского слова vertices – vertex. Аналогично единственное число слова matrices – matrix, единственное число слова indices – index. Если вы не говорите по-итальянски, то понимаете, что не существует таких слов как vertice, matrice, indice, appendice, helice, apice, vortice, radice, simplice, codice, directrice, dominatrice, Unice, Kleenice, Asterice, Obelice, Dogmatice, Getafice, Cacoфонice, Vitalstatistice, Geriатrice или Jimi Hendrice. Если вам трудно запомнить это правило, то просто используйте термин node (узел).

Конечными точками (endpoints) ребра uv или $u \rightarrow v$ являются его вершины u и v . Отдельно рассматриваются конечные точки направленного графа $u \rightarrow v$ – u называется *хвостом* (tail), а v – *головой* (head).

Определение графа как пары множеств запрещает существование нескольких ненаправленных ребер с одинаковыми конечными точками или нескольких направленных ребер с одинаковой головой и одинаковым хвостом. (Один и тот же направленный граф может содержать направленное ребро $u \rightarrow v$ и ребро, направленное противоположно, $v \rightarrow u$.) Кроме того, определение ненаправленного ребра как множества запрещает существование ненаправленного ребра от вершины к самой себе. Графы без циклов и параллельных ребер часто называют *простыми графами* (simple graphs), а непростые графы иногда называют *мультиграфами* или *множественными графами* (multigraphs). Несмотря на пробел в формальном определении, большинство алгоритмов для простых графов расширяется до мультиграфов с небольшими изменениями или без изменений, поэтому здесь я не вижу необходимости в формальном определении.

Для любого ребра uv в ненаправленном графе вершина u называется *соседней* вершиной (neighbor) вершины v и наоборот, и мы говорим, что u и v являются *смежными* (adjacent) вершинами. Степенью (degree) вершины является число вершин, соседних с ней. В направленных графах мы различаем два типа соседних вершин. Для любого направленного ребра $u \rightarrow v$ вершина u называется вершиной, *предшествующей* v , а вершина v называется вершиной, *следующей* за u (последователем u). Полустепень входа (in-degree) вершины – это число предшествующих ей вершин, а полустепень выхода (out-degree) вершины – это число ее последователей.

Граф $G' = (V', E')$ – это *подграф* (subgraph) графа $G = (V, E)$, если $V' \subseteq V$ и $E' \subseteq E$. *Собственный подграф* (proper subgraph) графа G – любой подграф, кроме самого графа G .

Маршрут (walk) в ненаправленном графе G – это последовательность вершин, где в каждой паре соседних вершин эти вершины являются соседними в G , неформально можно считать маршрут последовательностью ребер. Маршрут называется *путем* (path), если он посещает каждую вершину не более одного раза. Для любых двух вершин u и v в графе G мы говорим, что вершина v *достижима* (reachable) из вершины u , если G содержит маршрут (следовательно, и путь) между u и v . Ненаправленный граф является связным (connected), если каждая вершина достижима из каждой другой вершины. Каждый ненаправленный граф состоит из одной или нескольких компонент (components), которые являются максимально связанными подграфами этого графа, две вершины находятся в одной компоненте, если и только если между ними существует путь.⁴

Маршрут является *замкнутым* (closed), если он начинается и заканчивается в одной вершине. Цикл (cycle) – это замкнутый маршрут, входящий в каждую вершину и выходящий из нее не более одного раза. Ненаправ-

⁴ Компоненты часто называют «связными компонентами», но такой термин является избыточным, поскольку компоненты являются связными по определению.

ленный граф является ациклическим (acyclic), если ни один из подграфов не является циклом, ациклические графы также называют *леса*ми (forests). Дерево (tree) – это связный ациклический граф, или равнозначно – одна компонента леса. Остовное дерево (spanning tree) ненаправленного графа G – это подграф, являющийся деревом и содержащий каждую вершину графа G . Граф содержит остовное дерево, если и только если он является связным. Остовный лес (spanning forest) графа G – это набор остовных деревьев, по одному для каждой компоненты графа G .

Для направленных графов требуется немного другие определения. Направленный маршрут (directed walk) – это последовательность вершин $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_i$, такая что $v_{i-1} \rightarrow v_i$ является направленным ребром для каждого индекса i . Направленные пути и направленные циклы определяются аналогичным образом. Вершина v является достижимой из вершины u в направленном графе G , если и только если G содержит направленный маршрут (следовательно, и направленный путь) из u в v . Направленный граф является сильно связным (strongly connected), если каждая вершина доступна из каждой другой вершины. Направленный граф является ациклическим (acyclic), если он не содержит направленный цикл, направленный ациклический граф часто обозначают аббревиатурой DAG (directed acyclic graph – dag).

5.3. Представления и примеры

Наиболее широко распространенным способом визуального представления графов является их изображение в виде графической схемы. На графической схеме каждая вершина графа соответствует точке на плоскости (обычно изображаемой в небольшом кружочке или некоторой другой фигуры), а каждое ребро – кривой линии или отрезку прямой между двумя вершинами. Граф является *плоским* (planar), если на соответствующей ему схеме никакие два ребра не пересекаются, такая схема также называется *вложением* (embedding) графа⁵. Один и тот же граф можно отобразить многими различными схемами, поэтому важно не путать какую-либо конкретную схему и сам граф. В частности, плоские графы могут изображаться в виде неплоских схем.

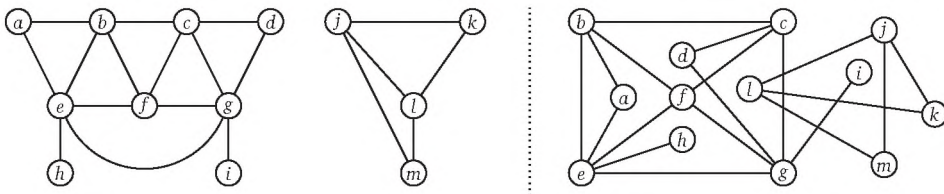


Рис. 5.4. Две схемы несвязного плоского графа с 13 вершинами, 19 ребрами и двумя компонентами. Вложением является только схема слева

⁵ По непонятной причине слово «вложение» (embedding) часто используется как синоним слова «изображение» (drawing), даже если ребра пересекаются. Такое использование термина «вложение» не рекомендуется.

Но графические схемы являются далеко не единственным удобным способом представления графов. Например, граф пересечений (intersection graph) группы геометрических объектов содержит узел для каждого объекта и ребро для каждой пары пересекающихся объектов. Возможность представления конкретного графа как графа пересечений зависит от типа объектов, которые должны использоваться как вершины. Различные типы объектов – отрезки прямых, прямоугольники, окружности и т. д. – определяют различные классы графов. Одним особенно полезным типом графа пересечений является граф интервалов (interval graph), вершины которого представляют отрезки координатной оси с ребрами между любыми двумя перекрывающимися отрезками.

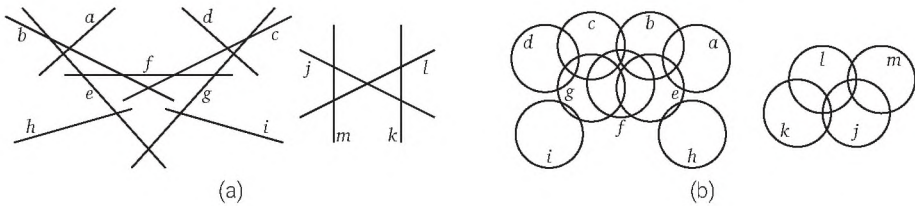


Рис. 5.5. Граф на рис. 5.4 также является графом пересечений:
(а) множества отрезков и (б) множества окружностей

Еще один хороший пример – граф зависимостей (dependency graph) рекурсивного алгоритма. Графы зависимостей – это направленные ациклические графы. Вершинами являются отдельные рекурсивные подзадачи, возникающие при выполнении алгоритма с конкретными входными данными. Ребро от одной подзадачи к другой существует, если для выполнения второй подзадачи требуется рекурсивное выполнение первой. Например, для рекуррентной последовательности чисел Фибоначчи

$$F_n = \begin{cases} 0 & , \text{ если } n = 0, \\ 1 & , \text{ если } n = 1, \\ F_{n-1} + F_{n-2} & \text{ иначе} \end{cases}$$

вершинами графа зависимостей являются целые числа $0, 1, 2, \dots, n$, а ребрами – пары $(i-1) \rightarrow i$ и $(i-2) \rightarrow i$ для каждого целого числа i между 2 и n .

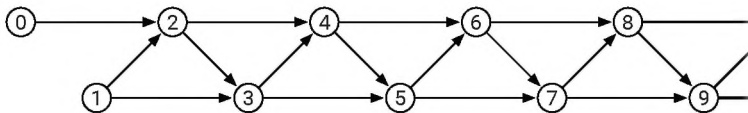


Рис. 5.6. Граф зависимостей для рекуррентной последовательности Пингала–Фибоначчи

Более сложный пример: вспомним рекуррентное выражение для задачи определения редакционного расстояния из главы 3:

$$Edit(i, j) = \begin{cases} i & , \text{ если } i = 0; \\ j & , \text{ если } j = 0; \\ \min \begin{cases} Edit(i, j - 1) + 1 \\ Edit(i - 1, j) + 1 \\ Edit(i - 1, j - 1) + [A[i] \neq B[j]] \end{cases} & \text{ иначе.} \end{cases}$$

Граф зависимостей для этого рекуррентного выражения представляет собой сетку $m \times n$ вершин (i, j) , соединенных вертикальными ребрами $(i - 1, j) \rightarrow (i, j)$, горизонтальными ребрами $(i, j - 1) \rightarrow (i, j)$ и диагональными ребрами $(i - 1, j - 1) \rightarrow (i, j)$. Метод динамического программирования эффективно работает для любого рекуррентного выражения, которому соответствует граф зависимостей приемлемо малого размера. Правильный порядок вычислений гарантирует, что каждая подзадача посещается после ее предшественников.

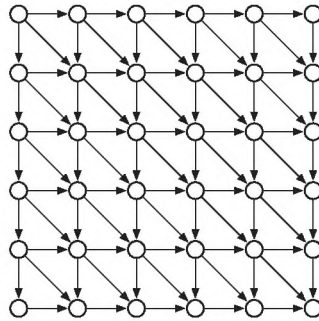


Рис. 5.7. Граф зависимостей для рекуррентного выражения редакторского расстояния

Еще один интересный пример – граф конфигурации (configuration graph) игры, головоломки или механизма, например игры в крестики-нолики, в шашки, кубика Рубика, головоломки «Ханойская башня» или машины Тьюринга. Вершинами графа конфигурации являются все допустимые конфигурации головоломки, ребро от одной конфигурации к другой существует, если возможно преобразование первой конфигурации во вторую с помощью одного простого «хода». (Разумеется, точное определение зависит от того, какие ходы разрешены.) Даже для относительно простых механизмов граф конфигурации может оказаться чрезвычайно сложным, а мы обычно получаем доступ только к локальной информации о графе конфигурации.

Графы конфигурации тесно связаны с деревьями игры, которые рассматривались в главе 2, но с одним весьма важным отличием. Каждое состояние игры встречается ровно один раз в ее графе конфигурации, но может встречаться многократно в дереве этой игры. Кратко говоря, графы конфигурации – это мемоизованные деревья игры.

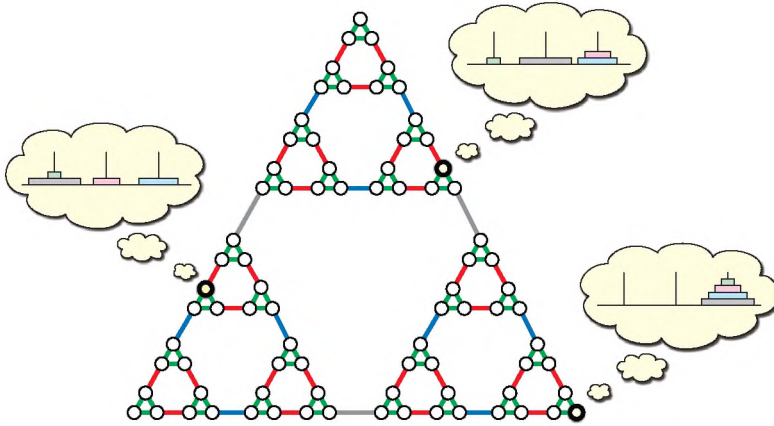


Рис. 5.8. Граф конфигурации головоломки «Ханойская башня» с четырьмя дисками

Конечные автоматы (finite state automata), используемые в теории формальных языков, можно моделировать как помеченные направленные графы. Напомним, что детерминированный конечный автомат формально определяется как кортеж из пяти элементов $M = (\Sigma, Q, s, A, \delta)$, где Σ – конечное множество, называемое алфавитом, Q – конечное множество состояний, $s \in Q$ – начальное состояние, $A \subseteq Q$ – множество поглощающих состояний, $\delta : Q \times \Sigma \rightarrow Q$ – функция перехода. Но зачастую более удобно представить M как направленный граф G_M , вершинами которого являются состояния Q , а ребра имеют форму $q \rightarrow \delta(q, a)$ для каждого состояния $q \in Q$ и символа $a \in \Sigma$. Простые вопросы о языке $L(M)$, соответствующем конечному автомату M , в дальнейшем могут быть переформулированы как вопросы о графе G_M . Например, $L(M) = \emptyset$, если и только если нет принятого состояния/вершины, которое доступно из начального состояния/вершины s .

Наконец, иногда граф может использоваться для неявного представления других более крупных графов. Хорошим примером такого неявного представления является конструкция подмножеств (subset construction), обычно используемая для преобразования недетерминированных конечных автоматов (НКА) в детерминированные конечные автоматы (ДКА), но также применимая к произвольным направленным графам, как показано ниже. Пусть задан любой направленный граф $G = (V, E)$, и мы можем определить новый направленный граф $G' = (2^V, E')$, все вершины которого являются подмножествами множества вершин V , а ребра E' определены следующим образом:

$$E' := \{A \rightarrow B \mid u \rightarrow v \in E \text{ для некоторых } u \in A \text{ и } v \in B\}.$$

Это определение можно механически преобразовать в алгоритм для конструкции G' из графа G , но, строго говоря, в такой конструкции нет необходимости, поскольку G уже является неявным представлением G' .

Важно не путать любой из этих примеров/представлений с действительным формальным определением: граф – это пара множеств (V, E) , где V – произвольное непустое конечное множество, а E – множество пар (упорядоченных или неупорядоченных) элементов V . Короче говоря, граф – это множество пар объектов.

5.4. Структуры данных

На практике графы обычно представлены одной из двух стандартных структур данных: списков смежных вершин (adjacency lists) и матриц смежности (adjacency matrices). На высоком уровне обе структуры данных представляют собой массивы, индексированные по вершинам, для этого требуется, чтобы каждой вершине соответствовал неповторяющийся целочисленный идентификатор в интервале от 1 до V . С формальной точки зрения эти целочисленные идентификаторы и есть вершины.

Списки смежных вершин

В основном наиболее часто используемой структурой данных для хранения графов является список смежных вершин (adjacency list). Список смежных вершин – это массив списков, при этом каждый список содержит соседние вершины для одной из вершин (или соседние вершины на выходе, если граф направленный).⁶ Для ненаправленных графов каждое ребро uv сохраняется дважды: сначала в списке соседей u , затем в списке соседей v . Для направленных графов каждое ребро $u \rightarrow v$ сохраняется только один раз, в списке соседей хвоста u . Для обоих типов графов общее пространство, требуемое для хранения списка смежных вершин, равно $O(V + E)$.

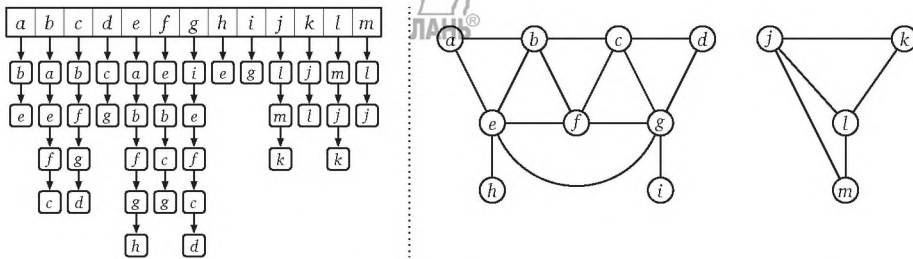


Рис. 5.9. Список смежных вершин (слева) для рассматриваемого примера графа (справа)

Существует несколько различных способов представления таких списков соседних вершин, но в стандартной реализации используется простой односвязный список. Полученная в результате структура данных позволяет

⁶ Внимательные читатели могут заметить, что, несмотря на название, список смежных вершин не является списком. Его наименование представляет собой пример принципа «копченной сельди» (Red Herring Principle) (отвлекающего маневра): в информационных технологиях, как и в математике, копченная сельдь не обязательно является копченной и даже не обязательно должна быть рыбой.

перечислить соседей (или соседей на выходе) узла v за время $O(1 + \deg(v))$ простым проходом (сканированием) по списку соседей v . Точно так же можно определить, является ли $u \rightarrow v$ ребром, за время $O(1 + \deg(u))$, проходя по списку соседей u . Для ненаправленных графов можно улучшить время до $O(1 + \min\{\deg(u), \deg(v)\})$, одновременно проходя по спискам соседей узлов u и v и останавливаясь, если найдено ребро или обнаружен конец списка.

Разумеется, связанные списки не единственная структура данных, которую можно использовать, для представления графов подходит любая другая структура, поддерживающая операции поиска, формирования списка, вставки и удаления. Например, можно сократить время определения, является ли uv ребром, до $O(1 + \log(\deg(u)))$, воспользовавшись сбалансированным двоичным деревом поиска для хранения соседей u , или даже до времени $O(1)$, если применить правильно сформированную хеш-таблицу⁷.

Широко распространенной реализацией списков смежных вершин является массив смежности (adjacency array), где используется один массив для хранения всех записей о ребрах вместе с записями о ребрах, смежных с каждой вершиной в непрерывном интервале, и отдельный массив, хранящий индекс первого ребра, смежного с каждой вершиной. Кроме того, полезно хранить интервалы для каждой вершины в отсортированном порядке, как показано на рис. 5.10, чтобы можно было за время $O(\log \deg(u))$ проверить, являются ли смежными две вершины u и v .

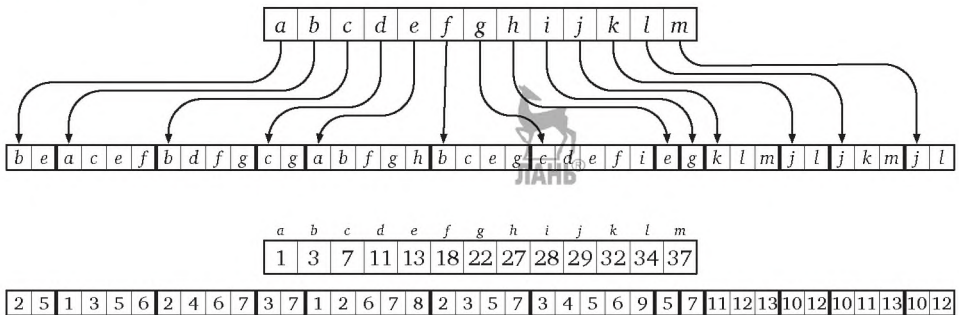


Рис. 5.10. Абстрактный массив смежности для рассматриваемого примера графа и его действительная реализация в виде пары массивов с целочисленными элементами

Матрицы смежности

Еще одна стандартная структура данных для графов – матрица смежности (adjacency matrix)⁸, впервые предложенная Жоржем Брюнелем (Georges Brunel) в 1894 г. Матрица смежности графа G – это матрица $V \times V$ из нулей и единиц, обычно представленная в виде двумерного массива $A[1..V, 1..V]$,

⁷ Это намного сложнее сделать, чем сказать. Наиболее широко распространенные методики хеширования не обеспечивают быстрое выполнение запросов, и даже самые лучшие методы хеширования могут гарантировать только лишь ожидаемое время $O(1)$. Более подробное обсуждение методик хеширования см. здесь: <http://algorithms.wtf>.

⁸ Смотрите сноску 3.

каждый элемент определяет, присутствует ли конкретное ребро в графе G . Более определенно для всех вершин u и v :

- если граф ненаправленный, то $A[u, v] := 1$, если и только если $uv \in E$,
- если граф направленный, то $A[u, v] := 1$, если и только если $u \rightarrow v \in E$.

Для ненаправленных графов матрица смежности всегда симметрична, т. е. $A[u, v] = A[v, u]$ для всех вершин u и v , так как uv и vu – это просто разные имена одного и того же ребра, а все диагональные элементы $A[u, u]$ равны нулю. Для направленных графов матрица смежности может быть симметричной или несимметричной, а диагональные элементы могут быть равны или не равны нулю.

	a	b	c	d	e	f	g	h	i	j	k	l	m
a	0	1	0	0	1	0	0	0	0	0	0	0	0
b	1	0	1	0	1	1	0	0	0	0	0	0	0
c	0	1	0	1	0	1	1	0	0	0	0	0	0
d	0	0	1	0	0	0	1	0	0	0	0	0	0
e	1	1	0	0	0	1	1	1	0	0	0	0	0
f	0	1	1	0	1	0	1	0	0	0	0	0	0
g	0	0	1	1	1	1	0	0	1	0	0	0	0
h	0	0	0	0	1	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	1	0	0	0	0	0	0
j	0	0	0	0	0	0	0	0	0	0	1	1	1
k	0	0	0	0	0	0	0	0	0	1	0	1	0
l	0	0	0	0	0	0	0	0	0	1	1	0	1
m	0	0	0	0	0	0	0	0	0	1	0	1	0

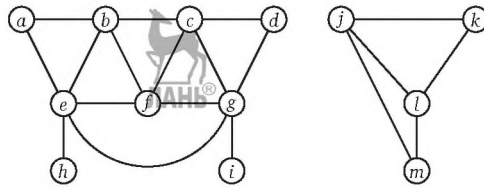


Рис. 5.11. Матрица смежности для рассматриваемого примера графа

Если задана матрица смежности, то можно определить за время $\Theta(1)$, соединены ли две вершины ребром, просто проверив соответствующую позицию в этой матрице. Также можно определить всех соседей некоторой вершины за время $\Theta(V)$, просматривая соответствующую строку (или столбец). Это время выполнения оптимально в наихудшем случае, но даже если вершина имеет лишь нескольких соседей, все же необходимо просмотреть всю строку полностью, чтобы найти всех. Кроме того, для матриц смежности требуется пространство $\Theta(V^2)$ независимо от того, сколько ребер в действительности содержит граф, поэтому матрицы смежности эффективно используют пространство только для весьма плотных графов.

Сравнение

В табл. 5.1 приведены показатели производительности по различным структурам данных для графов. Звездочки (*) обозначают ожидаемые амортизационные границы времени для сопровождения динамических хеш-таблиц.⁹

⁹ Не беспокойтесь, если вам непонятно выражение «ожидаемые амортизационные».

Таблица 5.1. Показатели времени для базовых операций в стандартных структурах данных графов

	Стандартный список смежных вершин (связные списки)	Быстрый список смежных вершин (хеш-таблицы)	Матрица смежности
Пространство	$\Theta(V + E)$	$\Theta(V + E)$	$\Theta(V^2)$
Проверка, если $uv \in E$	$O(1 + \min\{\deg(u), \deg(v)\}) = O(V)$	$O(1)$	$O(1)$
Проверка, если $u \rightarrow v \in E$	$O(1 + \deg(u)) = O(V)$	$O(1)$	$O(1)$
Список соседей (на выходе) вершины v	$\Theta(1 + \deg(v)) = O(V)$	$\Theta(1 + \deg(v)) = O(V)$	$\Theta(V)$
Список всех ребер	$\Theta(V + E)$	$\Theta(V + E)$	$\Theta(V^2)$
Вставка ребра uv	$O(1)$	$O(1)^*$	$O(1)$
Удаление ребра uv	$O(\deg(u) + \deg(v)) = O(V)$	$O(1)^*$	$O(1)$

С учетом этого сравнения можно с удивлением задать резонный вопрос: зачем вообще использовать матрицу смежности, если списки смежных вершин с применением хеш-таблиц поддерживают те же операции, выполняемые за то же время, но требуют меньше пространства. Главная причина – для достаточно плотных графов матрицы смежности на практике более просты и эффективны, потому что позволяют избежать излишних накладных расходов на перебор указателей и вычисление хеш-функций; это просто смежные блоки памяти.

Аналогично: зачем использовать связные списки в структуре списка смежных вершин для хранения соседних вершин вместо сбалансированных двоичных деревьев поиска или хеш-таблиц? На практике главная причина почти наверняка в преемственности – если связные списки достаточно хороши для кода Дональда Кнута, то они должны подходить и для нас, – но существуют и более обоснованные аргументы. Один из них – стандартные списки смежных вершин в действительности достаточно хороши для большинства приложений. Большинство стандартных графовых алгоритмов никогда (или редко) на практике не задаются вопросом о том, присутствует или отсутствует какое-либо ребро в графе, и не пытаются вставлять или удалять ребра, поэтому оптимизация структур данных для поддержки этих операций не является необходимостью.

Но, по моему мнению, самой убедительной причиной для использования обеих стандартных структур данных является то, что многие графы неявно представлены матрицами смежности и стандартными списками смежных вершин. Например:

- графы пересечений обычно представлены как список геометрических объектов более низкого уровня. Поскольку мы можем проверить за постоянное время, пересекаются ли какие-либо два объекта, можно применить любой графовый алгоритм к графу пересечений, представив, что исходный граф сохранен явно как матрица смежности;
- любая структура данных, состоящая из записей с указателями между ними, может быть представлена как направленный граф. Графовые алгоритмы можно применять к таким структурам данных, приняв, что граф сохранен в стандартном списке смежных вершин;
- также можно применить любой графовый алгоритм к графу конфигурации, как если бы он был представлен в виде стандартного списка смежных вершин, при условии, что мы можем пронумеровать все возможные ходы из заданной конфигурации, и каждый ход выполняется за постоянное время.

В двух последних примерах можно перечислить ребра, покидая каждую вершину за время, пропорциональное ее степени, но не всегда возможно определение за постоянное время, если две вершины являются смежными. (Существует ли указатель из этой записи на ту запись? Можно ли перейти из этой конфигурации в ту конфигурацию за один ход?) Кроме того, обычно мы не имеем возможности преобразовать указатели в каждой конфигурации или ходы из заданной конфигурации в более эффективную структуру данных. Таким образом, стандартный список смежных вершин с соседями, сохраненными в связанных списках, является подходящей структурой данных.

С этого момента и до конца книги, если явно не называется какой-либо другой вариант, все временные границы для графовых алгоритмов предполагают, что входной граф представлен стандартным списком смежных вершин. Аналогично если явно не называется какой-либо другой вариант, то, когда в упражнении предлагается спроектировать и проанализировать графовый алгоритм, вы должны считать, что входной граф представлен стандартным списком смежных вершин.

5.5. Поиск в любом направлении

До сих пор мы обсуждали только локальные операции в графах. Вероятно, самый главный глобальный вопрос, который можно задать о графах, – это достижимость (reachability). Пусть задан граф G и вершина s в G , и возникает вопрос о достижимости: какие вершины достижимы из s , т. е. для каких вершин v существует путь из s в v ? Пока рассмотрим только ненаправленные графы, в конце этого раздела будет кратко описана достижимость в направленных графах. Для ненаправленных графов вершины, достижимые из s , – это определенно вершины в той же компоненте, что и вершина s .

Возможно, наиболее естественный алгоритм определения достижимости – по крайней мере, для людей, похожих на нас, привыкших мыслить ре-

курсивно – поиск в глубину (depth-first search). Этот алгоритм можно записать либо рекурсивно, либо итеративно. В любом случае это один и тот же алгоритм с единственным различием – мы можем видеть стек «рекурсии» в нерекурсивной версии.

RecursiveDFS(v):

```

if v is unmarked
  mark v
  for each edge vw
    RecursiveDFS(w)

```

IterativeDFS(s):

```

Push(s)
while the stack is not empty
  v ← Pop
  if v is unmarked
    mark v
    for each edge vw
      Push(w)

```

Поиск в глубину – это лишь одна (возможно, чаще всего применяемая) разновидность из общего семейства алгоритмов обхода графа, которое я называю поиском в любом направлении (whatever-first search). Обобщенный алгоритм обхода сохраняет множество ребер-кандидатов в некоторой структуре данных, которую я буду называть мешком (bag). Самые важные свойства мешка – в него можно складывать вещи, а потом их вынимать. Стек является частным типом (случаем) мешка, но определенно не единственным. Ниже приведен обобщенный алгоритм.

WhateverFirstSearch(s):

```

put s into the bag
while the bag is not empty
  take v from the bag
  if v is unmarked
    mark v
    for each edge vw
      put w into the bag

```

Я утверждаю, что алгоритм `WhateverFirstSearch` помечает каждый узел, достижимый из вершины s , и никакие другие узлы. Этот алгоритм явно помечает каждую вершину в графе G не более одного раза. Чтобы показать, что каждый узел в связном графе посещается как минимум один раз, немного изменим алгоритм – изменения выделены темно-красным цветом в псевдокоде, приведенном ниже. Вместо сохранения в мешке вершин алгоритм сохраняет пары вершин. Это изменение позволяет запомнить при посещении вершины v в первый раз, какая из ранее посещенных соседних вершин поместила v в мешок. Эту предшествующую вершину назовем родителем вершины v .

```

WhateverFirstSearch(s):
  put ( $\emptyset, s$ ) in bag
  while the bag is not empty
    take ( $p, v$ ) from the bag      (★)
    if  $v$  is unmarked
      mark  $v$ 
       $\text{parent}(v) \leftarrow p$ 
      for each edge  $vw$           (†)
        put ( $v, w$ ) into the bag  (★★)

```

Лемма 5.1. Алгоритм `WhateverFirstSearch(s)` помечает каждую из вершин, достижимых из вершины s , и только их. Кроме того, множество всех пар $(v, \text{parent}(v))$ при $\text{parent}(v) \neq \emptyset$ определяет остовное дерево компоненты, содержащей s .

Доказательство. Сначала докажем, что алгоритм помечает каждую вершину v , достижимую из s , методом индукции по расстоянию наикратчайшего пути из s в v . Алгоритм помечает вершину s . Пусть v – любая другая вершина, достижимая из s , и пусть $s \rightarrow \dots \rightarrow u \rightarrow v$ – любой путь из s в v с минимальным числом ребер. (Такой путь обязательно должен существовать, потому что вершина v достижима из s .) Префиксный путь $s \rightarrow \dots \rightarrow u$ короче, чем наикратчайший путь из s в v , поэтому по индуктивному предположению алгоритм помечает вершину u . Когда алгоритм помечает u , он обязательно должен поместить пару (u, v) в мешок, чтобы впоследствии обязательно взять пару (u, v) из мешка. В этот момент алгоритм немедленно помечает вершину v , если она не была помечена ранее.

Каждая пара $(v, \text{parent}(v))$ при $\text{parent}(v) \neq \emptyset$ действительно является ребром в исходном графе G . Мы утверждаем, что для любой помеченной вершины v путь из родительских ребер $v \rightarrow \text{parent}(v) \rightarrow \text{parent}(\text{parent}(v)) \rightarrow \dots$ – в итоге приводящий обратно в s . Докажем это утверждение методом индукции по порядку, в котором помечены вершины. Очевидно, что вершина s достижима из s , тогда пусть v – любая другая помеченная вершина. Родители v непременно должны быть помечены раньше, чем помечена вершина v , поэтому по индуктивному предположению родительский путь $\text{parent}(v) \rightarrow \text{parent}(\text{parent}(v)) \rightarrow \dots$ ведет в вершину s , а добавление еще одного родительского ребра $s \rightarrow \text{parent}(s)$ доказывает исходное утверждение.

Предыдущее утверждение предполагает, что каждая вершина, помеченная алгоритмом, достижима из вершины s , а также что множество всех родительских ребер образует связный граф. Поскольку каждый помеченный узел, за исключением s , имеет связанного только с ним родителя, число родительских ребер ровно на единицу меньше числа помеченных вершин. Мы приходим к выводу о том, что родительские ребра образуют дерево. $\square$

Анализ

Время выполнения алгоритма обхода зависит от структуры данных, которую мы используем для мешка, но можно сделать несколько общих наблюдений. Пусть T – время, требуемое для вставки одного элемента в мешок или для удаления одного элемента из мешка. Цикл `for (†)` выполняется ровно по одному разу для каждой помеченной вершины, следовательно, не более V раз. Каждое ребро uv в компоненте вершины s помещается в мешок ровно два раза – один раз как пара (u, v) и один раз как пара (v, u) , поэтому строка $(\star\star)$ выполняется не более $2E$ раз. Наконец, из мешка невозможно извлечь больше вещей, чем в него было помещено, поэтому строка $(\star)$ выполняется не более $2E + 1$ раз. Следовательно, если предположить, что исходный граф G хранится в стандартном списке смежных вершин, то алгоритм `WhateverFirstSearch` выполняется за время $O(V + ET)$. (Если G сохранен в матрице смежности время выполнения алгоритма `WhateverFirstSearch` увеличивается до $O(V^2 + ET)$).

5.6. Важные варианты

Стек: поиск в глубину

Если мешок реализуется с использованием стека, мы возвращаемся к исходному алгоритму поиска в глубину. Стеки поддерживают операции вставки (`push`) и удаления (`pop`) элементов за время $O(1)$, поэтому алгоритм выполняется за время $O(V + E)$. Остовное дерево, сформированное родительскими ребрами, называется остовным деревом поиска в глубину (`depth-first spanning tree`). Конкретная форма дерева зависит от начальной вершины и от порядка, в котором посещаются соседние вершины в цикле `for (†)`, но в общем случае остовные деревья поиска в глубину длинные и тонкие. Некоторые важные свойства и приложения поиска в глубину будут рассматриваться в главе 6.

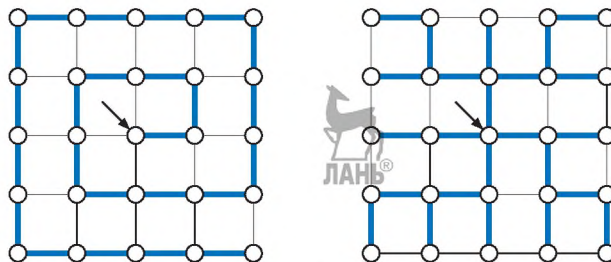


Рис. 5.12. Остовное дерево поиска в глубину и остовное дерево поиска в ширину одного и того же графа. Оба дерева начинаются с центральной вершины

Очередь: поиск в ширину

Если мешок реализуется с использованием очереди (`queue`), то мы получаем другой алгоритм обхода графа, который называется поиском в шири-

ну (breadth-first search). Очереди поддерживают операции вставки (push) и удаления (pull), выполняемые за время $O(1)$, поэтому алгоритм выполняется за время $O(V + E)$. В этом случае остовное дерево поиска в ширину (breadth-first spanning tree), формируемое из родительских ребер, содержит наикратчайшие пути от начальной вершины s к каждой прочей вершине своей компоненты. Более подробно наикратчайшие пути будут рассматриваться в главе 8. И здесь конкретная форма остовного дерева поиска в ширину зависит от начальной вершины и от порядка посещения соседних вершин в цикле `for` ($\dagger$), но в общем остовные деревья поиска в ширину короткие и развесистые.

Очередь с приоритетами: поиск по первому наилучшему совпадению

Если мешок реализуется с использованием очереди с приоритетами (priority queue), то применяется еще одно семейство алгоритмов, называемое поиском по первому наилучшему совпадению (best-first search). Поскольку очередь с приоритетами хранит не более одной копии каждого ребра, операция вставки ребра или удаления ребра с минимальным приоритетом требует времени $O(\log E)$, отсюда следует, что поиск по первому наилучшему совпадению выполняется за время $O(V + E \log E)$.

Я описываю поиск по первому наилучшему совпадению как «семейство алгоритмов», а не как единственный алгоритм, потому что существуют разнообразные методы присваивания приоритетов ребрам, и эти варианты выбора приводят к различному алгоритмическому поведению. Ниже будут описаны три наиболее широко известных варианта, но существует и множество других вариантов. Во всех трех примерах подразумевается, что каждое ребро uv или $u \rightarrow v$ в исходном графе имеет неотрицательный вес $w(uv)$ или $w(u \rightarrow v)$.

Первый вариант: если исходный граф ненаправленный и вес каждого ребра используется как его приоритет, то алгоритм поиска по первому наилучшему совпадению формирует минимальное остовное дерево компоненты вершины s . Как ни странно, в том случае, если веса всех ребер различны, то итоговое дерево не зависит от начальной вершины и от порядка посещения соседних вершин – это минимальное остовное дерево действительно является единственным в своем роде. Эта версия поиска по первому наилучшему совпадению широко (но, как обычно, ошибочно) известна как алгоритм Прима (Prim's algorithm). Более подробно эта и другие версии минимальных остовных деревьев будут рассматриваться в главе 7.

Определим длину пути (length of path) как сумму весов его ребер. Также можно вычислять наикратчайшие пути (shortest paths) во взвешенных графах с использованием алгоритма поиска по первому наилучшему совпадению, как показано ниже. Каждая помеченная вершина v хранит расстояние $dist(v)$. Изначально устанавливается $dist(s) = 0$. Для каждой другой вершины v при установке $parent(v) \leftarrow p$ также выполняется присваивание

$dist(v) \leftarrow dist(p) + w(p \rightarrow v)$, а при вставке ребра $v \rightarrow w$ в очередь с приоритетами используется приоритет $dist(v) + w(v \rightarrow w)$. Если предположить, что все веса ребер положительны, то $dist(v)$ – это длина наикратчайшего пути от s к v . Эта версия поиска по первому наилучшему совпадению широко (но, как обычно, строго говоря, ошибочно) известна как алгоритм Дейкстры (Dijkstra's algorithm). Мы снова встретимся с этим алгоритмом в главе 8.

Определим ширину пути (width of path) как минимальный вес любого ребра в этом пути. Простая модификация алгоритма поиска по первому наилучшему совпадению «Дейкстры» вычисляет пути максимальной ширины (widest paths) от s к каждой прочей достижимой вершине. Пути максимальной ширины также называют путями с максимальной пропускной способностью (bottleneck shortest paths). Каждая помеченная вершина v хранит значение $width(v)$. Изначально устанавливается $width(s) = \infty$. Для любой другой вершины v при установке $parent(v) \leftarrow p$ также выполняется присваивание $width(v) \leftarrow \min\{width(p), w(p \rightarrow v)\}$, а при вставке ребра $v \rightarrow w$ в очередь с приоритетами используется приоритет $\min\{width(v), w(v \rightarrow w)\}$. Пути максимальной ширины удобны для использования в алгоритмах вычисления максимальных потоков (maximum flows), которые (как вы уже догадались) будут рассматриваться в главе 10.

Несвязные графы

Алгоритм `WhateverFirstSearch` проходит только по вершинам, достижимым из единственной начальной вершины s . Для посещения каждой вершины графа G можно воспользоваться приведенной ниже простой функцией-оберткой.

```
WFSAll(G):
  for all vertices v
    unmark v
  for all vertices v
    if v is unmarked
      WhateverFirstSearch(v)
```

Подождите, – спросите вы, – зачем все так усложнять? Почему бы просто¹⁰ не пройти последовательно по массиву вершин?

```
MarkEveryVertexDuh(G):
  for all vertices v
    mark v
```

Разумеется, если у вас есть полный список вершин, то вы можете это сделать, но помните, что не все графы представлены в такой явной форме.¹¹

¹⁰ Это слово (в оригинале «just») почти всегда сообщает о том, что вы пропустили что-то важное.

¹¹ С другой стороны, если мы храним в каждой вершине метки времени, определяющие время последней «пометки», то можно «отменить пометку каждой вершины» за время $O(1)$, записав время начала обхода и считая вершину «помеченной», если ее метка времени более поздняя, чем записанное начальное время.

Но более важно то, что даже если мы имеем в явном виде полный список вершин, то порядок, в котором этот простейший алгоритм посещает вершины, определяется порядком записи вершин в структуре данных, а не абстрактной структурой графа.

В частности, в отличие от простейшего обхода всех вершин алгоритм WFSAll посещает все вершины в одной компоненте, затем все вершины в следующей компоненте и т. д. по каждой компоненте исходного графа. Обход по схеме компонента-за-компонентой позволяет, например, подсчитывать компоненты несвязного графа с помощью одного-единственного счетчика.

```
CountComponents(G):
    count ← 0
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            count ← count + 1
            WhateverFirstSearch(v)
    return count
```

После небольшой доработки этой функции можно записывать, какая компонента содержит каждую вершину, вместо простой пометки компонент.

```
CountAndLabel(G):
    count ← 0
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            count ← count + 1
            LabelOne(v, count)
    return count
```

```
⟨⟨Пометка одной компоненты.⟩⟩
LabelOne(v, count):
    while the bag is not empty
        take v from the bag
        if v is unmarked
            mark v
            comp(v) ← count
        for each edge vw
            put w into the bag
```

Алгоритм WFSAll помечает каждую вершину один раз, помещает каждое ребро в мешок один раз и вынимает каждое ребро из мешка один раз, поэтому общее время выполнения $O(V + ET)$, где T – время одной операции с мешком. В частности, если выполняется поиск в глубину или поиск в ширину в каждой вершине, то итоговый алгоритм по-прежнему требует только времени $O(V + E)$.

Кроме того, поскольку алгоритм WhateverFirstSearch вычисляет остовное дерево одной компоненты, можно воспользоваться AFSAll для вычисления остовного леса всего графа в целом. В частности, поиск по первому наилучшему совпадению с весами ребер как приоритетами вычисляет остовный лес с минимальным весом за время $O(V + E \log E)$.

Крайне удивительно, что как минимум одна весьма широко известная книга по алгоритмам утверждает, что эту обертку можно использовать только при поиске в глубину.¹² Это абсолютно неверное утверждение. В действительности самая первая реализация поиска в ширину, опубликованная приблизительно в 1945 г. Конрадом Цузе (Konrad Zuse) на его протоязыке Планкалкюль, была разработана специально для подсчета и пометки компонент ненаправленного графа.

Направленные графы

Алгоритм поиска в любом направлении легко адаптировать к направленным графам, при этом единственное различие состоит в том, что при пометке вершины в мешок помещаются все ее соседи на выходе (out-neighbors). В действительности, если используются стандартные списки смежных вершин или матрицы смежности, код вообще не нужно изменять.

```
WhateverFirstSearch(s):
    put s into the bag
    while the bag is not empty
        take v from the bag
        if v is unmarked
            mark v
            for each edge  $v \rightarrow w$ 
                put w into the bag
```

В нашем предыдущем доказательстве предполагалось, что алгоритм пометает каждую вершину, достижимую из начальной вершины s , а направленные ребра $parent(v) \rightarrow p$ определяют дерево с корнем, в котором все ребра направлены от корня s . Но, даже если граф является связным, здесь уже нет необходимости в формировании остоного дерева графа, потому что достижимость теряет свойство симметричности.

Алгоритм `WhateverFirstSearch` действительно определяет остоное дерево вершин, достижимых из s . Кроме того, при изменении варианта «мешка» можно получить остоное дерево поиска в глубину и поиска в ширину, направленное остоное дерево минимального веса, дерево наикратчайшего пути или дерево пути максимальной ширины из этих достижимых вершин.

5.7. Редукция графа: сплошная заливка

Один из самых первых современных примеров поиска в любом направлении был предложен Эдвардом Муром (Edward Moore) в середине 1950-х гг. Карта пикселей (pixel map) – это двумерный массив, значения которого представляют цвета. Отдельные элементы массива называются пикселями

¹² Дословная цитата: «В отличие от поиска в ширину, при котором предшествующий подграф формирует дерево, предшествующий подграф, созданный при поиске в глубину, может состоять из нескольких деревьев, потому что поиск может повторяться из нескольких источников».

(pixels), это аббревиатура, полученная из словосочетания picture elements (элементы изображения)¹³. Связная область (connected region) в карте пикселей – это связанное подмножество пикселей, имеющих одинаковый цвет, где два пикселя считаются смежными (соседними), если они находятся в соседних позициях по горизонтали или по вертикали. Операция сплошной заливки (flood fill), часто представленная значком банки с краской в программах редактирования растровой графики, изменяет цвет каждого пикселя в связной области на новый. Входные данные для этой операции состоят из индексов i и j каждого пикселя в целевой области и нового цвета.

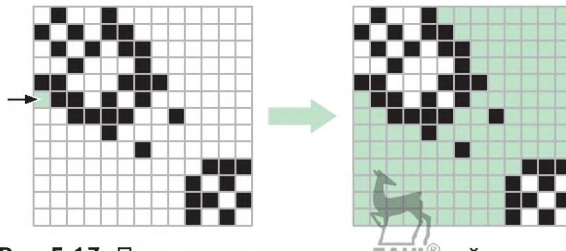


Рис. 5.13. Пример выполнения сплошной заливки

Задачу сплошной заливки можно свести к задаче определения достижимости, если изменить определения. Определим ненаправленный граф $G = (V, E)$, вершинами которого являются отдельные пиксели, а ребра соединяют соседние пиксели одинакового цвета. Каждая связная область в карте пикселей является компонентой графа G , следовательно, задача сплошной заливки сводится к задаче определения достижимости в графе G . Эту задачу достижимости можно решить, используя поиск в любом направлении в G , начиная с заданного пикселя (i, j) , с одним небольшим дополнением: при пометке вершины сразу же изменяется ее цвет. Для карты пикселей $n \times n$ граф G содержит n^2 вершин и не более $2n^2$ ребер, поэтому поиск в любом направлении выполняется за время $O(V + E) = O(n^2)$.

Этот простой пример демонстрирует самые важные составляющие метода редукции (reduction). Чтобы не решать задачу сплошной заливки с нуля, мы используем существующий алгоритм как подпрограмму типа черный ящик. Здесь совершенно не важно, как работает алгоритм поиска в любом направлении, значение имеет только его спецификация: задан граф G и начальная вершина s . Пометить каждую вершину G , которая достижима из s . Как и для любой другой подпрограммы, остается необходимость в описании того, как формируются входные данные и как используется вывод этой подпрограммы. Также необходимо проанализировать полученный ито-

¹³ До появления современных растровых дисплейных устройств в 1960-х гг. пикселы чаще называли stitches (швы, петли, стежки) или tesserae (мозаика, смальта, тессера) в зависимости от того, сделаны были они из нитей или из очень мелких камешков. Слово «pix» стало стандартным сокращением для «picture(s)» (изображение; изображения) в начале XX в. – вскоре после того, как слово «sock» стало общепринятым множественным числом слова «sock», – вытеснив более ранний разговорный вариант «piccy»; см. также «voxel» (volume element – элемент объема; воксел), «texel» (texture element – элемент текстуры; тексел) и «taxel» (tactile element – сенсорный элемент; таксел; он же badger).

вый алгоритм с учетом конкретных входных параметров, а не вершин и ребер какого-либо промежуточного графа, созданного применяемым алгоритмом.

У нас есть работающий алгоритм, но только теперь (и только теперь) можно применить два простых метода оптимизации, чтобы заставить его работать быстрее, – один метод практический, другой – теоретический:

- при практической реализации в действительности нет необходимости в создании отдельной графовой структуры данных для G . Вместо этого можно напрямую воспользоваться картой пикселей, как если бы она была стандартным списком смежных вершин, потому что можно перечислить всех соседей одинакового цвета для каждого пикселя за время $O(1)$. В сущности, нет необходимости в какой-либо особенной «пометке» вершин, вместо этого можно использовать цвет пикселей;
- при более тщательном анализе мы приходим к следующему выводу: время выполнения пропорционально числу пикселей в заливаемой области или, что равнозначно, числу вершин в компоненте графа G , содержащей вершину (i, j) , и это время выполнения может оказаться значительно меньшим, чем $O(n^2)$.

Упражнения

Графы

1. Доказать, что все приведенные ниже утверждения равнозначны.
 - Дерево – это связный ациклический граф.
 - Дерево – это одна компонента леса. (Лес – это ациклический граф.)
 - Дерево – это связный граф с количеством ребер не более $V - 1$.
 - Дерево – это минимально связный граф, в котором удаление любого ребра нарушает связность этого графа.
 - Дерево – это ациклический граф с количеством ребер не менее $V - 1$.
 - Дерево – это максимально ациклический граф, в котором ребро между любыми вершинами создает цикл.
 - Дерево – это граф, содержащий единственный в своем роде путь между каждой парой вершин.
2. Доказать, что любой связный ациклический граф с $n \geq 2$ вершинами содержит не менее двух вершин со степенью 1. Запрещено использовать слова «дерево» (tree), или «лист» (leaf), или любые хорошо известные свойства деревьев. Доказательство должно выводиться исключительно из определений «связный» (connected) и «ациклический» (acyclic).

3. Граф (V, E) является двудольным (bipartite), если вершины V можно разделить на два подмножества L и R , такие что каждое ребро имеет одну вершину в L , а другую – в R .
 - (a) Доказать, что каждое дерево является двудольным графом.
 - (b) Доказать, что граф G является двудольным, если и только если каждый цикл в G содержит четное число ребер.
 - (c) Описать и проанализировать эффективный алгоритм, определяющий, является ли заданный ненаправленный граф двудольным.
4. Когда собираются вместе группы голубей, они инстинктивно устанавливают иерархию в стае. Для любой пары голубей один голубь всегда клюет другого, отгоняя его от пищи или от потенциальных половых партнеров. Одна и та же пара голубей всегда выбирает одинаковый иерархический порядок отношений даже после нескольких лет раздельного существования вне зависимости от того, какие еще голуби их окружают. Удивительно, что общий порядок «клевания» может содержать циклы, например голубь i клюет голубя j , который клюет голубя k , который клюет голубя l , который клюет голубя i .
 - (a) Доказать, что любую конечную популяцию голубей можно разместить как участников некоторой процессии (возможно, на параде) так, чтобы каждый голубь клевал голубя, следующего за предшествующим. Пожалуйста, разберитесь как следует.
 - (b) Предположим, что задан направленный граф, представляющий иерархические отношения «клевания» в множестве из n голубей. Этот граф содержит по одной вершине для каждого голубя и ребро $i \rightarrow j$, если и только если голубь i клюет голубя j . Описать и проанализировать алгоритм вычисления иерархии для голубей, доказанной в части (a).
 - (c) Доказать, что для любого множества, состоящего как минимум из трех голубей, либо иерархия, описанная в части (a), является единственной в своем роде, либо существуют три голубя i, j и k , такие что голубь i клюет голубя j , который клюет голубя k , который клюет голубя i .
5. Маршрут Эйлера в графе G – это замкнутый маршрут обхода G , проходящий каждое ребро графа G ровно один раз.
 - (a) Доказать, что если связный граф G содержит маршрут Эйлера, то каждая вершина G имеет четную степень. (Эйлер доказал это.)
 - (b) Доказать, что если каждая вершина в связном графе G имеет четную степень, то G содержит маршрут Эйлера. (Эйлер не доказал это.)
 - (c) Описать и проанализировать алгоритм вычисления маршрута Эйлера в заданном графе или корректно сообщить о том, что такого маршрута не существует. (На это Эйлер неопределенно взмахивает руками.)

6. d -мерный гиперкуб – это граф, определяемый следующим образом. Существует 2^d вершин, каждая вершина помечена отличающейся от прочих строкой из d бит. Две вершины соединены ребром, если их метки отличаются ровно на один бит.
- (а) Гамильтонов цикл в графе G – это цикл из ребер G , при этом каждое ребро входит в каждую вершину G ровно один раз. Доказать, что для всех $d \geq 2$ d -мерный гиперкуб содержит гамильтонов цикл.
- (б) Какие гиперкубы содержат маршрут Эйлера (замкнутый маршрут, который проходит по каждому ребру ровно один раз)? [Подсказка: это очень простая задача.]

Алгоритмы обхода

7. Напомню, что направленный граф G является сильно связным, если для любых двух вершин u и v существует путь в графе G от u к v и путь в G от v к u .
Описать алгоритм, определяющий при заданном в качестве входных данных ненаправленном графе G , можно ли направить каждое ребро графа G так, чтобы итоговый граф стал сильно связным.
8. Пусть G – связный граф и пусть T – остовное дерево поиска в глубину для графа G с корнем в некотором узле v . Доказать, что если T также является остовным деревом поиска в ширину для графа G с корнем в v , то $G = T$.
9. Профессоры Эпприх (Epprich) и Гудстейн (Goodstein) предложили следующую оптимизацию обобщенного алгоритма поиска в любом направлении. Вместо проверки того факта, что извлекаемые из мешка вершины являются помеченными, их алгоритм проверяет тот же факт еще перед отправлением вершины в мешок, следовательно, гарантируется, что каждая вершина помещается в мешок не более одного раза. Их алгоритм также назначает родителя каждой вершине при ее пометке.

```
EagerWFS(s):
  mark s
  put s into the bag
  while the bag is not empty
    take v from the bag
    for each edge vw
      if w is unmarked
        mark w
        parent(w) ← v
        put w into the bag
```

- (а) Доказать, что алгоритм `EagerWFS(s)` помечает каждый узел, достижимый из s , и никакие другие узлы. Равнозначная формулировка задачи: доказать, что родительские ребра $v \rightarrow \text{parent}(v)$, вычисляемые алгоритмом `EagerWFS(s)`, определяют остовные компоненты, содержащей s .
- (б) Доказать, что если мешок реализован как очередь, то алгоритм `EagerWFS` равнозначен алгоритму поиска в ширину, и это означает, что оба алгоритма помечают одни и те же вершины в одном и том же порядке и создают одинаковое остовное дерево. [Подсказка: как определяется очередь?]
- (с) Доказать, что алгоритм `EagerWFS` никогда не равнозначен алгоритму поиска в глубину независимо от того, какая структура данных используется в качестве мешка (и, следовательно, в частном случае, когда мешок представлен стеком).

Ни `EagerWFS`, ни `RecursiveDFS` не определяют порядок, в котором рассматриваются ребра vw в каждой вершине v , а различные порядки рассмотрения ребер могут приводить к различным остовным деревьям. Таким образом, вы должны доказать для некоторого явно заданного графа G , что никакое остовное дерево для G , сгенерированное алгоритмом `RecursiveDFS`, не может быть сформировано алгоритмом `EagerWFS` (при использовании любой структуры данных для мешка), и наоборот.

10. Авторами одного из самых первых опубликованных описаний алгоритма поиска в любом направлении как обобщенного класса алгоритмов были Эдсгер Дейкстра (Edsger Dijkstra), Лесли Лэмпорт (Leslie Lamport), Элайн Мартин (Alain Martin), Капел Схольтен (Carel Scholten) и Элизабет Стеффенс (Elisabeth Steffens) в 1975 г. Это описание являлось частью проекта автоматического сборщика мусора (garbage collector). Вместо отслеживания помеченных и непомеченных вершин этот алгоритм использовал цвет для каждой вершины: белый, серый или черный. Как обычно, в приведенном ниже алгоритме мы мысленно представляем конкретный неявно заданный граф G .

```
ThreeColorSearch(s):
    color all nodes white
    color s gray
    while at least one vertex is gray
        ThreeColorStep()
```

```
ThreeColorStep():
    v ← any gray vertex
    if v has no white neighbors
        color v black
    else
        w ← any white neighbor of v
        parent(w) ← v
        color w gray
```

- (а) Доказать, что в алгоритме `ThreeColorSearch` всегда сохраняется следующий инвариант: ни одна черная вершина не является соседом белой вершины. [Подсказка: это должно быть просто.]

- (b) Доказать, что после завершения алгоритма `ThreeColorSearch(s)` все вершины, достижимые из s , являются черными, все вершины, не достижимые из s , являются белыми и что родительские ребра $v \rightarrow \text{parent}(v)$ определяют остовное дерево с корнем компоненты, содержащей s . [Подсказка: по интуиции черные узлы являются «помеченными», а серые узлы «помещены в мешок». Но, в отличие от формулировки алгоритма `WhateverFirstSearch`, алгоритм трех цветов не требует одновременной обработки всех ребер, выходящих из узла.]
- (c) Доказать, что приведенный ниже вариант алгоритма `ThreeColorSearch`, сохраняющий множество серых вершин в стандартном стеке, равнозначен алгоритму поиска в глубину. [Подсказка: порядок последних двух строк в псевдокоде подпрограммы `ThreeColorStackStep` весьма важен.]

```

ThreeColorStackSearch(s):
    color all nodes white
    color s gray
    push s onto the stack
    while at least one vertex is gray
        ThreeColorStackStep()

```

```

ThreeColorStackStep():
    pop v from the stack
    if v has no white neighbors
        color v black
    else
        w ← any white neighbor of v
        parent(w) ← v
        color w gray
    push v onto the stack
    push w onto the stack

```

- (d) Доказать, что приведенный ниже вариант алгоритма `ThreeColorSearch`, сохраняющий множество серых вершин в стандартной очереди, не равнозначен алгоритму поиска в ширину. [Подсказка: порядок последних двух строк в псевдокоде подпрограммы `ThreeColorQueueStep` не имеет никакого значения.]

```

ThreeColorQueueSearch(s):
    color all nodes white
    color s gray
    push s onto the queue
    while at least one vertex is gray
        ThreeColorQueueStep()

```

```

ThreeColorQueueStep():
    pull v from the queue
    if v has no white neighbors
        color v black
    else
        w ← any white neighbor of v
        parent(w) ← v
        color w gray
    push v into the queue
    push w into the queue

```

♥(е) Теперь предположим, что при выполнении алгоритма `ThreeColorSearch` существует еще один процесс: добавление ребер в граф G . Эти новые ребра могут нарушить корректность цветового инварианта, описанного в части (а), следовательно, нарушить корректность и всего алгоритма в целом, – в частности, после завершения `ThreeColorSearch` некоторые вершины, достижимые из s , могут оказаться белыми. Это привело бы к катастрофическим последствиям, если мы полагаемся на то, что «белая» (вершина) означает «недостижимая, следовательно, ее можно безопасно удалить».

Но если другой процесс явно сохраняет цветовой инвариант, то мы можем продолжать использование алгоритма трех цветов для безопасной идентификации недостижимых вершин. Смоделируем два параллельных алгоритма, как показано ниже. Конструкция выбора `either/or` в псевдокоде `GarbageCollect` и выбор одной из вершин u и w для изменения (`Mutate`) никоим образом не управляется основным алгоритмом.¹⁴

GarbageCollect(s):

```
color all vertices white
color s gray
while at least one vertex is gray
  either
    CollectStep()
  or
    Mutate()
```

CollectStep():

```
v ← any gray vertex
if v has no white neighbors
  color v black
else
  w ← any white neighbor of v
  color w gray
```

Mutate():

```
u ← any vertex
w ← any vertex
if uw is not an edge
  add edge uw
  if u is black and w is white
    color u gray
  if u is white and w is black
    color w gray
```

Доказать, что алгоритм `GarbageCollect` в итоге завершается с каждой вершиной, достижимой из s , окрашенной в черный цвет, и с каждой вершиной, не достижимой из s , окрашенной в белый цвет.

¹⁴ Это весьма существенное упрощение алгоритмов сборки мусора типа «mark and sweep» (пометить и стереть), реально используемых в многопоточных языках, таких как Lua и Go. К сожалению, более подробное обсуждение управления динамической памятью в многопоточном режиме не относится к теме этой книги, за исключением Первой заповеди: «Не сотвори собственный сборщик мусора».

- ♥(f) Предположим, что вместо перекрашивания черных вершин в серый цвет подпрограмма `Mutate` сохраняет цветовой инвариант, окрашивая некоторые белые вершины в серый цвет, как показано ниже.

```

Mutate():
  u ← any vertex
  w ← any vertex
  if uw is not an edge
    add edge uw
    if u is black and w is white
      color w gray
    if u is white and w is black
      color u gray

```

Доказать, что алгоритм `GarbageCollect` в итоге завершается с вершиной s , окрашенной в черный цвет, и каждая вершина, достижимая из черной вершины, окрашена в черный цвет, а каждая вершина, не достижимая из черной вершины, окрашена в белый цвет.

Сведения

11. Числовой лабиринт (number maze) – это сетка $n \times n$ из положительных целых чисел. В начале фишка находится в верхнем левом углу. Ваша цель – переместить фишку в нижний правый угол. При каждом ходе разрешается передвигать фишку вверх, вниз, влево или вправо, а расстояние, на которое передвигается фишка, определяется числом в текущей клетке. Например, если фишка находится в клетке с числом 3, то можно передвинуть ее на три клетки вверх, вниз, влево или вправо. Но никогда не разрешается перемещать фишку за границу доски.

Описать и проанализировать эффективный алгоритм, который либо возвращает минимальное число ходов, требуемых для решения заданного числового лабиринта, либо корректно сообщает, что данный лабиринт не имеет решения. Например, если задан числовой лабиринт, показанный на рис. 5.14, то алгоритм должен вернуть целое число 8.

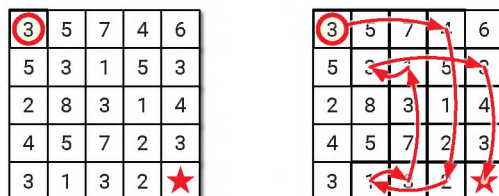


Рис. 5.14. Числовой лабиринт 5×5 , который можно решить за восемь ходов

12. Snakes and Ladders – классическая настольная игра, появившаяся в Индии не позже XVI в. Игровое поле представляет собой сетку из $n \times n$ клеток, последовательно пронумерованных от 1 до n^2 , начиная с нижнего левого угла. Строки нумеруются снизу вверх, и нумерация меняется попеременно слева направо и справа налево в каждой строке. Определенные пары клеток на этом поле всегда в разных строках соединены либо змеями (snakes) (ведущими вниз), либо лестницами (ladders) (ведущими вверх). Каждая клетка может быть конечным пунктом не более чем одной змеи или лестницы.

Игра начинается в клетке 1 в нижнем левом углу, где стоит фишка. При каждом ходе вы передвигаете фишку на k клеток при постоянном заданном значении k . Если ход завершается на верхнем конце змеи, то фишка скатывается вниз к хвосту змеи. А если ход заканчивается на нижнем конце лестницы, то фишка взбирается на верхний конец этой лестницы.

Описать и проанализировать алгоритм вычисления минимального числа ходов, требуемое для того, чтобы фишка дошла до самой последней клетки игрового поля (с номером 100).

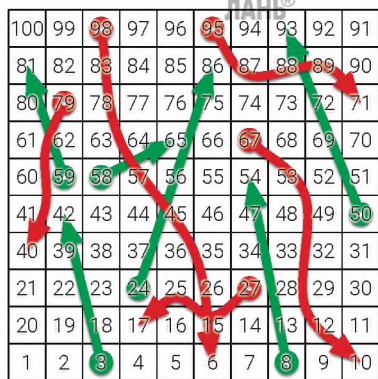


Рис. 5.15. Игровое поле Snakes and Ladders.

Прямые стрелки, направленные вверх, – лестницы.

Волнистые стрелки, направленные вниз, – змеи

13. Скандально известный монгольский специалист по головоломкам Видрах Итки Леда (Vidrach Itky Leda) в 1473 г. изобрел следующую головоломку. Эта головоломка состоит из сетки клеток $n \times n$, в которой каждая клетка помечена положительным целым числом, и двух фишек: красной и синей. Фишки всегда располагаются в различных клетках игрового поля. Начальные позиции фишек находятся в верхнем левом и нижнем правом углах поля. Цель этой головоломки – поменять местами фишки.

При каждом ходе можно передвигать фишку вверх, вправо, вниз или влево на расстояние, определяемое другой фишкой. Например, если

красная фишка находится на клетке с меткой 3, то можно переместить синюю фишку на три клетки вверх, влево, вправо или вниз. Но нельзя выводить фишку за пределы доски, и в конце каждого хода две фишки не должны располагаться на одной клетке.

Описать и проанализировать эффективный алгоритм, который либо возвращает минимальное число ходов, требуемое для решения заданной головоломки Видраха Итки Леда, либо корректно сообщает, что эта головоломка не имеет решения. Например, если задана головоломка, показанная на рис. 5.16, то алгоритм должен вернуть число 5.

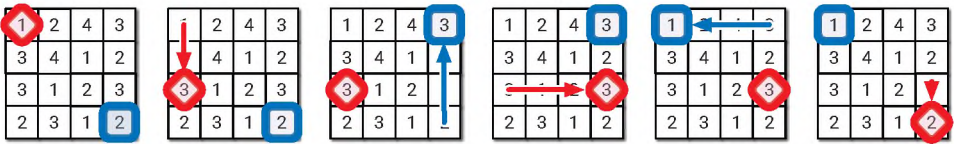


Рис. 5.16. Решение в пять ходов для головоломки Видраха Итки Леда размером 4×4

14. Предположим, что задан направленный граф $G = (V, E)$ и две вершины s и t . Описать и проанализировать алгоритм, определяющий, существует ли маршрут в G от s к t (возможно, с повторяющимися вершинами и/или ребрами), длина которого делится нацело на 3.

Например, если задан граф, показанный на рис. 5.17, с обозначенными вершинами s и t , то алгоритм должен вернуть True, поскольку маршрут $s \rightarrow w \rightarrow y \rightarrow x \rightarrow s \rightarrow w \rightarrow t$ имеет длину 6.

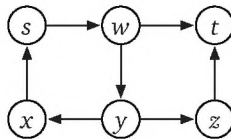


Рис. 5.17. Пример графа для задания 14

15. Предположим, что задан направленный граф G , в котором некоторые ребра красные, а остальные синие. Описать алгоритм поиска наикратчайшего маршрута в G от одной вершины s до другой вершины t , в котором нет трех последовательных ребер одинакового цвета. То есть если маршрут содержит два красных ребра подряд (одно за другим), то следующее ребро обязательно должно быть синим, а если два синих ребра следуют друг за другом, то следующее ребро непременно должно быть красным.

Например, если задан исходный граф, показанный на рис. 5.18, алгоритм должен вернуть целое число 7, потому что $s \rightarrow a \rightarrow b \Rightarrow d \rightarrow c \Rightarrow a \rightarrow b \rightarrow t$ – наикратчайший разрешенный маршрут от s к t .

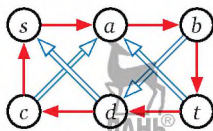


Рис. 5.18. Пример графа для задания 15

16. Рассмотреть направленный граф G , в котором каждое ребро окрашено в красный, белый или синий цвет. Маршрут в графе G называется маршрутом флага Франции, если включает последовательность ребер: красное, белое, синее, красное, белое, синее и т. д. Более формально: маршрут $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k$ является флагом Франции, если для каждого целого числа i ребро $v_i \rightarrow v_{i+1}$ красное, если $i \bmod 3 = 0$, белое, если $i \bmod 3 = 1$, и синее, если $i \bmod 3 = 2$.

Описать алгоритм поиска всех вершин в графе G , достижимых из заданной вершины v по маршруту флага Франции.

17. Существует n галактик, связанных m межгалактическими маршрутами телепортации. Каждый маршрут телепортации соединяет две галактики, и по нему можно перемещаться в обоих направлениях. Кроме того, для каждого маршрута телепортации e установлена соответствующая стоимость $c(e)$ долл., где $c(e)$ – положительное целое число. Маршрут телепортации можно использовать многократно, но при каждом использовании непременно нужно оплачивать его.

Джуди хочет переместиться из галактики s в галактику t , но процедура телепортации не очень приятна, поэтому Джуди предпочла бы минимизировать количество необходимых телепортаций. Но она также хочет, чтобы суммарная стоимость была кратна пяти долларам, поскольку носить с собой сдачу мелочью не слишком удобно.

- (а) Описать и проанализировать алгоритм вычисления минимального числа телепортаций, необходимых Джуди для перемещения из галактики s в галактику t , такого что общая стоимость путешествия была бы кратной пяти долларам.
 - (б) Решить задание (а), но теперь предположить, что у Джуди есть купон, позволяющий бесплатно использовать ровно один маршрут телепортации.
18. Three Seashells (Три раковины) – игра типа «солитер» в связном не-направленном графе G . В начальной позиции три фишки находятся на различных исходных вершинах a, b, c . При каждом ходе необходимо обязательно переместить все три фишки – каждую вдоль ребра, исходящего из текущей вершины в соседнюю. В конце каждого хода три фишки непременно должны находиться на трех различных вершинах. Цель игры – переместить фишки в три конечные вершины

x, y, z , при этом неважно, в какой конечной вершине окажется каждая фишка.

Описать и проанализировать алгоритм, определяющий, разрешима ли заданная головоломка Three Seashells. Входные данные состоят из графа G , начальных вершин a, b, c и конечных вершин x, y, z . Вывод состоит из одного бита: True или False.

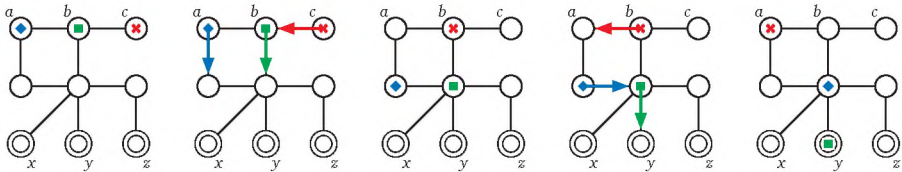


Рис. 5.19. Начальная конфигурация головоломки Three Seashells и первые два хода ее решения

19. Пусть G – связный ненаправленный граф. Предположим, что мы начинаем с двумя монетами в двух произвольно выбранных вершинах графа G и хотим переместить их так, чтобы они оказались в одной вершине, используя для этого минимальное число перемещений. При каждом ходе каждая монета обязательно должна передвигаться только на соседнюю вершину.
 - (a) Описать и проанализировать алгоритм вычисления минимального числа ходов для достижения конфигурации, в которой обе монеты находятся в одной вершине, или корректно сообщаящего о невозможности достижения такой конфигурации. Входные данные для этого алгоритма состоят из графа $G = (V, E)$ и двух вершин $u, v \in V$ (вершины могут или не могут быть различными).
 - (b) Теперь предположим, что в игре участвуют три монеты. Описать и проанализировать алгоритм, вычисляющий минимальное число ходов для достижения конфигурации, в которой все три монеты находятся в одной вершине, или корректно сообщаящего о невозможности достижения такой конфигурации.
 - (c) Наконец, предположим, что играют 42 монеты. Описать и проанализировать алгоритм, определяющий, возможно ли переместить все 42 монеты в одну вершину. И в этом случае при каждом ходе обязательно должна перемещаться каждая монета. Для получения наивысшей оценки алгоритм должен выполняться за время $O(V + E)$.
20. В одном из учебных пособий для начальной школы¹⁵, где учится моя дочь, содержится несколько головоломок следующего типа, как показано на рис. 5.20.

¹⁵ Jason Batterson and Shannon Rogers, Beast Academy Math: Practice 3A, 2012. Несколько дополнительных примеров см. здесь: <https://www.beastacademy.com/resources/printables.php>.

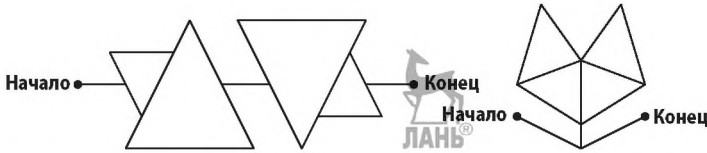


Рис. 5.20. Пройдите каждый показанный здесь угловой лабиринт по такому пути от начала до конца, который содержит только острые углы

Описать и проанализировать алгоритм решения произвольных остроугольных лабиринтов.

Задан связный ненаправленный граф G , вершинами которого являются точки на плоскости, а ребрами – отрезки прямых. Ребра не представляют интереса, за исключением их конечных точек. Например, для изображения буквы X потребуется пять вершин и четыре ребра, а первый лабиринт на рис. 5.20 содержит 18 вершин и 21 ребро. Также заданы две вершины Начало и Конец.

Алгоритм должен возвращать True, если G содержит маршрут от Начала к Концу, в котором имеются только острые углы, иначе должно возвращаться значение False. Формально маршрут в графе G является допустимым, если для любых двух следующих друг за другом ребер $u \rightarrow v \rightarrow w$ в маршруте либо $\angle uvw = \pi$, либо $0 < \angle uvw < \pi/2$. Предположим, что имеется подпрограмма, которая может определить за время $O(1)$, является ли угол между двумя заданными отрезками развернутым, тупым, прямым или острым.

21. Предположим, что задано множество n горизонтальных и вертикальных отрезков прямых и две точки s и t на плоскости. Описать эффективный алгоритм, определяющий, существует ли путь от s к t , который не пересекает ни один из заданных отрезков.

Каждый горизонтальный отрезок определяется своей левой и правой координатой x и единственной в своем роде координатой y . Каждый вертикальный отрезок определяется единственной в своем роде координатой x и координатами y верхнего и нижнего конца. Каждая точка s и t определяется собственными координатами x и y .

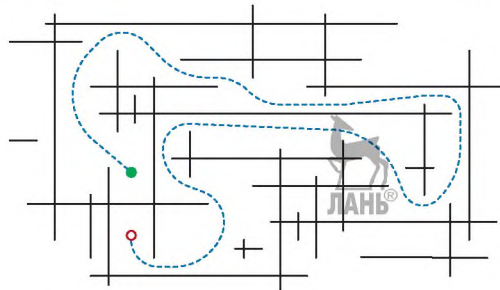


Рис. 5.21. Путь между двумя точками в лабиринте из горизонтальных и вертикальных отрезков прямых

22. В каждом веселом романтическом фильме есть сцена, где двое влюбленных после долгой и досадной разлуки внезапно видят друг друга на большом расстоянии, а затем медленно приближаются, постоянно глядя в глаза, и начинает звучать музыка, дождь прекращается, и солнце выходит из-за облаков, музыка звучит все громче, и все начинают танцевать на радугах, котята и шоколадные единороги и...

Предположим, что романтическая пара влюбленных – по великой традиции ИТ их зовут Алиса (Alice) и Боб (Bob) – входит в свой любимый парк с восточного и западного входа и сразу же устанавливает зрительный контакт. Они не могут сразу же броситься друг другу навстречу, вместо этого они непременно должны оставаться на тропинке, которая проложена зигзагами между восточным и западным входами в парк. Чтобы сохранить должное драматическое напряжение, Алиса и Боб должны идти по этой тропинке так, чтобы всегда оставаться на прямой линии с направлением восток–запад.

Можно описать зигзагообразную тропинку как два массива $X[0..n]$ и $Y[0..n]$, содержащие координаты x и y всех углов тропинки в порядке от юго-западной конечной точки до юго-восточной конечной точки. Массив X отсортирован в возрастающем порядке, а $Y[0] = Y[n]$. Тропинка представляет собой последовательность отрезков прямых, соединенных в углах с заданными координатами.

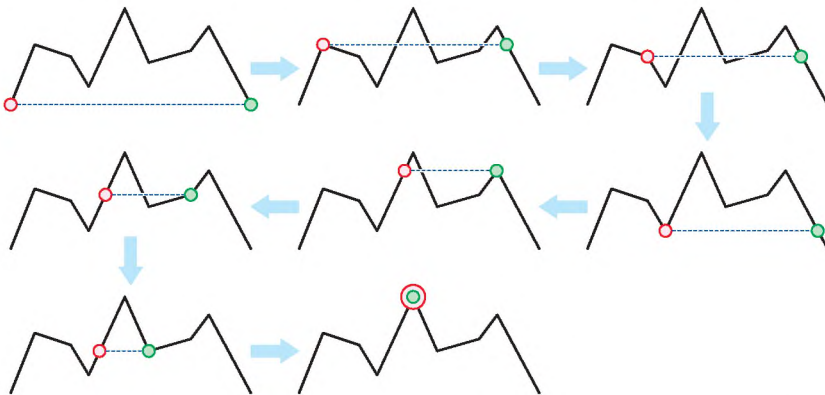


Рис. 5.22. Алиса и Боб встречаются. Алиса пятится назад на шаге 2, а Боб пятится назад на шагах 5 и 6

- (а) Предположим, что $Y[0] = Y[n] = 0$ и $Y[i] > 0$ для всех прочих индексов i , т. е. конечные точки тропинки расположены строго ниже любой другой точки этой тропинки.

Доказать, что на любой тропинке P , соответствующей вышеописанным условиям, Алиса и Боб всегда могут встретиться. [Подсказка: нужно описать граф, моделирующий все возможные локации этой пары на тропинке. Что обозначают вершины такого графа?

Что обозначают ребра? Используйте лемму рукопожатия: каждый граф содержит четное число вершин с нечетной степенью.]

- (b) Если конечные точки тропинки не находятся ниже любой другой вершины, то Алиса и Боб все еще могут встретиться, но, возможно, и нет. Описать алгоритм, определяющий, могут ли встретиться Алиса и Боб, не разрывая визуальный контакт по линии восток-запад и не сходя с тропинки, если в качестве входных данных заданы массивы $X[0..n]$ и $Y[0..n]$.

♥(c) Описать алгоритм для части (b), выполняемый за время $O(n)$.

23. Известный автор головоломок Кэниел «Датчанин» Дейн (Kaniel the Dane; анаграмма Daniel Kane) придумал игру типа «солитер» с двумя фишками на квадратном поле $n \times n$. Некоторые клетки игрового поля помечены как препятствия, а одна из клеток помечена как цель. При каждом ходе игрок обязан передвинуть одну из фишек из ее текущего положения настолько далеко, насколько это возможно, вверх, вниз, вправо или влево, останавливаясь только если фишка встречается (1) край игрового поля, (2) клетку с препятствием или (3) другую фишку. Задача игрока – привести одну из фишек в целевую клетку.

Например, можно решить головоломку, показанную на рис. 5.23, перемещая красную фишку вниз до встречи с препятствием, затем переместив зеленую фишку влево до встречи с красной фишкой, после чего передвигать красную фишку влево, вниз, вправо и вверх.

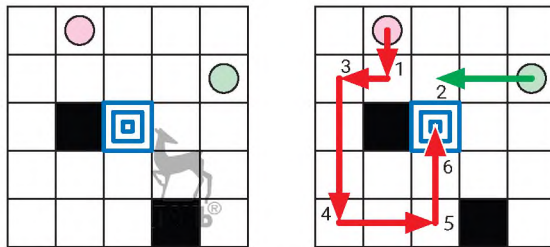


Рис. 5.23. Экземпляр головоломки Кэниела «Датчанина» Дейна, которую можно решить за шесть ходов. Круги обозначают начальные позиции фишек, черные квадраты – препятствия, центральная клетка – целевая

Описать и проанализировать алгоритм, определяющий, можно ли решить конкретный экземпляр этой головоломки. Входные данные состоят из целого числа n , списка локаций препятствий, локаций цели и начальных позиций фишек. Вывод алгоритма – единственное логическое значение: True, если заданная головоломка разрешима,

иначе – False. [Подсказка: не забывайте о времени, требуемом для формирования графа.]

- ♥24. Rectangle Walk (Блуждание прямоугольника) – новая абстрактная игра-головоломка, которую можно купить всего лишь за 99 центов для Steam, iOS, Android, Xbox One, Playstation 5, Nintendo Wii U, Atari 2600, Palm Pilot, Commodore 64, TRS-80, Sinclair ZX-1, DEC PDP-8, PLATO, Zuse Z3, Duramesc, арифмометра Однера (Odhner Arithmometer), вычислительной машины Чарльза Бэббиджа (Charles Babbage) Analytical Engine, ткацкого станка Жакарда (Jacquard Loom), астрономических часов в Лунде (Швеция) (Horologium Mirabile Lundense), арифмометра Лейбница (Leibniz Stepped Reckoner), ансамбля роботов Аль-Джазари (Al-Jazari's Robot Band), автомата Янь Ши (Yan Shi's Automaton), антикитерского механизма (Antikythera Mechanism), веревки с узлами (Knotted Rope), кости Ишанго (Ishango Bone) и кучки камней (Pile of Rocks).

Для игры необходимо поле $n \times n$ клеток черного и белого цвета. Игрок перемещает прямоугольник по этому полю, выполняя следующие условия:

- прямоугольник обязательно должен быть выровнен по клеткам поля, т. е. координаты верхней, нижней, левой и правой сторон должны быть целыми числами;
- прямоугольник непременно должен находиться целиком внутри игрового поля и обязательно содержать не менее одной клетки поля;
- прямоугольник не должен содержать черные клетки;
- во время одного хода игрок может заменить текущий прямоугольник r на любой прямоугольник r' , который содержит r или содержится в r .

В начале игры прямоугольник игрока размером 1×1 клеток находится в верхнем левом углу. Цель игрока – достичь клетки (прямоугольник 1×1) в нижнем правом углу, используя для этого как можно меньше ходов.

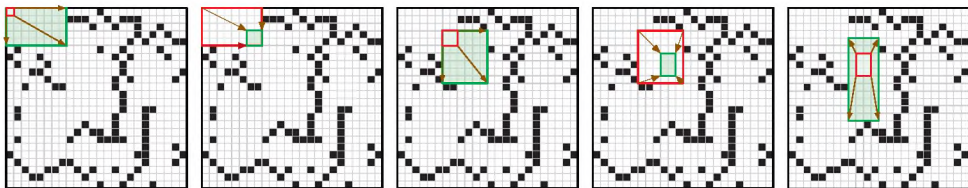


Рис. 5.24. Первые пять ходов игры Rectangle Walk

Описать и проанализировать алгоритм вычисления длины наикратчайшего маршрута блуждания прямоугольника в заданной битовой

карте. Входные данные состоят из массива $M[1..n, 1..n]$, где $M[i, j] = 1$ обозначает черную клетку, а $M[i, j] = 0$ – белую клетку. Предположить, что существует верный маршрут блуждания прямоугольника, а именно $M[1, 1] = M[n, n] = 0$. Например, если задана битовая карта, приведенная на рис. 5.24, то алгоритм должен вернуть целое число 18. [Подсказка: не забывайте о времени, требуемом для формирования графа.]

25. В игре Racetrack (также Graph Racers и Vector Rally) два игрока с помощью бумаги и карандаша имитируют автогонку. В эту игру Джефф играл в школьном автобусе, когда учился в 5 классе.¹⁶ Игра заключается в изображении пути движения автомобиля на листке бумаги в клетку. Игроки поочередно выбирают последовательность клеток, представляющих маршрут движения автомобиля по гоночному треку, при этом соблюдая определенные условия, описанные ниже.

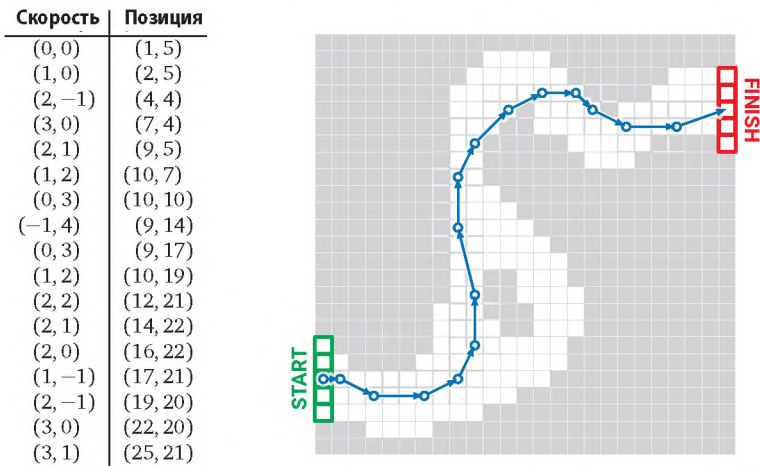


Рис. 5.25. Маршрут в игре Racetrack из 16 ходов на треке 25×25.
Это не самый короткий проход данного трека

Характеристиками каждого автомобиля являются позиция и скорость, определяемые координатами x и y . Некоторое подмножество клеток помечается как стартовая зона, другое подмножество клеток помечается как финишная зона. Начальную позицию автомобиля выбирает игрок в любой клетке стартовой зоны. Начальная скорость каждого автомобиля всегда равна $(0, 0)$. При каждом ходе игрок может (необязательно) увеличивать или уменьшать одну из координат или обе координаты скорости, другими словами, каждая компонента скорости может изменяться не более чем на 1 в течение одного хода. После этого новая позиция автомобиля определяется прибавлением нового значения скорости к предыдущей позиции. Новая позиция

¹⁶ Настоящая игра немного сложнее, чем описанная здесь версия. Превосходную онлайн-версию можно найти здесь: <http://hammade.com/vectorracer/>.

обязательно должна находиться в пределах гоночного трека, иначе автомобиль разбивается, и игрок выбывает из гонки. Гонка завершается, когда первый автомобиль достигает позиции, расположенной в финишной зоне.

Предположим, что гоночный трек представлен массивом битов $n \times n$, где каждый бит 0 обозначает клетку внутри трека, а каждый бит 1 – клетку вне трека. «Стартовая зона» находится в первом столбце, а «финишная зона» – в последнем столбце.

Описать и проанализировать алгоритм поиска минимального числа ходов, требуемых для передвижения автомобиля с линии старта до линии финиша на заданном гоночном треке.

26. Лабиринт катящегося кубика (rolling die maze) – головоломка с использованием обычного шестигранного кубика (с числами на каждой грани) и поля из квадратных клеток. Вы должны мысленно представить, что это поле лежит на столе, а кубик всегда находится на поле и накрывает ровно одну клетку поля. При каждом шаге вы перекачиваете кубик с поворотом на 90° относительно одного из его нижних ребер, перемещаясь на соседнюю клетку на север, юг, восток или запад.

Некоторые клетки поля могут быть заблокированы, кубику никогда не разрешается останавливаться на заблокированном поле. Некоторые клетки могут быть помечены числом, и когда кубик останавливается на такой помеченной клетке, число на верхней грани обязательно должно быть равно числу-метке. Непомеченные и незаблокированные клетки являются свободными. Нельзя выкатывать кубик за пределы игрового поля. Задача лабиринта катящегося кубика является разрешимой, если возможно размещение кубика в нижней левой угловой клетке и перекач его в верхнюю правую угловую клетку с учетом описанных выше ограничений.

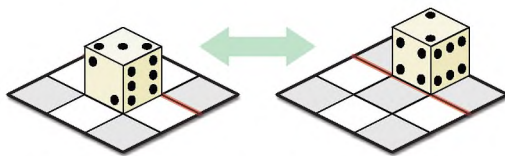


Рис. 5.26. Перекач кубика (один ход)

На рис. 5.27 показаны четыре лабиринта катящегося кубика. Предположим, что используется стандартный игровой кубик с точками, обозначающими числа от 1 до 6 на противоположных гранях. Разрешимыми являются только первые два лабиринта. Например, решить первый лабиринт можно, поместив кубик в нижний левый угол с 1 на верхней грани, затем покатав кубик на восток, потом два раза на север и снова на восток.

- (а) Предположить, что входными данными является двумерный массив $L[1..n, 1..n]$, где в каждом элементе $L[i, j]$ хранится метка клетки в i -й строке и j -м столбце, при этом 0 означает, что клетка свободна, а -1 – что клетка заблокирована.

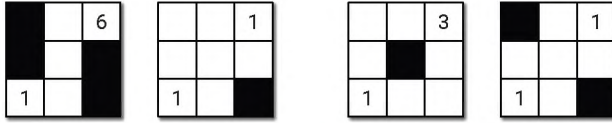


Рис. 5.27. Четыре примера лабиринта катящегося кубика, из которых разрешимыми являются только первые два

Описать и проанализировать алгоритм, выполняемый за полиномиальное время и определяющий, является ли разрешимым заданный лабиринт катящегося кубика.

- ♥(b) Теперь предположим, что лабиринт задан неявно списком помеченных и заблокированных клеток. А именно: предположим, что входные данные состоят из целого числа M , определяющего высоту и ширину лабиринта и массива $S[1..n]$, где каждый элемент $S[i]$ – тройка чисел (x, y, L) , означающая, что клетка (x, y) содержит метку L . Как и при явном кодировании, метка -1 означает, что клетка заблокирована, а свободные клетки вообще не перечисляются в массиве S . Описать и проанализировать эффективный алгоритм определения разрешимости заданного лабиринта катящегося кубика. Для получения максимальной оценки время выполнения алгоритма должно быть полиномиальным относительно размера входных данных n .

[Подсказка: вам предоставляется некоторая свобода при выборе начальной позиции кубика. Существуют лабиринты катящегося кубика, которые можно решить только при правильном выборе начальной позиции.]

- ♥27. Предположим, что задан произвольный направленный граф G , в котором каждое ребро окрашено в красный или синий цвет, с двумя особыми вершинами s и t .

- (а) Описать алгоритм, который вычисляет маршрут от s до t , такой что набор красных и синих ребер в этом маршруте является палиндромом, или корректно сообщает, что такого маршрута не существует.
- (b) Описать алгоритм, вычисляющий наикратчайший маршрут от s к t , такой что набор красных и синих ребер в этом маршруте является палиндромом или корректно сообщает, что такого маршрута не существует.

[Подсказка: где мы в последний раз видели палиндромы?]

- ♠♥28. Шашки (в США их называют checkers) – это игра на доске из $m \times m$ черно-белых клеток¹⁷. Обычно играют на доске 8×8 или 10×10 , но правила легко обобщить для доски любого размера. Каждая темная клетка занята не более чем одной шашкой (называемой в США checker) черного или белого цвета. Светлые клетки всегда пусты. Один участник играет белыми шашками (White), другой – черными (Black). Игрок проигрывает, когда его последняя шашка удаляется с доски.

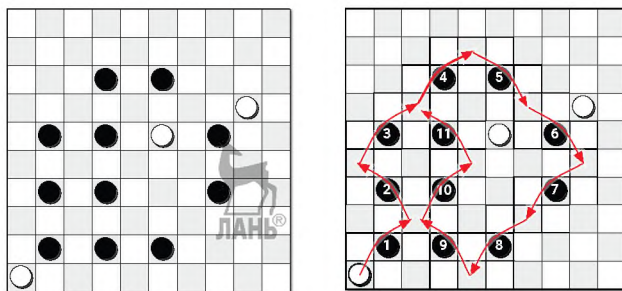


Рис. 5.28. Белые выигрывают в один ход

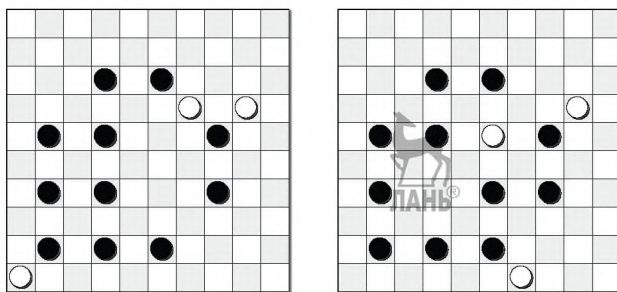


Рис. 5.29. Белые не могут выиграть в один ход в любой из показанных здесь позиций

Рассмотрим следующую простую версию этой игры, по существу представляющую собой обычную игру в шашки (в Америке – checkers, в Великобритании – draughts), но в нашей версии каждая шашка является дамкой¹⁸. Шашки можно передвигать в любом из четырех диагональных направлений. При каждом ходе игрок либо передвигает одну из своих шашек по диагонали на пустую соседнюю клетку либо выполняет последовательность прыжков одной из своих шашек. При каждом прыжке шашка перемещается на две клетки

¹⁷ Счетные столы, использовавшиеся средневековыми английскими учетчиками государственных финансов, были покрыты зеленой тканью с черными клетками «в шашечку», счетчики в форме диска размещались в этих клетках для представления числовых значений. Поэтому британские учетчики государственных финансов с X в. стали известны под общим названием Exchequer. Такие реальные счетные столы использовались эксчекерами для вычисления налоговых платежей почти до конца XIX в.

¹⁸ В большинстве других вариантов шашек имеются «летающие дамки» (flying kings), поведение которых сильно отличается от дамек в британской/американской версии игры, что значительно усложняет эту задачу, как мы увидим в главе 12.

по диагонали от начальной клетки, но только если промежуточная клетка занята шашкой противоположного цвета, при этом шашка противника считается побитой и немедленно удаляется с доски. Все прыжки в течение одного хода обязательно должны выполняться одной и той же шашкой.

Описать алгоритм, определяющий, могут ли белые побить каждую черную шашку, следовательно, выиграть партию за один ход. Входные данные состоят из значения ширины доски m , списка позиций белых и черных шашек. Для получения максимальной оценки алгоритм должен выполняться за время $O(n)$, где n – общее количество шашек. [Подсказка: жадная стратегия – выполнение произвольно выбираемых прыжков, пока не застрянем, – далеко не всегда позволяет найти победную комбинацию, даже если она существует. Смотрите задание 5. Четность, четность, четность.]



Глава 6



Поиск в глубину

*And, for the hous is crinkled to and fro,
And hath so queinte weyes for to go —
For hit is shapen as the mase is wrought —
Therto have I a remedie in my thoght,
That, by a dewe of twyne, as he hath goon,
The same wey he may returne anoon,
Folwing alwey the threed, as he hath come.*

— Джеффри Чосер (Geoffrey Chaucer), «Легенда о славных женщинах»
(*The Legend of Good Women*) (около 1385 г.)

«Как хорош мир и как отвратительны лабиринты», — с облегчением произнес я.

«Как хорош был бы мир, если бы имелось правило хождения по лабиринтам», — ответил учитель.



— Умберто Эко (Umberto Eco), «Имя розы»
(*Il nome della rosa*) (1980 г.)

В предыдущей главе мы рассматривали обобщенный алгоритм поиска в любом направлении для обхода произвольных ненаправленных и направленных графов. В этой главе мы сосредоточимся на конкретном варианте этого алгоритма, который называется поиском в глубину (depth-first search), и главным образом на поведении этого алгоритма в направленных графах.

Несмотря на то что алгоритм поиска в глубину может быть в точности описан как «поиск в любом направлении с применением стека», обычно он реализуется рекурсивно, без явного использования стека:

```
DFS(v):  
  if v is unmarked  
    mark v  
  for each edge v → w  
    DFS(w)
```

Можно сделать этот алгоритм немного быстрее (на практике), проверяя, помечен ли узел, перед его рекурсивным использованием. Это изменение гарантирует, что $\text{DFS}(v)$ вызывается только один раз для каждой вершины v . В дальнейшем можно изменить алгоритм для получения другой полезной информации о вершинах и ребрах, добавив две подпрограммы типа «черный ящик»: PreVisit и PostVisit , которые мы пока оставим неопределенными.

```

DFS(v);
mark v
PreVisit(v)
for each edge vw
  if w is unmarked
    parent(w) ← v
    DFS(w)
PostVisit(v)

```

Напомним, что узел w является достижимым из другого узла v в направленном графе, G или, проще говоря, v может достичь w , если и только если G содержит направленный путь из v в w . Пусть $\text{reach}(v)$ обозначает множество вершин, достижимых из v (включая саму вершину v). Если снять метки всех вершин в графе G , а затем вызвать $\text{DFS}(v)$, то множество помеченных вершин будет в точности соответствовать $\text{reach}(v)$.

Достижимость в ненаправленных графах симметрична: v может достичь w , если и только если w может достичь v . В результате после снятия меток со всех вершин в ненаправленном графе G вызов $\text{DFS}(v)$ выполняет обход всей компоненты v , а указатели на родителей определяют остовное дерево этой компоненты.

В направленных графах положение более изощренное, как показано на рис. 6.1. Даже если граф «связный», разные вершины могут достигать разных и, возможно, пересекающихся частей графа. Указатели на родителей, присвоенные $\text{DFS}(v)$, определяют дерево с корнем в вершине v , в котором вершины в точности соответствуют $\text{reach}(v)$, но это дерево необязательно представляет собой остовное дерево этого графа.

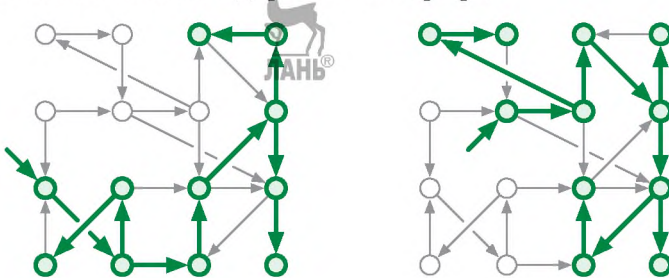


Рис. 6.1. Деревья поиска в глубину с корнями в различных вершинах одного и того же направленного графа

Как обычно, можно расширить этот алгоритм определения достижимости для обхода всего исходного графа в целом, даже если он несвязный, используя для этого стандартную функцию-обертку, показанную слева на рис. 6.2. Здесь мы добавляем обобщенную подпрограмму типа «черный ящик» *Preprocess* для выполнения любой необходимой предварительной обработки для функций *PreVisit* и *PostVisit*.

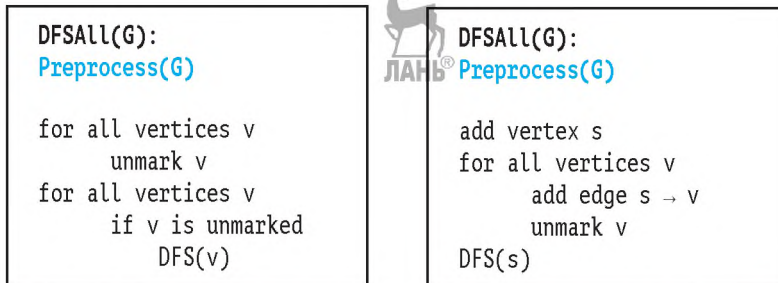


Рис. 6.2. Две формулировки стандартного алгоритма-обертки для поиска в глубину

Другой вариант: если разрешено изменять граф, то можно добавить новую исходную вершину s с ребрами в каждую другую вершину в графе G , а затем выполнить один вызов $DFS(s)$, как показано справа на рис. 6.2. В этом случае получившиеся указатели на родителей всегда определяют остовное дерево пополненного начального графа, но не исходного начального графа. В остальном эти две функции-обертки обладают совершенно одинаковым поведением, поэтому выбор одной из них – исключительно вопрос удобства.¹

Кроме того, поведение этого алгоритма немного различается для ненаправленных и направленных графов. В ненаправленных графах, как вы видели в предыдущей главе, легко адаптировать *DFSAll* для подсчета компонент графа. В частности, указатели на родителей, вычисленные *DFSAll*, определяют остовный лес исходного графа, содержащий остовное дерево для каждой компоненты. Но если граф направленный, то *DFSAll* может обнаружить любое число «компонент» от 1 до V , даже если этот граф является «связным», в зависимости от точной структуры графа и порядка, в котором функция-обертка алгоритма посещает вершины.

6.1. Обход в прямом и обратном порядке

Надеюсь, вы уже знакомы с методами обхода в прямом и обратном порядке (обхода с предварительной и отложенной выборкой) деревьев с корнями. Оба варианта обхода можно вычислить, используя поиск в глубину. Аналогичные порядки обхода можно определить для произвольных направленных

¹ Равнозначность этих двух функций-обертки – особое свойство поиска в глубину. В сущности, обертывание поиска в ширину циклом *for* для посещения каждой вершины не позволяет получить тот же порядок обхода, что и при добавлении исходной вершины и вызове функции поиска в ширину в этой вершине s .

ных графов, даже если они несвязные, – посредством передачи счетчика, как показано на рис. 6.3. Точно так же можно использовать наш обобщенный алгоритм поиска в глубину с приведенными ниже подпрограммами Preprocess, PreVisit и PostVisit.

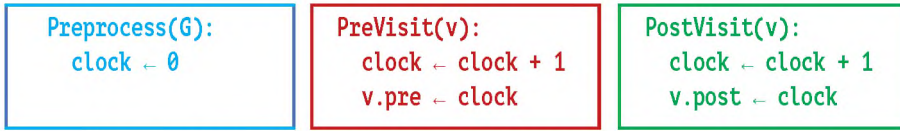


Рис. 6.3. Определение обхода в прямом и обратном порядке через поиск в глубину

При любой формулировке этот алгоритм присваивает $v.pre$ (и переводит часы вперед) сразу после записи вершины v в стек рекурсии, а также присваивает $v.post$ (и переводит часы вперед) непосредственно перед извлечением v из стека рекурсии. Из этого следует, что для любых двух вершин u и v отрезки $[u.pre, u.post]$ и $[v.pre, v.post]$ либо непересекающиеся, либо вложенные. Кроме того, $[u.pre, u.post]$ содержит $[v.pre, v.post]$, если и только если $DFS(v)$ вызывается во время выполнения $DFS(u)$, или, что равнозначно, если и только если u является предком v в конечном лесу указателей на родителей.

После того как $DFSAll$ помечает каждый узел в графе, метки $v.pre$ определяют прямой порядок обхода вершин, а метки $v.post$ – обратный порядок обхода². С некоторыми тривиальными исключениями в каждом графе имеется несколько различных прямых и обратных порядков обхода, зависящих от порядка, в котором DFS рассматривает ребра, исходящие из каждой вершины, и от порядка, в котором $DFSAll$ рассматривает вершины.

В оставшейся части этой главы мы будем называть $v.pre$ начальным временем v (или менее формально: «когда v начинается»), $v.post$ – конечным временем v (или менее формально: «когда v завершается»), а отрезок между начальным и конечным временем – активным интервалом v (или менее формально: «пока v активна»).

Классификация вершин и ребер

При выполнении $DFSAll$ каждая вершина v исходного графа находится в одном из трех состояний:

- **новая** (вершина), если функция $DFS(v)$ не была вызвана, т. е. если $clock < v.pre$;
- **активная** (вершина), если функция $DFS(v)$ была вызвана, но возврата из нее не было, т. е. если $v.pre \leq clock < v.post$;
- **завершенная** (вершина), если произошел возврат из функции $DFS(v)$, т. е. если $v.post \leq clock$.

² Иногда оба этих порядка ошибочно называют «порядком обхода в глубину» (depth-first ordering). Так делать не следует.

Поскольку начальное и конечное время соответствует операциям записи и извлечения из стека рекурсии, вершина является активной, если и только если она находится в стеке рекурсии. Из этого следует, что активные узлы всегда образуют направленный путь в графе G .

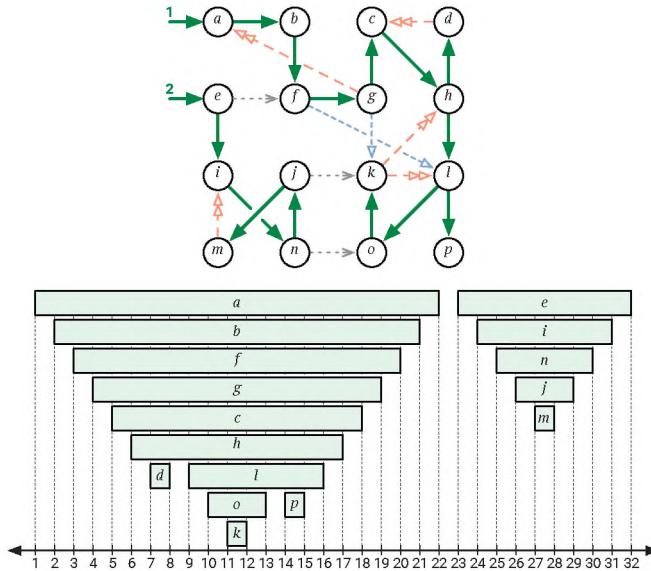


Рис. 6.4. Лес направленного графа поиска в глубину и соответствующие активные интервалы его вершин, определяющие прямой «abfgchdllokpeijnm» и обратный «dkoplhcgfbamjnle» порядок обхода. Ребра леса обозначены сплошными стрелками, смысл штриховых стрелок будет описан на рис. 6.5

Ребра исходного графа разделяются на четыре класса в зависимости от того, как пересекаются их активные отрезки. Рассмотрим уже хорошо знакомое ребро $u \rightarrow v$.

- Если v – новая вершина при начале работы $\text{DFS}(u)$, то функция $\text{DFS}(v)$ обязательно должна вызываться при выполнении $\text{DFS}(u)$ напрямую или через какие-либо промежуточные рекурсивные вызовы. В любом случае u является истинным предком v в лесу поиска в глубину, и $u.pre < v.pre < v.post < u.post$.
 - ♦ Если $\text{DFS}(u)$ вызывает $\text{DFS}(v)$ напрямую, то $u = v.parent$ и $u \rightarrow v$ называется *ребром дерева* (tree edge).
 - ♦ Иначе $u \rightarrow v$ называется *прямым ребром* (forward edge).
- Если v – активная вершина при начале работы $\text{DFS}(u)$, то она уже находится в стеке рекурсии, что подразумевает противоположный порядок вложенности $v.pre < u.pre < u.post < v.post$. Кроме того, граф G обязательно должен содержать прямой путь от v к u . Ребра, соответствующие этому условию, называются *обратными ребрами* (back edges).

- Если v завершается при начале работы $\text{DFS}(u)$, мы немедленно получаем $v.\text{post} < u.\text{pre}$. Ребра, соответствующие этому условию, называются *секущими ребрами* (cross edge).
- Наконец, четвертый порядок $u.\text{post} < v.\text{pre}$ невозможен.

Перечисленные выше классы показаны на рис. 6.5. Напомню еще раз, что реальная классификацию ребер зависит от порядка, в котором DFS все рассматривает вершины, и от порядка, в котором DFS рассматривает ребра, исходящие из каждой вершины.

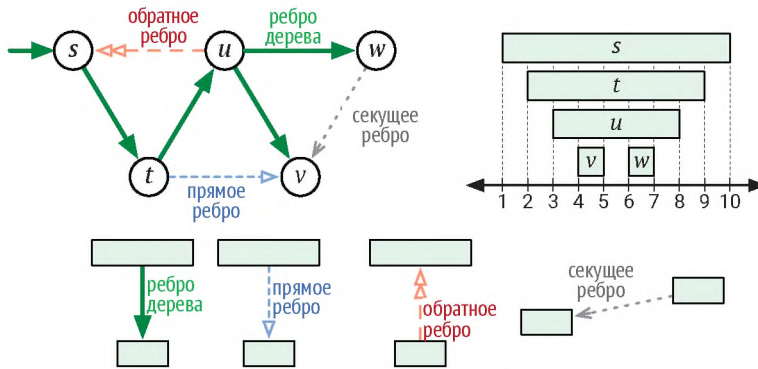


Рис. 6.5. Классификация ребер при поиске в глубину

Наконец, следующая весьма важная лемма характеризует предков и потомков в любом лесу поиска в глубину в соответствии с состояниями вершин во время обхода.

Лемма 6.1. Задан произвольный обход для поиска в глубину в любом направленном графе G . Приведенные ниже утверждения равнозначны для всех вершин u и v графа G :

- u – предок вершины v в лесу поиска в глубину;
- $u.\text{pre} \leq v.\text{pre} < v.\text{post} \leq u.\text{post}$;
- сразу после вызова $\text{DFS}(v)$ вершина u активна;
- непосредственно перед вызовом $\text{DFS}(u)$ существует путь от u к v , в котором каждая вершина (включая u и v) является новой.

Доказательство. Во-первых, предположим, что u – предок v в лесу поиска в глубину. Тогда по определению существует путь P из трех ребер от u к v . По индукции относительно длины пути имеем $u.\text{pre} \leq w.\text{pre} < w.\text{post} \leq u.\text{post}$ для каждой вершины w в пути P , следовательно, каждая вершина в пути P является новой перед вызовом $\text{DFS}(u)$. В частности, мы имеем $u.\text{pre} \leq v.\text{pre} < v.\text{post} \leq u.\text{post}$, из чего следует, что u – активная вершина при выполнении $\text{DFS}(v)$.

Поскольку указатели на родителей соответствуют рекурсивным вызовам, из $u.\text{pre} \leq v.\text{pre} < v.\text{post} \leq u.\text{post}$ следует, что u является предком v .

Предположим, что вершина u активна сразу после вызова $\text{DFS}(v)$. Тогда $u.\text{pre} \leq v.\text{pre} < v.\text{post} \leq u.\text{post}$, из чего следует, что существует путь из (нуля

или более) ребер дерева из вершины u через промежуточные узлы в стеке рекурсии (если такие существуют) к вершине v .

Наконец, предположим, что u не является предком v . Задан произвольный путь P из u в v , пусть x – первая вершина в пути P , которая не является потомком u , и пусть w – предшественник вершины x в пути P . Ребро $w \rightarrow x$ гарантирует, что $x.pre < w.post$ и $w.post < u.post$, потому что w – потомок u , так что $x.pre < u.post$. Из этого следует, что $x.pre < u.pre$, потому что иначе вершина x была бы потомком u . Поскольку активные интервалы являются корректно вложенными, существует только два возможных варианта:

- если $u.post < x.post$, то x – активная вершина при вызове $\text{DFS}(u)$;
- если $x.post < u.pre$, то x уже завершена при вызове $\text{DFS}(u)$.

Мы приходим к выводу, что каждый путь из u в v содержит вершину, которая не является новой при вызове $\text{DFS}(u)$. □

6.2. Обнаружение циклов

Направленный ациклический граф, или НАГ, – это направленный граф без направленных циклов. Любая вершина в НАГ, которая не имеет входящих ребер, называется *источником* (source), а любая вершина без исходящих ребер называется *приемником* (или получателем – sink). Изолированная вершина, которая вообще не имеет смежных ребер, является и источником, и приемником. Любой НАГ содержит как минимум один источник и один приемник, но может существовать и более одного источника и приемника. Например, в графе с n вершинами, но без ребер каждая вершина является источником и приемником.

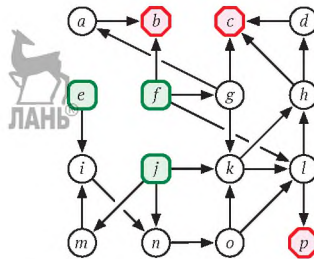


Рис. 6.6. Направленный ациклический граф.
Вершины e, f и j – источники, вершины b, c и p – приемники

Вспомним из разбора предыдущего примера, что если $u.post < v.post$ для любого ребра $u \rightarrow v$, то граф содержит направленный путь из v в u , следовательно, содержит направленный цикл, проходящий через ребро $u \rightarrow v$. Таким образом, мы можем определить, является ли заданный направленный граф G НАГ, за время $O(V + E)$, вычисляя обратный порядок обхода вершин, а затем проверяя каждую вершину простым перебором.

В качестве альтернативы вместо нумерации вершин мы можем явно сохранять состояние каждой вершины и немедленно возвращать `False`,

как только обнаружим ребро к активной вершине. Этот алгоритм также выполняется за время $O(V + E)$, см. рис. 6.7.

```

IsAcyclic(G):
  for all vertices  $v$ 
     $v.status \leftarrow \text{New}$ 
  for all vertices  $v$ 
    if  $v.status = \text{New}$ 
      if IsAcyclicDFS( $v$ ) = False
        return False
  return True

```

IsAcyclicDFS(v):

```

 $v.status \leftarrow \text{Active}$ 
for each edge  $v \rightarrow w$ 
  if  $w.status = \text{Active}$ 
    return False
  else if  $w.status = \text{New}$ 
    if IsAcyclicDFS( $w$ ) = False
      return False
 $v.status \leftarrow \text{Finished}$ 
return True

```

Рис. 6.7. Алгоритм с линейным временем выполнения, определяющий, является ли граф ациклическим

6.3. Топологическая сортировка

Топологическое упорядочение направленного графа G – это общий порядок $<$ («предшествует») для вершин, такой что $u < v$ для каждого ребра $u \rightarrow v$. Выражаясь менее формально, топологическое упорядочение расставляет вершины вдоль горизонтальной линии так, чтобы все ребра имели направление слева направо. Топологическое упорядочение абсолютно невозможно, если в графе G имеется направленный цикл – в этом случае крайняя правая вершина этого цикла будет иметь направленное влево ребро!

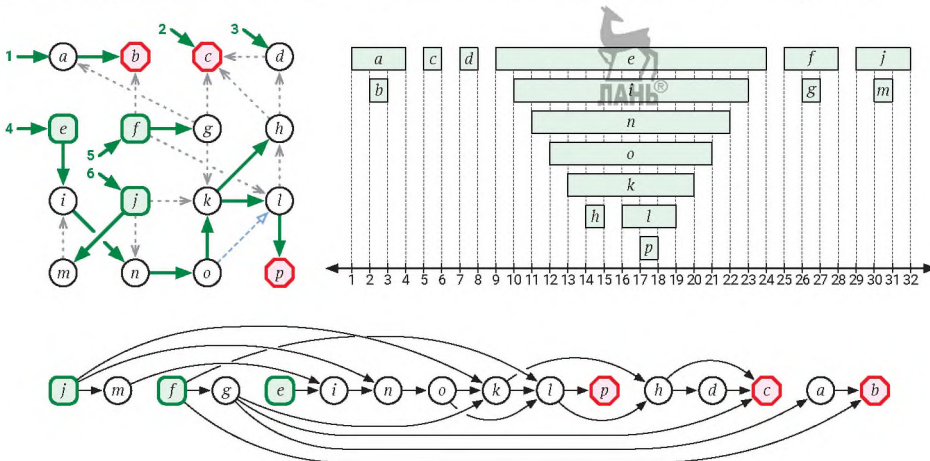


Рис. 6.8. Обращенный обход в обратном порядке для НАГ с рис. 6.6

С другой стороны, рассмотрим произвольный обход в обратном порядке в произвольном направленном графе G , см. рис. 6.8. Из нашего предыдущего анализа следует, что $u.post < v.post$ для любого ребра $u \rightarrow v$, тогда G содержит направленный путь из v в u , следовательно, содержит направлен-

ный цикл, проходящий через $u \rightarrow v$. Равнозначно: если G – ациклический граф, то $u.post > v.post$ для любого каждого ребра $u \rightarrow v$. Из этого следует, что каждый направленный ациклический граф G топологически упорядочен. В частности, обращение любого обхода в обратном порядке графа G является топологическим упорядочением G .

Если требуется топологическое упорядочение в отдельной структуре данных, то мы можем просто записать вершины в массив в обратном порядке обхода за время $O(V + E)$, как показано на рис. 6.9.

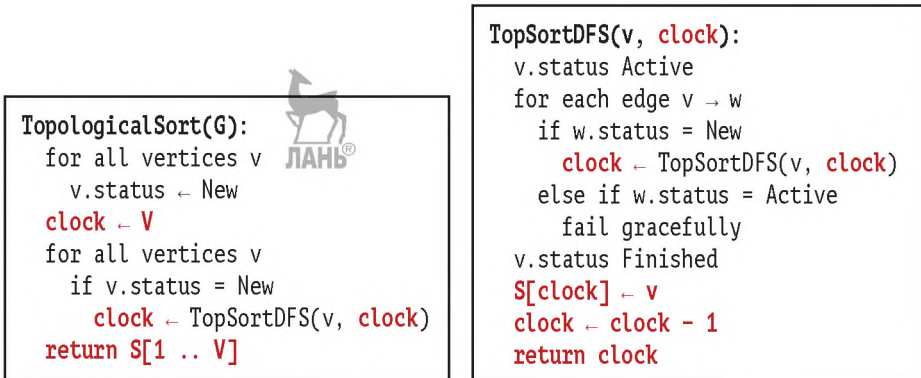
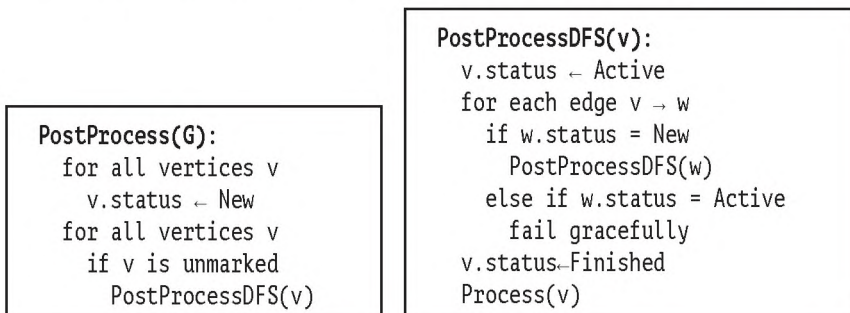


Рис. 6.9. Явная топологическая сортировка

Неявная топологическая сортировка

Но записывать топологический порядок в отдельную структуру данных обычно излишне. В большинстве приложений топологической сортировки упорядоченный список вершин не является настоящей целью, вместо этого необходимо выполнить некоторое вполне определенное вычисление в каждой вершине графа – либо в топологическом порядке, либо в обратном топологическом порядке. Для таких приложений вообще нет необходимости в записи топологического порядка.

Если необходимо обработать направленный ациклический граф в обратном топологическом порядке, то достаточно обработать каждую вершину в конце рекурсивного поиска в глубину. В сущности, топологический порядок – это то же самое, что и реверсированный обход в обратном порядке.



Если мы уже знаем, что исходный граф является ациклическим, то можем еще больше упростить этот алгоритм, просто пометив вершины вместо записи их поискового статуса.

PostProcessDag(G):

```
for all vertices v
  unmark v
for all vertices v
  if v is unmarked
    PostProcessDagDFS(s)
```

PostProcessDagDFS(v):

```
mark v
for each edge  $v \rightarrow w$ 
  if w is unmarked
    PostProcessDagDFS(w)
Process(v)
```

Это просто стандартный алгоритм поиска в глубину с переименованием подпрограммы `PostVisit` в `Process`.

Поскольку это весьма обобщенная операция в направленных ациклических графах, иногда я выражаю обработку в обратном порядке для НАГ идиоматически следующим образом:

PostProcessDag(G):

```
for all vertices v in postorder
  Process(v)
```

Например, наш предыдущий алгоритм явной топологической сортировки можно записать так:

TopologicalSort(G):

```
clock ← V
for all vertices v in postorder
  S[clock] ← v
  clock ← clock - 1
return S[1..V]
```

Для обработки НАГ в прямом топологическом порядке мы можем записать топологическое упорядочение вершин в массив, а затем выполнить простой цикл `for`. В качестве альтернативы можно применить поиск в глубину к обратному (реверсированному) представлению графа G , обозначенного $rev(G)$ и полученного заменой каждого ребра $v \rightarrow w$ на обратное ему ребро $w \rightarrow v$. Обращение направленного цикла дает другой направленный цикл с противоположной ориентацией, поэтому обратный НАГ – это другой НАГ. Каждый источник в G является приемником в $rev(G)$ и наоборот. По индукции из этого следует, что каждая топологическая упорядоченность $rev(G)$ является обращением топологической упорядоченности G ³. Обращение любого направленного графа (представленного в виде стандартного списка смежных вершин) может быть вычислено за время $O(V + E)$. Детали этой конструкции вынесены в простое упражнение.

³ Обратный порядок обхода обращенного графа G необязательно является обращением обратного порядка обхода G , даже если они оба представляют собой топологические упорядоченности G .

6.4. Мемоизация и динамическое программирование

Пожалуй, наш алгоритм топологической сортировки является эталоном для широкого класса алгоритмов динамического программирования. Напомним, что граф зависимостей рекуррентного выражения содержит вершину для каждой рекурсивной подзадачи и ребро от одной подзадачи к другой, если для вычисления первой подзадачи требуется рекурсивное вычисление второй. Граф зависимостей непременно должен быть ациклическим, иначе простейший рекурсивный алгоритм никогда не завершится.

Вычисление любого рекуррентного выражения с мемоизацией – это в точности то же самое, что выполнение поиска в глубину в графе зависимостей. По существу, вершина графа зависимостей «маркируется», если значение соответствующей подзадачи уже вычислено. Подпрограммы типа «черный ящик» PreVisit и PostVisit являются посредниками для такого вычисления реального значения (см. рис. 6.10).

Memoize(x):

```
if value[x] is undefined
  initialize value[x]

for all subproblems y of x
  Memoize( y )
  update value[x] based on value[y]
finalize value[x]
```

DFS(v):

```
if v is unmarked
  mark v
  PreVisit(x)
  for all edges v → w
    DFS(w)

  PostVisit(x)
```

Рис. 6.10. Мемоизированная рекурсия – это поиск в глубину.

Поиск в глубину – это мемоизированная рекурсия

Продолжая эту аналогию, приходим к выводу, что вычисление рекуррентного выражения с использованием динамического программирования – это то же самое, что вычисление всех подзадач в графе зависимостей рекуррентного выражения в обратном топологическом порядке, – каждая подзадача рассматривается после подзадач, от которых она зависит. Таким образом, каждый алгоритм динамического программирования равнозначен обходу в обратном порядке графа зависимостей, представляющего его внутренний рекуррентный процесс.

DynamicProgramming(G):

```
for all subproblems x in postorder
  initialize value[x]
  for all subproblems y of x
    update value[x] based on value[y]
  finalize value[x]
```

Рис. 6.11. Динамическое программирование – это обход в обратном порядке

Тем не менее существуют некоторые небольшие различия между большинством алгоритмов динамического программирования и топологической сортировкой. Во-первых, в большинстве алгоритмов динамического программирования граф зависимостей является неявным – узлы и ребра явно не записываются в память, вместо этого они кодируются внутренним рекуррентным процессом. Но это различие действительно незначительное – пока мы можем пронумеровать рекурсивные подзадачи за постоянное время, есть возможность выполнить обход графа зависимостей в точности так, как если бы он был явно сохранен в списке смежности вершин.

Более существенное различие: большинство рекуррентных выражений динамического программирования имеет сильно структурированные графы зависимостей. Например, как мы обсуждали в главе 5, граф зависимостей для рекуррентного выражения расстояния редактирования – это регулярная решетка с диагоналями, а граф зависимостей для оптимальных деревьев двоичного поиска – это верхняя треугольная решетка со всеми возможными правыми и верхними ребрами. Эта регулярная структура позволяет жестко закодировать подходящий порядок вычислений непосредственно в алгоритм обычно как набор вложенных циклов, поэтому здесь нет необходимости в топологической сортировке графа зависимостей во время выполнения. Ранее мы называли обратный топологический порядок порядком оценки.

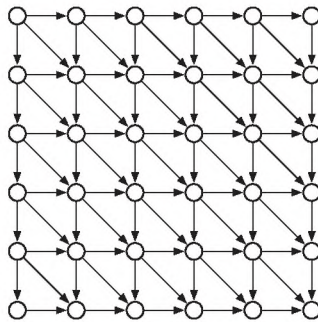


Рис. 6.12. НАГ зависимостей для рекуррентного выражения расстояния редактирования

Динамическое программирование в НАГ

С другой стороны, мы можем использовать поиск в глубину для создания алгоритмов динамического программирования для задач с менее структурированными графами зависимостей. Например, рассмотрим задачу о максимальном пути, в которой требуется найти путь с максимальным суммарным весом из узла s в другой узел t в направленном графе G с весовыми ребрами. В обобщенных направленных графах задача о максимальном пути является NP-трудной (простым приведением от задачи коммивояжера, см. главу 12), но она становится простой, если исходный граф G ацикли-

ческий, тогда можно вычислить максимальный путь в G за линейное время, как показано ниже.

Зададим целевую вершину t , и для любого узла v пусть $LLP(v)$ обозначает длину максимального пути в G из v в t . Если G – НАГ, то функции $LLP(v)$ соответствует рекуррентное выражение:

$$LLP(v) = \begin{cases} 0 & , \text{ если } v = t, \\ \max\{\ell(v \rightarrow w) + LLP(w) \mid v \rightarrow w \in E\} & \text{ иначе,} \end{cases}$$

где $\ell(v \rightarrow w)$ обозначает заданный вес («длину») ребра $v \rightarrow w$, а $\max \emptyset = -\infty$. В частности, если вершина v – приемник, но не равна t , то $LLP(v) = -\infty$.

Граф зависимостей для этого рекуррентного выражения – это сам исходный граф G : подзадача $LLP(v)$ зависит от подзадачи $LLP(w)$, если и только если $v \rightarrow w$ является ребром в графе G . Следовательно, мы можем вычислить эту рекурсивную функцию за время $O(V + E)$, выполняя поиск в глубину в G , начиная с вершины s . Алгоритм мемоизирует каждую длину $LLP(v)$ в дополнительном поле соответствующего узла v .

```
LongestPath(v, t):
  if v = t
    return 0
  if v.LLP is undefined
    v.LLP ← -∞
  for each edge v → w
    v.LLP ← max{v.LLP, ℓ(v → w) + LongestPath(w, t)}
  return v.LLP
```

Теоретически мы можем преобразовать этот мемоизованный рекурсивный алгоритм в алгоритм динамического программирования с помощью топологической сортировки:

```
LongestPath(s, t):
  for each node v in postorder
    if v = t
      v.LLP ← 0
    else
      v.LLP ← -∞
      for each edge v → w
        v.LLP ← max{v.LLP, ℓ(v → w) + w.LLP}
  return s.LLP
```

Вероятно, эти два алгоритма одинаковы – рекурсия в первом алгоритме и цикл `for` во втором алгоритме представляют один и тот же поиск в глубину. Выбор любой из двух формулировок – это исключительно вопрос удобства.

Почти каждая задача динамического программирования, для которой требуется оптимальная последовательность принятия решений, может быть переформулирована как поиск оптимального пути в некотором соответствующем НАГ. Например, задачи сегментации текста, суммы подмножеств, поиска максимальной возрастающей подпоследовательности и расстояния редактирования, рассмотренные в главах 2 и 3, можно переформулировать как поиск максимального или минимального пути в НАГ, предположительно со взвешенными вершинами или ребрами. В любом случае обсуждаемый здесь НАГ является графом зависимостей внутреннего рекуррентного выражения. С другой стороны, имеющие «древовидную форму» задачи динамического программирования, такие как поиск оптимальных деревьев двоичного поиска или максимальных независимых множеств в деревьях, невозможно переформулировать как поиск оптимального пути в НАГ.

6.5. Сильная связность

Вернемся к точному определению связности в направленных графах. Напомним, что из вершины u можно достичь другой вершины v в направленном графе G , если G содержит направленный путь из u в v , и что $reach(u)$ обозначает множество всех вершин, достижимых из u . Две вершины u и v являются сильно связными, если из u можно достичь v , и из v – u . Направленный граф является *сильно связным*, если и только если каждая пара вершин является сильно связной.

Это громоздкое определение полагает, что сильная связность – это равнозначное отношение в множестве вершин любого направленного графа по аналогии со связностью в ненаправленных графах. Классы равнозначности этого отношения называются *сильно связными компонентами* или проще – *сильными компонентами графа G* . Равнозначно сильная компонента G – это максимальный сильно связный подграф G . Направленный граф G является сильно связным, если и только если G содержит ровно одну сильную компоненту. Другая крайность: G является НАГ, если и только если каждая сильная компонента G состоит из одной вершины.

Граф *сильных компонент* $scc(G)$ – это другой направленный граф, полученный из G объединением всех сильных компонент в одну вершину и сворачиванием параллельных ребер. (Граф сильных компонент также иногда называют метаграфом или уплотнением графа G .) Нетрудно доказать (подсказка, подсказка), что $scc(G)$ всегда является НАГом. Таким образом, по крайней мере теоретически возможно топологическое упорядочение сильных компонент G , т. е. вершины можно упорядочить так, чтобы каждое обратное ребро соединяло два ребра в той же сильной компоненте.

Вычислить сильную компоненту одной вершины v очень просто за время $O(V + E)$. Сначала мы вычисляем $reach(v)$ методом поиска в любом направлении. Затем вычисляется $reach^{-1}(v) = \{u \mid v \in reach(u)\}$ методом поиска в обратном графе G . Наконец, сильная компонента вершины v – это пере-

сечение $reach(v) \in reach^{-1}(v)$. В частности, мы можем определить, является ли весь граф в целом сильно связным, за время $O(V + E)$.

Также можно вычислить все сильные компоненты в направленном графе, объединив приведенный выше алгоритм с нашей стандартной функцией-оберткой. Но итоговый алгоритм выполняется за время $O(VE)$. Существует не более V сильных компонент, и для обнаружения каждой из них требуется время $O(E)$, даже если граф представляет собой НАГ. И конечно же, мы можем добиться большего! В конце концов, нам нужно только время $O(V + E)$, чтобы решить, является ли каждая сильная компонента единственной вершиной.

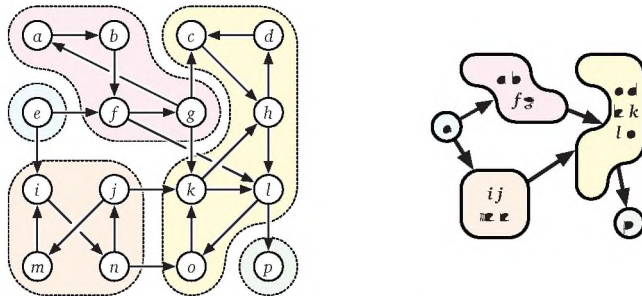


Рис. 6.13. Сильные компоненты графа G и граф сильных компонент $scc(G)$

6.6. Сильные компоненты за линейное время

На самом деле существует несколько алгоритмов вычисления сильных компонент за время $O(V + E)$, и все они опираются на следующее наблюдение.

Лемма 6.2. Пусть определен обход в глубину в любом направленном графе G . Каждая сильная компонента C в графе G содержит ровно один узел, который не имеет родителя в C . (Этот узел либо имеет родителя в другой сильной компоненте, либо вообще не имеет родителей.)

Доказательство. Пусть C – произвольная сильная компонента в G . Рассмотрим любой путь из одной вершины $v \in C$ в другую вершину $w \in C$. Каждая вершина в этом пути может достигать w , а следовательно, и каждой вершины в C . По симметрии каждый узел в этом пути является достижимым из вершины v , а следовательно, достижим из каждой вершины в C . Отсюда выводим, что каждая вершина в этом пути также находится в компоненте C .

Пусть v – вершина в C с самым ранним начальным временем. Если v имеет родителя, то $parent(v)$ начинается раньше v , следовательно, не может находиться в C .

Теперь пусть w – другая вершина в C . Непосредственно перед вызовом $DFS(v)$ каждая вершина в C является новой, поэтому существует путь из новых вершин из v в w . Тогда из леммы 6.1 следует, что w – потомок v в лесу поиска в глубину. Каждая вершина в пути из ребер дерева от v к w лежит в C , в частности, $parent(w) \in C$. $\square$

Лемма 6.2 подразумевает, что каждая сильная компонента направленного графа G определяет связное поддерево любого леса поиска в глубину графа G . В частности, для любой сильной компоненты C вершина в C с самым ранним временем начала – это наименьший общий предок всех вершин в C . Мы называем эту вершину корнем компоненты C .

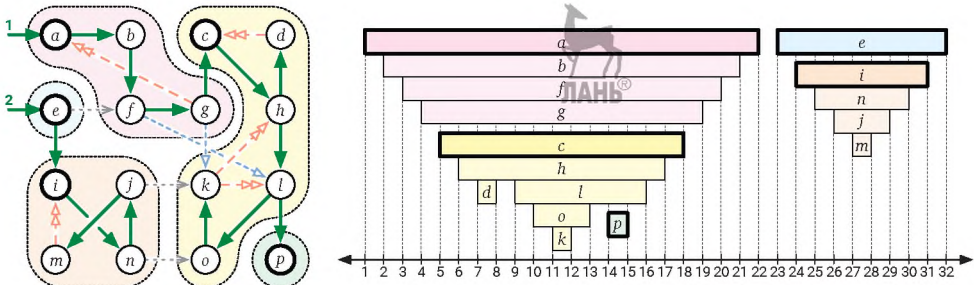


Рис. 6.14. Сильные компоненты непрерывны в лесу поиска в глубину

Я предлагаю два алгоритма, которые следуют одинаковой интуитивной схеме. Пусть C – любая сильная компонента графа G , т. е. приемник в $scc(G)$. Мы называем C компонентой-приемником. Равнозначно C – компонента-приемник, если область досягаемости любой вершины в C – это в точности C . Мы можем найти все сильные компоненты в G путем многократного нахождения вершины v в некоторой компоненте-приемнике (каким-либо способом), нахождения вершин, достижимых из v , и удаления этой компоненты-приемника из исходного графа до тех пор, пока не останется вершин. Это пока еще не совсем алгоритм, потому что не ясно, как найти вершину в компоненте-приемнике.

```

StrongComponents( $G$ ):
    count  $\leftarrow 0$ 
    while  $G$  is non-empty
         $C \leftarrow \emptyset$ 
        count  $\leftarrow$  count + 1
         $v \leftarrow$  any vertex in a sink component of  $G$  <<Волшебство!>>
        for all vertices  $w$  in reach( $v$ )
             $w$ .label  $\leftarrow$  count
            add  $w$  to  $C$ 
        remove  $C$  and its incoming edges from  $G$ 
    
```

Рис. 6.15. Примерный алгоритм вычисления сильных компонент

Алгоритм Косараджу–Шарира

На первый взгляд быстро найти вершину в компоненте-приемнике кажется достаточно трудным. Однако на самом деле найти вершину в компоненте-источнике – сильной компоненте графа G , соответствующей источнику в $scc(G)$, – довольно легко, используя поиск в глубину.

Лемма 6.3. Последняя вершина в любом обратном пути обхода графа G лежит в компоненте-источнике G .

Доказательство. Пусть задан путь обхода в глубину графа G и пусть v – последняя вершина в итоговом обратном пути обхода. Тогда $\text{DFS}(v)$ обязательно должен быть последним прямым вызовом подпрограммы DFS , выполненным алгоритмом-оберткой DFSAll . Кроме того, v – корень одного из деревьев в лесу поиска в глубину, поэтому любой узел x при $x.\text{post} > v.\text{pre}$ является потомком v . Наконец, v – корень сильной компоненты S .

Чисто теоретически предположим, что существует ребро $x \rightarrow y$, такое что $x \notin S$ и $y \in S$. Тогда из x можно достичь y , а из y можно достичь v , поэтому из x можно достичь v . Так как v – корень S , вершина y является потомком v , следовательно, $v.\text{pre} < x.\text{post}$. Из этого следует, что x – потомок v . Но тогда из v можно достичь x (проходя через три ребра), а это противоречит нашему предположению о том, что $x \notin S$. $\square$

Легко проверить (намекаяю), что $\text{rev}(\text{scc}(G)) = \text{scc}(\text{rev}(G))$ для любого направленного графа G . Таким образом, последняя вершина в обратном порядке обхода $\text{rev}(G)$ лежит в компоненте-приемнике исходного графа G . Следовательно, если мы обходим граф во второй раз, где функция-обертка следует реверсивному обратному пути обхода $\text{rev}(G)$, то каждый вызов DFS посещает ровно одну сильную компоненту графа G .⁴

```
KosarajuSharir(G):
  S ← new empty stack
  for all vertices v
    unmark v
    v.root ← None
  <<Этап 1: Передача обхода в обратном порядке в rev(G)>>
  for all vertices v
    if v is unmarked
      PushPostRevDFS(v, S)
  <<Этап 2: DFS снова в стеке порядка обхода>>
  while S is non-empty
    v ← Pop(S)
    if v.root = None
      LabelOneDFS(v, v)
```

```
PushPostRevDFS(v, S):
  mark v
  for each edge u → v <<Смена направления!>>
    if u is unmarked
      PushPostRevDFS(u, S)
  Push(v, S)
```

```
LabelOneDFS(v, r):
  v.root ← r
  for each edge v → w
    if w.root = None
      LabelOneDFS(w, r)
```

Рис. 6.16. Алгоритм Косараджу–Шарира вычисления сильных компонент

⁴ И снова: реверсивный обратный путь обхода $\text{rev}(G)$ – это не то же самое, что обратный путь обхода графа G .

Соединив все вместе, получим алгоритм, показанный на рис. 6.16, который подсчитывает и помечает сильные компоненты любого направленного графа за время $O(V + E)$. Этот алгоритм был разработан (но никогда не публиковался) Рао Косараджу (Rao Kosaraju) в 1978 г., а позже был независимо повторно разработан Михой Шариром (Micha Sharir) в 1981 г.⁵ Алгоритм Косараджу–Шарира состоит из двух этапов. На первом этапе выполняется поиск в глубину в $rev(G)$ с записью каждой вершины в стек при ее завершении. На втором мы выполняем обход в любом направлении исходного графа G , рассматривая вершины в том порядке, в котором они появляются в стеке. Алгоритм помечает каждую вершину с корнем ее сильной компоненты (с учетом второго этапа обхода в глубину).

На рис. 6.17 показан алгоритм Косараджу–Шарира, выполняющийся для нашего примера графа. При минимальных изменениях в этом алгоритме мы можем также вычислить граф сильных компонент $scc(G)$ за время $O(V + E)$.

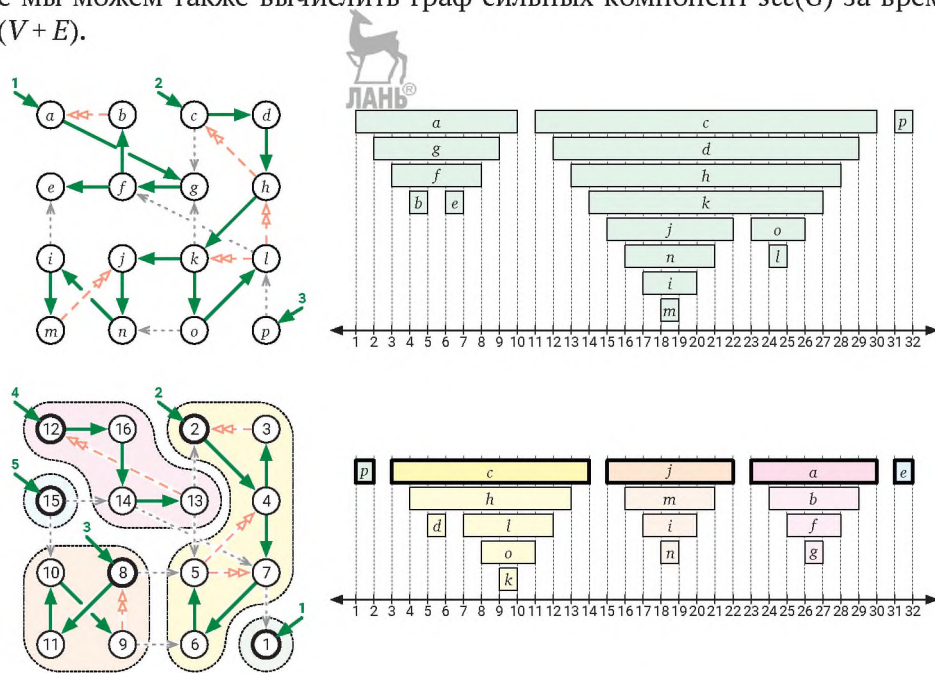


Рис. 6.17. Алгоритм Косараджу–Шарира в действии.

Вверху: обход в глубину обращенного графа.

Внизу: обход в глубину исходного графа с посещением корневых вершин в реверсированном обратном порядке, полученном из первого обхода

♥ Алгоритм Тарьяна

Более ранний, но значительно более хитроумный алгоритм, выполняемый за линейное время, предназначенный для вычисления сильных ком-

⁵ Ходят слухи, что этот алгоритм появился в русскоязычной литературе даже раньше разработки Косараджу, но я пока еще не нашел надежного источника, подтверждающего эти слухи.

понент, был опубликован Робертом Тарьяном (Robert Tarjan) в 1972 г.⁶ Интуитивно говоря, алгоритм Тарьяна идентифицирует компоненту-источник графа G , «удаляет» ее, а затем «рекурсивно» находит оставшиеся сильные компоненты, но полностью вычисление происходит во время одного поиска в глубину.

Пусть определен произвольный обход в глубину некоторого направленного графа G . Для каждой вершины v пусть $low(v)$ обозначает наименьшее начальное время из всех вершин, достижимых из v по пути из ребер дерева, за которым следует не более одного ребра, не относящегося к дереву. Очевидно, что $low(v) \leq v.pre$, потому что v может достичь саму себя через ноль ребер дерева, за которыми следует ноль ребер, не относящихся к дереву. Тарьян заметил, что компоненты-приемники могут быть охарактеризованы с помощью этой функции low .

Лемма 6.4. Вершина v является корнем компоненты-приемника графа G , если и только если $low(v) = v.pre$ и $low(w) < w.pre$ для каждого подходящего потомка w вершины v .

Доказательство. Во-первых, пусть v – вершина, такая что $low(v) = v.pre$. Тогда не существует ребра $w \rightarrow x$, где w – потомок v и $x.pre < v.pre$. С другой стороны, v не может достичь любой вершины u , такой что $u.pre > v.post$. Из этого следует, что v может достигать только своих потомков, следовательно, любой потомок v может достигать только потомков v . В частности, v не может достичь своего родителя (если он имеется), поэтому v является корнем своей сильной компоненты.

Теперь дополнительно предположим, что $low(w) < w.pre$ для каждого потомка w вершины v . Тогда каждый потомок w может достигать другой вершины x (которая обязательно должна быть другим потомком вершины v), такой что $x.pre < w.pre$. Таким образом, по индукции каждый потомок v может достигать v . Из этого следует, что потомки v составляют сильную компоненту C , корнем которой является v . Кроме того, C обязательно должна быть компонентой-приемником, потому что v не может достигать любой вершины за пределами C .

С другой стороны, предположим, что v – корень компоненты-приемника C . Тогда v может достигать другой вершины w , если и только если $w \in C$. Но v может достигать всех своих потомков, и каждая вершина в C является потомком v , поэтому потомки вершины v составляют C . Если $low(w) = w.pre$ для любого другого узла $w \in C$, то w должен быть другим корнем C , что невозможно. $\square$

Вычисление $low(v)$ для каждой вершины v методом поиска в глубину выполняется просто, см. рис. 6.18.

⁶ По легенде, Косараджу, вероятно, придумал свой алгоритм во время лекции по алгоритмам. На лекции был представлен алгоритм Тарьяна, но Косараджу забыл свой конспект, поэтому ему пришлось придумывать что-то другое на лету. Единственное, что меня удивляет в этом рассказе, – никто не упоминает Шарира или Тарьяна.

FindLow(G):

```

clock ← 0
for all vertices v
  unmark v
for all vertices v
  if v is unmarked
    FindLowDFS(v)

```

FindLowDFS(v):

```

mark v
clock ← clock + 1
v.pre ← clock
v.low ← v.pre
for each edge v → w
  if w is unmarked
    FindLowDFS(w)
    v.low ← min(v.low, w.low)
  else
    v.low ← min(v.low, w.pre)

```

Рис. 6.18. Вычисление $low(v)$ для каждой вершины v

Лемма 6.4 предполагает, что после запуска FindLow мы можем идентифицировать корень каждой компоненты-приемника за время $O(V + E)$ (методом глобального поиска в любом направлении), затем пометить и удалить эти компоненты-приемники за дополнительное время $O(V + E)$ (вызывая процедуру поиска в любом направлении в каждом корне), после чего применить рекурсию. К сожалению, итоговый алгоритм может потребовать V итераций, каждая из которых удаляет только одну вершину, что упрощенно дает нам общее время выполнения $O(VE)$.

Для ускорения этой стратегии алгоритм Тарьяна поддерживает вспомогательный стек вершин (отдельно от стека рекурсии). Когда мы начинаем работать с новой вершиной v , то записываем ее в стек. При завершении работы с вершиной v мы сравниваем $v.low$ с $v.pre$. Когда в первый раз мы обнаруживаем, что $v.low = v.pre$, то нам известно три факта:

- вершина v является корнем компоненты-приемника C ;
- все вершины в C последовательно появляются на вершине вспомогательного стека;
- самой глубокой вершиной из C во вспомогательном стеке является v .

В этот момент мы можем идентифицировать вершины в C , извлекая их из вспомогательного стека по одной и останавливаясь, когда извлекается вершина v .

Можно было бы удалить вершины в C и рекурсивно вычислять сильные компоненты оставшегося графа, но это было бы неэффективно, потому что пришлось бы повторять дословно все вычисления, выполненные перед посещением v . Вместо этого мы помечаем каждую вершину в C , идентифицируя v как корень ее сильной компоненты, а затем исключая помеченные вершины для оставшейся части поиска в глубину. Формально эта модификация изменяет определение $low(v)$ для наименьшего начального времени по всем вершинам в той же сильной компоненте, где находится v , так что v достижима по пути из трех ребер дерева, за которыми следует не более одного ребра, не относящегося к дереву. Но для доказательства кор-

ректности проще заметить, что игнорирование помеченных вершин приводит к точно такому же поведению алгоритма, что и при действительном удалении вершин.

Итоговый алгоритм Тарьяна показан на рис. 6.19, где все необходимые модификации, взятые из FindLow (см. рис. 6.18), выделены полужирным шрифтом красного цвета. Время выполнения этого алгоритма можно разделить на две части. Каждая вершина передается в S и извлекается из S один раз, поэтому общее время, затраченное на поддержку вспомогательного стека (псевдокод, выделенный красным цветом), равно $O(V)$. Если мы игнорируем поддержку вспомогательного стека, то оставшая часть алгоритма – это просто стандартный поиск в глубину. Отсюда заключаем, что алгоритм выполняется за время $O(V + E)$.

```
Tarjan(G):
  clock ← 0
  S ← new empty stack
  for all vertices v
    unmark v
    v.root ← None
  for all vertices v
    if v is unmarked
      TarjanDFS(v)
```

```
TarjanDFS(v):
  mark v
  clock ← clock + 1
  v.pre ← clock
  v.low ← v.pre
  Push(S, v)
  for each edge v → w
    if w is unmarked
      TarjanDFS(w)
      v.low ← min(v.low, w.low)
    else if w.root = None
      v.low ← min(v.low, w.pre)
  if v.low = v.pre
    repeat
      w ← Pop(S)
      w.root ← v
    until w = v
```

Рис. 6.19. Алгоритм Тарьяна для вычисления сильных компонент

Упражнения

Поиск в глубину, топологическая сортировка и сильные компоненты

0. (a) Описать алгоритм вычисления обращения $rev(G)$ направленного графа за время $O(V + E)$.
- (b) Доказать, что для каждого направленного графа G граф сильных компонент $scc(G)$ является ациклическим.
- (c) Доказать, что $scc(rev(G)) = rev(scc(G))$ для каждого направленного графа G .

- (d) Пусть определен произвольный направленный граф G . Для любой вершины v графа G пусть $S(v)$ обозначает сильную компоненту G , которая содержит v . Для всех вершин u и v графа G доказать, что u может достичь v в G , если и только если $S(u)$ может достичь $S(v)$ в $scc(G)$.
1. Направленный граф G является полусвязным (semi-connected), если для каждой пары вершин u и v либо u достижима из v , либо v достижима из u (или обе взаимно достижимы).
 - (a) Привести пример направленного ациклического графа с неповторяющимся источником, который не является полусвязным.
 - (b) Описать и проанализировать алгоритм, определяющий, является ли заданный направленный ациклический граф полусвязным.
 - (c) Описать и проанализировать алгоритм, определяющий, является ли произвольный направленный граф полусвязным.
 2. Полицейское управление города Шам-Пубанана (Sham-Poobanana) приняло решение об одностороннем движении на каждой городской улице. Несмотря на многочисленные жалобы от сбитых с толку автолюбителей, мэр заявила, что всегда возможно без нарушения правил проехать с любого перекрестка Шам-Пубанана на любой другой перекресток.
 - (a) Горожанам необходимо либо подтвердить, либо опровергнуть заявление мэра. Формализовать эту задачу в терминах графов, затем описать и проанализировать алгоритм ее решения.
 - (b) После выполнения алгоритма из части (a) мэр с неохотой призналась, что она солгала была дезинформирована. Перекресток x называется правильным, если для любого перекрестка u , до которого можно без нарушения правил доехать с перекрестка x , существует возможность проезда без нарушения правил от u обратно к x . Теперь мэр заявляет, что более 95 % перекрестков в Шам-Пубанана являются правильными. Описать и проанализировать эффективный алгоритм подтверждения или опровержения заявления мэра.

Для получения максимальной оценки оба алгоритма должны выполняться за линейное время.

3. Предположим, что задан направленный ациклический граф G с неповторяющимся источником s и приемником t . Вершина $v \notin \{s, t\}$ называется (s, t) -разделяющей вершиной, если каждый путь из s в t проходит через v , или – равнозначно – если удаление v приводит к недостижимости t из s . Описать и проанализировать алгоритм поиска каждой (s, t) -разделяющей вершины в графе G .

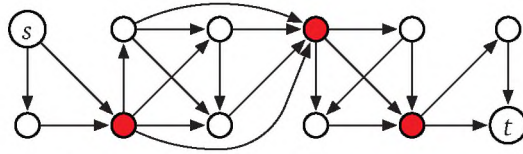


Рис. 6.20. Направленный ациклический граф с тремя (s, t) -разделяющими вершинами

4. Вершина v в связном ненаправленном графе G называется разделяющей вершиной (cut vertex), если подграф $G - v$ (полученный удалением v из G) является несвязным.

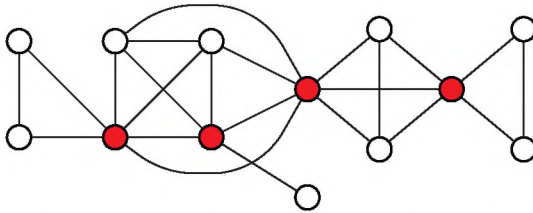


Рис. 6.21. Ненаправленный граф с четырьмя разделяющими вершинами

- (a) Описать алгоритм с линейным временем выполнения, определяющий при заданном ненаправленном графе G и вершине v , является ли v разделяющей вершиной в G . Чему равно время выполнения поиска всех разделяющих вершин при проверке этим алгоритмом каждой вершины?
- (b) Пусть T – остовное дерево поиска в глубину ненаправленного графа G .
 - i. Доказать, что корень T является разделяющей вершиной G , если и только если в T содержится более одного прямого потомка.
 - ii. Доказать, что некорневая вершина v является разделяющей вершиной G , если и только если как минимум один потомок (в T) каждого прямого потомка v (в T) является соседом (в G) некоторого корректного предка v (в T).

[Подсказка: эти утверждения становятся неверными, если T не является остовным деревом поиска в глубину и/или G не является направленным графом.]

- (c) Описать алгоритм, идентифицирующий каждую разделяющую вершину в заданном ненаправленном графе за время $O(V + E)$.
5. Ребро e в связном ненаправленном графе G называется мостом (bridge) (или разделяющим ребром – cut edge), если подграф $G - e$ (полученный удалением e из G) является несвязным.

- (а) При заданном графе G и ребре e описать алгоритм, выполняемый за линейное время, который определяет, является ли e мостом или нет. Чему равно время выполнения поиска всех мостов, если ваш алгоритм проверяет каждое ребро?
 - (б) Пусть T – произвольное остовное дерево графа G . Доказать, что каждый мост в G также является ребром в T . Это утверждение предполагает, что G содержит не более $V - 1$ мостов. Как эта информация может улучшить алгоритм из части (а) для поиска всех мостов?
 - (с) Теперь предположим, что корень T находится в произвольной вершине r . Для любой вершины v пусть T_v обозначает поддерево T с корнем в v . Например, $T_r = T$. Пусть uv – произвольное ребро в T , где u – родитель v . Доказать, что uv является мостом в G , если и только если uv – единственное ребро в G с ровно одной конечной точкой в T_v .
 - (д) Описать алгоритм с линейным временем выполнения, который идентифицирует каждый мост в графе G . [Подсказка: пусть T – остовное дерево поиска в глубину графа G .]
6. Транзитивное замыкание (transitive closure) G^T направленного графа G – это направленный граф с теми же вершинами, что и в G , который содержит любое ребро $u \rightarrow v$, если и только если существует направленный путь из u в v в графе G . Транзитивное сокращение (transitive reduction) графа G – это граф с минимальным возможным числом ребер, транзитивным замыканием которых является G^T . Тот же граф может иметь несколько транзитивных сокращений.
- (а) Описать эффективный алгоритм, вычисляющий транзитивное замыкание заданного направленного графа.
 - (б) Доказать, что направленный граф G содержит неповторяющееся транзитивное сокращение, если и только если G – ациклический граф.
 - (с) Описать эффективный алгоритм, вычисляющий транзитивное сокращение заданного направленного графа.
7. Один из старейших алгоритмов для исследования произвольных связных графов был предложен Гастоном Тарри (Gaston Tarry) в 1895 г. как систематизированная процедура для решения лабиринтов.⁷ Входными данными для алгоритма Тарри является ненаправленный граф G , но для упрощения представления мы формально разделяем каждое ненаправленное ребро uv на два направленных ребра $u \rightarrow v$ и $v \rightarrow u$. (В реальной реализации это разделение три-

⁷ Еще более старые алгоритмы обхода графов были описаны Шарлем Тремо (Charles Trémaux) в 1882 г., Кристианом Винаром (Christian Wiener) в 1873 г., Карлом Гиргольцером (Carl Hierholzer) в 1873 г. и (в неявной форме) Леонардом Эйлером (Leonhard Euler) в 1736 г. В частности, алгоритм Винара равнозначен алгоритму поиска в глубину в связных ненаправленных графах.

виально – алгоритм просто использует заданный список смежных вершин для G , как если бы G был направленным графом.)

Tarry(G):

```
unmark all vertices of  $G$ 
color all edges of  $G$  white
 $s \leftarrow$  any vertex in  $G$ 
RecTarry( $s$ )
```

RecTarry(v):

```
mark  $v$                                 <<"посещение  $v$ ">>
if there is a white arc  $v \rightarrow w$ 
  if  $w$  is unmarked
    color  $w \rightarrow v$  green
    color  $v \rightarrow w$  red } <<"обход  $v \rightarrow w$ ">>
    RecTarry( $w$ )      }
else if there is a green arc  $v \rightarrow w$ 
  color  $v \rightarrow w$  red } <<"обход  $v \rightarrow w$ ">>
  RecTarry( $w$ )      }
```

Мы неформально говорим, что алгоритм Тарри «посещает» вершину v каждый раз, когда он помечает v , и что алгоритм «обходит» ребро $v \rightarrow u$, когда он окрашивает это ребро в **красный** цвет и рекурсивно вызывает RecTarry(w). В отличие от нашего предыдущего алгоритма обхода графа алгоритм Тарри может помечать одни и те же вершины несколько раз.

- Описать, как нужно реализовать алгоритм Тарри, чтобы он выполнялся за время $O(V + E)$.
- Доказать, что ни одно из направленных ребер в графе G не проходит более одного раза.
- Когда алгоритм посещает вершину v в k -й раз, то сколько в точности ребер, входящих в v , окрашены в **красный** цвет и сколько в точности ребер, исходящих из v , окрашены в **красный** цвет? [Подсказка: рассмотрите начальную вершину s отдельно от всех прочих вершин.]
- Доказать, что каждая вершина v посещается не более $\deg(v)$ раз, исключая начальную вершину s , которая посещается не более $\deg(s) + 1$ раз. Это утверждение сразу же естественным образом подразумевает, что алгоритм Tarry(G) завершается.
- Доказать, что последняя вершина, посещенная алгоритмом Tarry(G), – это начальная вершина s .
- Для каждой вершины v , которую посещает Tarry(G), доказать, что все ребра, входящие в v и исходящие из нее, являются **красными**, когда Tarry(G) останавливается. [Подсказка: рассматривайте вершины в том порядке, в котором они маркируются в первый раз, начиная с s , и докажите это утверждение по индукции.]
- Доказать, что Tarry(G) посещает каждую вершину графа G . Это и предыдущее утверждение предполагает, что Tarry(G) проходит каждое ребро G ровно один раз.

8. Рассмотрим следующий вариант алгоритм Тарри обхода графа. Этот вариант проходит **зеленые** ребра без перекрашивания их в **красный** цвет и присваивает две числовые метки каждой вершине:

Tarry2(G):

```
unmark all vertices of G
color all edges of G white
s ← any vertex in G
RecTarry2(s, 1)
```

RecTarry2(v, clock):

```
if v is unmarked
    v.pre ← clock; clock ← clock + 1
    mark v
if there is a white arc v → w
    if w is unmarked
        color w → v green
        color v → w red
        RecTarry2(w, clock)
else if there is a green arc v → w
    v.post ← clock; clock ← clock + 1
    RecTarry2(w, clock)
```

Доказать или опровергнуть следующее утверждение: когда Tarry2(G) останавливается, зеленые ребра определяют остовное дерево, а метки $v.pre$ и $v.post$ определяют разметку обхода в прямом и обратном направлении, полностью согласованную с единственным поиском в глубину в графе G . Другими словами, доказать или опровергнуть тот факт, что Tarry2 формирует те же выходные данные, что и поиск в глубину, даже если посещает ребра в абсолютно другом порядке.

9. У вас есть набор n запертых ящиков и m золотых ключей. Каждый ключ отпирает не более одного ящика. Но каждый ящик можно отпереть одним ключом или несколькими ключами, или он открывается без ключа. Существует только два способа открыть каждый ящик, если он заперт: отпереть его аккуратно (для этого требуется наличие одного подходящего ключа в вашей руке) или разбить его в щепки молотком.

Ваш младший брат, который любит играть с блестящими вещами, иногда случайно запирает все ключи в ящиках. К счастью, ваша домашняя система обеспечения безопасности записывает все, поэтому вы точно знаете, какой именно ключ находится в каждом ящике (если ключ действительно в ящике). Вы должны достать все ключи из ящиков, потому что они золотые. Очевидно, что вам придется разбить как минимум один ящик.

- (а) Ваш младший брат раздобыл молоток и пристально смотрит на один из ящиков. Описать и проанализировать алгоритм, определяющий, возможно ли достать все ключи, не разбив ни один ящик за исключением того, который выбрал ваш брат.

- (b) Описать и проанализировать алгоритм, вычисляющий минимальное число ящиков, которые обязательно должны быть разбиты, чтобы достать все ключи.
10. Предположим, что вы преподаете курс алгоритмов. На экзамене в середине второго семестра вы дали своим студентам рисунок графа, затем предложили им определить дерево поиска в ширину и дерево поиска в глубину с корнем в конкретной вершине. К сожалению, как только вы начали выставлять оценки за экзамен, то заметили, что выданный студентам граф имеет несколько таких остовных деревьев – слишком много, чтобы их перечислять. Вместо этого необходимо найти способ, позволяющий объявить, является ли корректной сданная работа каждого студента.
- В каждой из описанных ниже задач предполагается, что задан связный граф G , начальная вершина s и остовное дерево T графа G .
- (a) Предположим, что граф G ненаправленный. Описать и проанализировать алгоритм, определяющий, является ли T остовным деревом поиска в глубину с корнем s .
- (b) Предположим, что граф G ненаправленный. Описать и проанализировать алгоритм, определяющий, является ли T остовным деревом поиска в ширину с корнем s . [Подсказка: недостаточно, чтобы T было невзвешенным деревом кратчайшего пути. Да, это правильная глава для решения этой задачи.]
- (c) Предположим, что граф G направленный. Описать и проанализировать алгоритм, определяющий, является ли T остовным деревом поиска в ширину с корнем s . [Подсказка: сначала решите часть (b).]
- (d) Предположим, что граф G направленный. Описать и проанализировать алгоритм, определяющий, является ли T остовным деревом поиска в глубину с корнем s .
11. Некоторые современные языки программирования, в том числе JavaScript, Python, Perl и Ruby, включают функциональную возможность, называемую параллельным присваиванием (parallel assignment), которая позволяет в одной строке кода выполнить несколько операций присваивания. Например, в коде на языке Python $x, y = 0, 1$ одновременно присваивается x значение 0, а y значение 1. Все значения справа от оператора присваивания определяются старыми значениями переменных. Таким образом, код на языке Python $a, b = b, a$ меняет местами значения a и b , а приведенный ниже код Python вычисляет n -е число Фибоначчи:

```
def fib(n):
    prev, curr = 1, 0
    while n > 0:
```

```

    prev, curr, n = curr, prev + curr, n - 1
    return curr

```

Предположим, что интерпретатор, который вы пишете, должен преобразовывать каждое параллельное присваивание в равнозначную последовательность отдельных присваиваний. Например, параллельное присваивание $a, b = 0, 1$ можно сериализовать в любом порядке – $a=0; b=1$ или $b=1; a=0$, – но параллельное присваивание $x, y = x+1, x+y$ можно сериализовать только как $y=x+y; x=x+1$. При сериализации может потребоваться одна или несколько дополнительных временных переменных, например сериализация $a, b = b, a$ требует одну временную переменную, а для сериализации $x, y = x+y, x-y$ требуются две временных переменных.

- (а) Описать алгоритм, определяющий, можно ли сериализовать заданное параллельное присваивание без дополнительных временных переменных.
- (б) Описать алгоритм, определяющий, можно ли сериализовать заданное параллельное присваивание ровно с одной дополнительной временной переменной.

Предположить, что заданное параллельное присваивание подразумевает использование только простых целых переменных (без косвенной адресации через указатели или массивы), переменные появляются слева не более одного раза, а выражения справа не имеют побочных эффектов. Не беспокойтесь о подробностях парсинга (синтаксического разбора) выражения присваивания, просто представьте соответствующее графовое представление (но опишите его).

Динамическое программирование

12. Предположим, что задан направленный ациклический граф G , узлы которого представляют задания, а ребра – ограничения по предшествованию, т. е. каждое ребро $u \rightarrow v$ означает, что задание u непременно должно быть завершено до начала выполнения задания v . Каждый узел v также имеет вес $T(v)$, обозначающий время, требуемое для выполнения задания v .

- (а) Описать алгоритм, определяющего кратчайший интервал времени, за который могут быть выполнены все задания в графе G .
- (б) Предположить, что первое задание начинается во время 0. Описать алгоритм, определяющий для каждой вершины v самое раннее время, в которое может начаться выполнение задания v .
- (с) Описать алгоритм, определяющий для каждой вершины v самое позднее время, в которое может начаться выполнение задания v без нарушения ограничений по предшествованию или без уве-

личения общего времени выполнения (вычисленного в части (а)), предполагая, что каждое задание за исключением v начинается в свое самое раннее время (вычисленное в части (б)).

13. Пусть G – направленный ациклический граф с единственным источником s и единственным приемником t .
 - (а) *Гамильтонов путь* в графе G – это направленный путь в G , который содержит каждую вершину G . Описать алгоритм, определяющий, содержит ли G гамильтонов путь.
 - (б) Предположить, что вершины G имеют веса. Описать эффективный алгоритм поиска пути из s в t с максимальным суммарным весом.
 - (с) Предположить, что также задано целое число ℓ . Описать эффективный алгоритм поиска пути с максимальным весом из s в t , который содержит не более ℓ ребер. (Предположить, что существует хотя бы один такой путь.)
 - (d) Предположить, что некоторые из вершин G помечены как важные, а кроме того, задано целое число k . Описать эффективный алгоритм поиска пути с максимальным весом из s в t , который посещает как минимум k важных вершин. (Предположить, что существует хотя бы один такой путь.)
 - (е) Описать алгоритм, вычисляющий число путей из s в t в графе G . (Предположить, что можно выполнить сложение произвольно больших целых чисел за время $O(1)$.)
14. Пусть G – направленный ациклический граф, вершины которого имеют метки из некоторого строго определенного алфавита, и пусть $A[1..\ell]$ – строка из символов того же алфавита. Любой направленный путь в G имеет метку, представляющую собой строку, полученную соединением меток вершин этого пути.
 - (а) Описать алгоритм, который либо находит путь в G , меткой которого является A , либо корректно сообщает, что такого пути не существует.
 - (б) Описать алгоритм поиска числа путей в G , меткой которых является A . (Предположить, что можно выполнить сложение произвольно больших целых чисел за время $O(1)$.)
 - (с) Описать алгоритм поиска максимального пути в G , меткой которого является подпоследовательность строки A .
 - (d) Описать алгоритм поиска минимального пути в G , меткой которого является суперпоследовательность строки A .
 - (е) Описать алгоритм поиска пути в G , метка которого имеет минимальное расстояние редактирования от строки A .

15. *Ломаный путь* (polygonal path) – это последовательность отрезков прямой, соединенных концами, а конечные точки этих отрезков называются вершинами этого пути. *Длиной ломаного пути* является сумма длин этих отрезков. Ломаный путь с вершинами $(x_1, y_1), (x_2, y_2), \dots, (x_k, y_k)$ является *монотонно возрастающим*, если $x_i < x_{i+1}$ и $y_i < y_{i+1}$ для каждого индекса i неформально: каждая вершина этого пути выше и правее предшествующей.

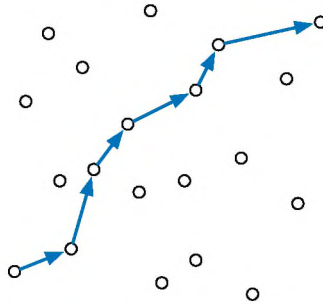


Рис. 6.22. Монотонно возрастающий ломаный путь с семью вершинами через множество точек

Предположить, что задано множество S из n точек на плоскости, представленное как два массива $X[1..n]$ и $Y[1..n]$. Описать и проанализировать алгоритм вычисления длины максимального монотонно возрастающего пути с вершинами из множества S . Предположить, что имеется подпрограмма $Length(x, y, x', y')$, которая возвращает длину отрезка от (x, y) до (x', y') .

16. Для любых двух узлов u и w в направленном ациклическом графе G интервал $G[u, w]$ – это объединение всех направленных путей в G из u в w . Равнозначно: $G[u, w]$ состоит из всех вершин v , таких что $v \in reach(u)$ и $w \in reach(v)$ вместе со всеми ребрами в G , соединяющими эти вершины.

Предположить, что задан направленный ациклический граф G , в котором каждая вершина имеет числовой вес – положительный, отрицательный или ноль.

- Описать эффективный алгоритм поиска интервала с максимальным весом в графе G , где вес каждого интервала является суммой весов его вершин.
- Описать эффективный алгоритм поиска вершины с максимальным весом в каждом интервале в G . Алгоритм должен вычислять двумерный массив $MaxWt[1..V, 1..V]$, где каждый элемент $MaxWt[u, w]$ – максимальный вес по всем вершинам в интервале $G[u, w]$. В частности, если интервал $G[u, w]$ пуст, то элемент $MaxWt[u, w]$ должен быть равен $-\infty$.

17. Пусть G – направленный ациклический граф, вершины которого имеют метки из некоторого строго определенного алфавита. Любой направленный путь в G имеет метку, представляющую собой строку, полученную соединением меток вершин этого пути. Напомню, что палиндром – это строка, которая читается одинаково в обе стороны.

- (a) Описать и проанализировать алгоритм поиска длины максимального палиндрома, являющегося меткой пути в G . Например, для заданного графа на рис. 6.23 алгоритм должен вернуть целое число 6, соответствующее длине палиндрома **HANNAH**.

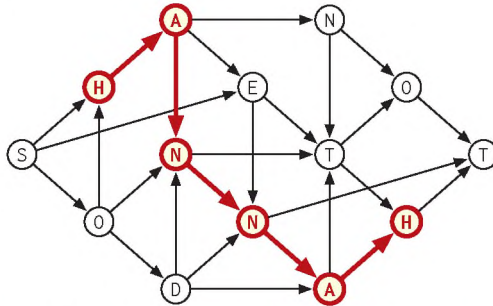


Рис. 6.23. НАГ, в котором метка максимального пути-палиндрома имеет длину 6

- (b) Описать алгоритм поиска самого длинного палиндрома, являющегося подпоследовательностью метки пути в G .
- (c) Предположить, что G содержит единственный источник s и единственный приемник t . Описать алгоритм поиска самого короткого палиндрома, являющегося суперпоследовательностью метки пути в G из s в t .
18. Предположить, что заданы два направленных ациклических графа G и H , в которых каждый узел имеет метку из некоторого конечного алфавита. Различные узлы могут иметь одинаковую метку. Метка пути в любом НАГ – это строка, полученная соединением меток вершин этого пути.
- (a) Описать и проанализировать алгоритм вычисления длины максимальной строки, которая является меткой пути в обоих графах G и H .
- (b) Описать и проанализировать алгоритм вычисления длины максимальной строки, которая является подпоследовательностью метки пути в обоих графах G и H .
- (c) Описать и проанализировать алгоритм вычисления длины самой короткой строки, которая является суперпоследовательностью метки пути в обоих графах G и H . [Подсказка: это проще, чем кажется.]

19. Пусть G – произвольный (необязательно ациклический) направленный граф, в котором каждая вершина v имеет целочисленный вес $w(v)$.
- (а) Описать алгоритм поиска максимального направленного пути в G , веса вершин которого определяют возрастающую последовательность.
 - (б) Описать и проанализировать алгоритм, определяющий вершину с максимальным весом, достижимую из каждой вершины в G . То есть для каждой вершины v алгоритм должен вычислять $\text{maxreach}(v) := \max\{w(x) \mid x \in \text{reach}(v)\}$.
20. (а) Предположить, что задан направленный ациклический граф G с n вершинами и целое число $k \leq n$. Описать эффективный алгоритм поиска множества из не более чем k путей, не пересекающихся в вершинах, которые посещают каждую вершину в G .
- (б) Теперь предположить, что ребра исходного НАГ G имеют веса – положительные, отрицательные или нули. Описать эффективный алгоритм поиска множества из не более чем k путей, не пересекающихся в вершинах, с минимальным общим весом, которые посещают каждую вершину в G .

Эти алгоритмы должны выполняться за время $O(n^{k+c})$ при некоторой малой константе c . Одна вершина – это путь с весом ноль. (Мы рассмотрим более эффективный алгоритм для части (а) в главе 11.)

21. Крис – профессиональный скалолаз, участник национальных соревнований в США. На соревнованиях от Криса требуется использование столькох зацепов на скалодроме, сколько возможно, и выполнение только тех переходов, которые явно разрешены установщиком маршрута.

На скалодроме имеется n зацепов. Крис получил список m пар (x, y) зацепов. Каждая пара означает, что разрешен переход напрямую от зацепа x к зацепу y , но прямой переход от y к x не разрешен, если в список также не включена пара (y, x) . Крис должен определить последовательность разрешенных переходов, в которой используется столько зацепов, сколько возможно, поскольку каждый новый зацеп увеличивает счет на одно очко. Правила позволяют Крису выбрать первый и последний зацеп на маршруте скалодрома. Правила также позволяют использовать каждый зацеп столько раз, сколько нужно, но только первое использование каждого зацепа увеличивает счет Криса.

- (а) Определить естественный граф, представляющий входные данные. Описать и проанализировать алгоритм, который решает задачу скалолазания Криса, если вы обеспечите, чтобы исходный граф являлся НАГом.

- (b) Описать и проанализировать алгоритм решения задачи скалолазания Криса без ограничений для исходного графа.

Оба алгоритма должны выводить максимальное возможное число очков, которое может набрать Крис.

22. Существует n галактик, связанных m межгалактическими маршрутами телепортации. Каждый маршрут телепортации соединяет две галактики, и по нему можно перемещаться в обоих направлениях. Но компания, обслуживающая маршруты телепортации, установила весьма выгодную структуру оплаты: любой может телепортироваться из своей домашней галактики вообще без оплаты, но оплата телепортации в домашнюю галактику непомерно высока.

Джуди решила в выходные дни совершить путешествие по вселенной, посещая как можно больше галактик, начиная с домашней. Для экономии транспортных расходов Джуди хочет телепортироваться из своей домашней галактики на каждом шаге, за исключением самой последней телепортации домой.

- (a) Описать и проанализировать алгоритм, вычисляющий максимальное число галактик, которые может посетить Джуди. Входные данные состоят из ненаправленного графа G с n вершинами и m ребрами, описывающего сеть телепортации, целого числа $1 \leq s \leq n$, идентифицирующего домашнюю галактику Джуди, и массива $D[1..n]$, содержащего расстояния каждой галактики от s .

- ♥(b) Прямо перед своим путешествием по вселенной Джуди выигрывает в космическую лотерею денежную сумму, как раз достаточную, чтобы оплатить две телепортации в свою домашнюю галактику. Описать новый алгоритм вычисления максимального числа различных галактик, которые может посетить Джуди. Она может посетить одну и ту же галактику более одного раза, но только первое посещение учитывается в общей сумме.

23. Доктор Ху (Doctor Who) и Ривер Сонг (River Song) решили сыграть в игру на направленном ациклическом графе G , содержащем один источник s и один приемник t ⁸.

У каждого игрока есть фишка на одной из вершин графа G . В начале игры фишка Доктора находится в вершине-источнике s , а фишка Ривер – в вершине-приемнике t . Игроки ходят по очереди, первый ход делает Доктор. При каждом ходе Доктор передвигает свою фишку вперед по направленному ребру, а Ривер – в обратном направлении ребра.

⁸ Метки s и t – это аббревиатуры от Untempered Schism (Вневременной разлом) и Time Vortex (Временной вихрь) (эпизод «Барабанная дробь»), или Shining World of the Seven Systems (Сияющий мир семи систем) (также известный как Галлифрей (Gallifrey)) и Trenzalore (Трензалор (планета)), или Skaro (Скаро (планета)) и Telos (Телос), или Something else Timey-wimey (что-то еще временное-швременное).

Если две фишки встречаются в одной вершине, Ривер выигрывает. («Hello, Sweetie!» («Привет, дорогой!»)) Если фишка Доктора достигает t или фишка Ривер достигает s прежде, чем фишки встречаются, то выигрывает Доктор.

Описать и проанализировать алгоритм, определяющий, кто выигрывает описанную выше игру, предполагая, что оба участника играют безошибочно. То есть если Доктор может выиграть независимо от того, как ходит Ривер, то алгоритм должен вывести «Doctor», а если Ривер может выиграть независимо от того, как ходит Доктор, то алгоритм должен вывести «River». (Почему существуют только эти две возможности?) Входными данными для алгоритма является граф G .

- ♥ 24. Пусть $x = x_1x_2 \dots x_n$ – заданная строка из n символов на некотором конечном алфавите Σ , и пусть A – детерминированный конечный автомат с m состояниями на том же алфавите.
- (a) Описать и проанализировать алгоритм вычисления длины максимальной подпоследовательности в x , которая принимается конечным автоматом A . Например, если A принимает язык $(AR)^*$ и $x = \text{ABRACADABRA}$, то алгоритм должен вывести число 4, т. е. длину подпоследовательности $ARAR$.
 - (b) Описать и проанализировать алгоритм вычисления длины минимальной суперпоследовательности в x , которая принимается конечным автоматом A . Например, если A принимает язык $(ABCD)^*$ и $x = \text{ABRACADABRA}$, то алгоритм должен вывести число 25, т. е. длину суперпоследовательности $ABCDRABCDRABCDRABCDRABCDR$.

Проанализировать алгоритмы относительно длины n исходной строки, числа m состояний конечного автомата и размера алфавита Σ .

25. Не каждый алгоритм динамического программирования можно смоделировать как поиск оптимального пути в направленном ациклическом графе, но каждый алгоритм динамического программирования реально обрабатывает некоторый внутренний граф зависимостей в обратном порядке обхода.
- (а) Предположить, что задан направленный ациклический граф G , в котором каждый узел хранит числовой ключ поиска. Описать и проанализировать алгоритм поиска максимального двоичного дерева поиска, которое является подграфом G .
 - (б) Предположить, что задан направленный ациклический граф G и две вершины s и t . Описать алгоритм определения числа направленных путей в графе G из s в t . (Предположить, что любая арифметическая операция требует времени $O(1)$.)

(с) Пусть G – направленный ациклический граф со следующими свойствами:

- граф G содержит один источник s и несколько приемников $t_1, t_2, \dots, t_k$;
- с каждым ребром $v \rightarrow w$ связан вес $p(v \rightarrow w)$ от 0 до 1;
- для каждой вершины v , не являющейся приемником, суммарный вес всех ребер, исходящих из v , равен 1, т. е. $\sum_w p(v \rightarrow w) = 1$.

Веса $p(v \rightarrow w)$ определяют случайное блуждание в G из источника s в некоторый приемник t_i . После достижения любой вершины v , не являющейся приемником, блуждание следует по ребру $v \rightarrow w$ с вероятностью $p(v \rightarrow w)$. Все вероятности взаимно независимые. Описать и проанализировать алгоритм вычисления вероятности того, что это случайное блуждание достигает источника t_i для каждого индекса i . (Предположить, что любая арифметическая операция требует времени $O(1)$.)

Глава 7

.....

Минимальные остовные деревья

Все мы должны держаться вместе, джентльмены, иначе, несомненно, будем висеть по отдельности.

— Бенджамин Франклин (Benjamin Franklin),
при подписании Декларации независимости (Declaration of Independence) (4 июля 1776 г.)

Помню, как обращался за советом к кому-то, — кто бы это мог быть? — стоит ли публиковать эту работу; причиной сомнений была ее чрезвычайная простота... К счастью, он посоветовал мне продолжать без сомнений, и прошло много лет, прежде чем еще одна из моих публикаций стала такой же известной, как эта — очень простая.

— Джозеф Крускал (Joseph Kruskal),
описывая свой алгоритм минимального остовного дерева (1997 г.)

Очищайте ВСЁ! (Clean ALL the things!)

— Элли Брош (Allie Brosh),
«Вот почему я никогда не стану взрослой» («This is Why I'll Never be an Adult»),
веб-комиксы и блог «Преувеличение с половиной» («Hyperbole and a Half», 17 июня 2010 г.)

Предположим, что задан связный, ненаправленный, взвешенный граф. Это граф $G = (V, E)$ вместе с функцией $w: E \rightarrow \mathbb{R}$, которая присваивает вещественный вес $w(e)$ каждому ребру e . Вес может быть положительным, отрицательным или нулевым. В этой главе описывается несколько алгоритмов поиска минимального остовного дерева графа G , т. е. остовное дерево T , которое минимизирует функцию:

$$w(T) := \sum_{e \in T} w(e).$$

Пример см. на рис. 7.1.

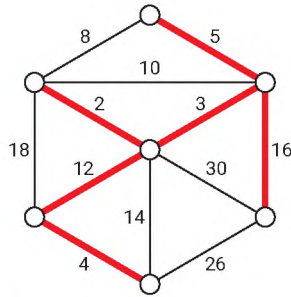


Рис. 7.1. Взвешенный граф и его минимальное остовное дерево

7.1. Различные веса ребер

Досадной тонкой деталью в формулировке этой задачи является то, что взвешенные графы могут иметь более одного остовного дерева с одинаковым минимальным весом. В особенности если каждое ребро в G имеет вес 1, то каждое остовное дерево в G является минимальным остовным деревом с весом $V - 1$. Эта неоднозначность затрудняет разработку алгоритмов, все могло бы стать легче, если бы мы просто предположили, что минимальные остовные деревья являются неповторяющимися.

К счастью, существует простое условие, которое предполагает необходимую нам неповторяемость.

Лемма 7.1. Если все веса ребер в связном графе G различны, то G содержит неповторяющееся минимальное остовное дерево.¹

Доказательство. Пусть G – произвольный связный граф с двумя минимальными остовными деревьями T и T' . Нам надо доказать, что некоторая пара ребер в G имеет одинаковый вес. По существу, это доказательство является жадным алгоритмом перестановки («обмена») аргументов.

Каждое из рассматриваемых здесь остовных деревьев обязательно должно содержать ребро, отсутствующее в другом дереве. Пусть e – ребро с минимальным весом в $T' \setminus T$, и пусть e' – ребро с минимальным весом в $T \setminus T'$ (разрешение ничейной ситуации в произвольном порядке). Без потери для обобщения предположим, что $w(e) \leq w(e')$.

Подграф $T' \cup \{e\}$ содержит ровно один цикл C , который проходит через ребро e . Пусть e'' – любое ребро в этом цикле, которого нет в T . Как минимум одно такое ребро обязательно должно существовать, потому что T – дерево. (Мы можем иметь или не иметь $e'' = e'$.) Поскольку $e \in T$, сразу же получаем $e'' \neq e$, следовательно, $e'' \in T' \setminus T$. Из этого следует, что $w(e'') \geq w(e') \geq w(e)$.

Теперь рассмотрим остовное дерево $T'' = T' + e - e''$. (Это новое дерево T'' может быть равным T .) Мы сразу же получаем $w(T'') = w(T') + w(e) - w(e'') \leq w(T)$.

¹ Обратная лемма ложна – связный граф с повторяющимися весами ребер все же может содержать неповторяющееся остовное дерево. Тривиальный пример: предположим, что граф G – дерево.

Но T' – минимальное остовное дерево, поэтому мы непременно получаем $w(T'') = w(T')$, другими словами, T'' – это также минимальное остовное дерево. Из этого заключаем, что $w(e) = w(e'')$, т. е. доказательство завершено. $\square$

Если у нас уже есть алгоритм, в котором предполагаются различные веса ребер, то можно продолжать выполнять его на графах, где некоторые ребра имеют равные веса, пока у нас есть непротиворечивый метод для разрешения ничейных ситуаций. Один такой метод использует следующий алгоритм вместо простого сравнения весов. Как входные данные алгоритм ShorterEdge принимает четыре целые числа i, j, k, l , представляющие четыре (не обязательно различные) вершины, и решает, какое из двух ребер (i, j) и (k, l) имеет «меньший» вес. (Поскольку исходный граф ненаправленный, пары (i, j) и (j, i) представляют одно и то же ребро.)

```
ShorterEdge(i, j, k, l)
  if w(i, j) < w(k, l)      then return (i, j)
  if w(i, j) > w(k, l)      then return (k, l)
  if min(i, j) < min(k, l)  then return (i, j)
  if min(i, j) > min(k, l)  then return (k, l)
  if max(i, j) < max(k, l)  then return (i, j)
  << if max(i, j) > max(k, l) >> return (k, l)
```

Имея в виду лемму 7.1 и это правило для разрешения ничьи, мы без риска предполагаем в остальной части главы, что веса ребер всегда различны, следовательно, минимальные остовные деревья всегда неповторяющиеся (единственные в своем роде). В сущности, мы можем свободно обсуждать минимальное остовное дерево без каких-либо сомнений.

7.2. Единственный алгоритм минимального остовного дерева

Существует множество алгоритмов вычисления минимальных остовных деревьев, но почти все они являются вариантами описанной ниже общей стратегии. Здесь наблюдается аналогия с обходом графов, где несколько различных алгоритмов представляют собой варианты обобщенного алгоритма обхода с поиском в любом направлении.

Обобщенный алгоритм минимального остовного дерева работает с ациклическим подграфом F исходного графа G , который мы будем называть промежуточным остовным лесом (intermediate spanning forest). При любых условиях F соответствует следующему условию инвариантности:

F – подграф минимального остовного дерева графа G .

Изначально F состоит из V деревьев с одной вершиной. Обобщенный алгоритм соединяет деревья в F , добавляя определенные ребра между ними. Когда алгоритм останавливается, F состоит из единственного остовного дерева. Наш инвариант предполагает, что это непременно должно быть минимальное остовное дерево графа G . Очевидно, мы должны внимательно относиться к тому, какие именно ребра добавляются к растущему лесу, потому что не каждое ребро входит в состав минимального остовного дерева.

На любом этапе формирования промежуточный остовный лес F порождает два особых типа ребер в оставшейся части графа.

- Ребро является бесполезным (useless), если оно не представляет собой ребро F , но обе его конечные точки находятся в той же компоненте F .
- Ребро является надежным (safe), если оно представляет собой ребро с минимальным весом с ровно одной конечной точкой в той же компоненте F .

Одно и то же ребро может быть надежным для двух различных компонент F . Некоторые ребра $G \setminus F$ не являются ни надежными, ни бесполезными – мы называем их *неоднозначными* (undecided).

Все алгоритмы минимального остовного дерева основаны на двух простых высказываниях. Первое высказывание было доказано Робертом Примом (Robert Prim) в 1957 г. (хотя неявно применялось в нескольких более ранних алгоритмах), а второе напрямую связано с первым.

Лемма 7.2 (Лемма Прима). Минимальное остовное дерево графа G содержит все надежные ребра.

Доказательство. В действительности мы доказываем следующее более строгое утверждение: для любого подмножества S вершин графа G минимальное остовное дерево G содержит ребро с минимальным весом с ровно одной конечной вершиной в S . Как и в предыдущей лемме, мы доказываем это утверждение с использованием техники «жадного обмена» (greedy exchange).

Пусть S – произвольное подмножество вершин графа G , и пусть e – самое легкое ребро с ровно одной конечной точкой в S . (Наше предположение о том, что все веса ребер различны, предполагает, что e является неповторяющимся ребром.) Пусть T – произвольное остовное дерево, которое не содержит e . Необходимо доказать, что T не является минимальным остовным деревом в G .

Поскольку T – связный граф, он содержит путь из одной конечной вершины ребра e в другую. Так как этот путь начинается в одной из вершин S и заканчивается в одной из вершин не из подмножества S , он обязательно должен содержать как минимум одно ребро с ровно одной конечной вершиной в S . Пусть e' – любое такое ребро. Поскольку T – ациклический граф, удаление e' из T создает остовный лес с ровно двумя компонента-

ми, из которых одна содержит каждую конечную вершину e . Следовательно, добавление e в этот лес создает новое остовное дерево $T' = T - e' + e$. По определению e предполагается, что $w(e') > w(e)$, из чего следует, что T' имеет меньший общий вес, чем T . Следовательно, T не является минимальным остовным деревом G , что и требовалось доказать. $\square$

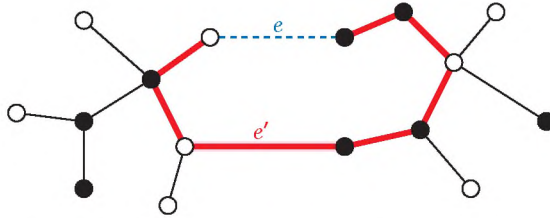


Рис. 7.2. Каждое надежное ребро находится в минимальном остовном дереве. Черные вершины содержатся в подмножестве S

Лемма 7.3. Минимальное остовное дерево не содержит бесполезных ребер.

Доказательство. Добавление любого бесполезного ребра в F вводит цикл. $\square$

Наш обобщенный алгоритм минимального остовного дерева повторно добавляет надежные ребра в растущий лес F . Если F пока не является связным, то обязательно должно существовать как минимум одно надежное ребро, потому что исходный граф G связный. Следовательно, вне зависимости от того, какие ребра мы добавляем на каждой итерации, наш обобщенный алгоритм в итоге делает F связным. По индукции лемма 7.2 предполагает, что итоговое дерево действительно является минимальным остовным деревом. Когда мы добавляем новые ребра в F , некоторые неоднозначные ребра могут становиться надежными, а другие неоднозначные ребра – бесполезными. (Когда ребро становится бесполезным, оно остается таким навсегда.) Для полной спецификации конкретного алгоритма необходимо обязательно описать, какое надежное ребро (или несколько ребер) добавляется на каждой итерации и как найти эти ребра.

7.3. Алгоритм Борувки

Самый старый и, вероятно, самый простой алгоритм минимального остовного дерева был разработан чешским математиком Отакаром Борувкой (Otakar Borůvka) в 1926 г., примерно через год после того, как Индржих Саксел (Jindřich Saxel) спросил его, как сконструировать электрическую сеть, соединяющую несколько городов, используя наименьшее количество

проводов². Этот алгоритм был повторно разработан Густавом Шоке (Gustav Choquet) в 1938 г., еще раз повторно открыт группой польских математиков, возглавляемой Йозефом Лукашевичем (Józef Łukasiewicz) в 1951 г., и в очередной раз повторно открыт Джорджем Соллином (George Sollin) в 1961 г. Хотя Соллин никогда не публиковал свою разработку, он подробно описал ее и зафиксировал авторские права в одной из первых книг по графовым алгоритмам. В итоге этот алгоритм иногда называют алгоритмом Соллина.

Суть алгоритма Борувки / Флорека–Лукашевича–Перкала–Штейнхауса–Зубржицки / Прима / Соллина / Брош (Borůvka / Choquet / Florek–Łukaszewicz–Perkal–Steinhaus–Zubrzycki / Prim / Sollin / Brosh)³ можно выразить одной строкой:

Borůvka: Добавить **BCE** надежные ребра и выполнить рекурсию.

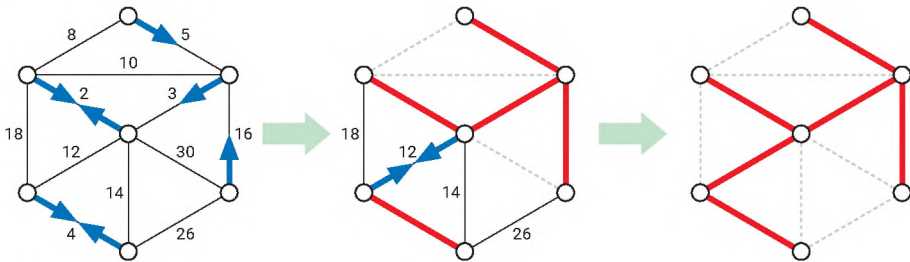


Рис. 7.3. Работа алгоритма Борувки на примере графа.

Утолщенные красные ребра находятся в F , пунктирные ребра – бесполезные.

Стрелки направлены вдоль каждого надежного ребра компоненты.

Алгоритм завершается после всего лишь двух итераций

Ниже приведено более подробное описание алгоритма Борувки. Он вызывает алгоритм CountAndLabel из главы 5 (стр. 258) для подсчета компонент в F и пометки каждой вершины v целым числом $comp(v)$, обозначающим соответствующую компоненту.

```

Borůvka( $V, E$ ):
     $F = (V, \emptyset)$ 
    count  $\leftarrow$  CountAndLabel( $F$ )
    while count > 1
        AddAllSafeEdges( $E, F, \text{count}$ )
        count  $\leftarrow$  CountAndLabel( $F$ )
    return  $F$ 

```

² Саксел был сотрудником энергетической компании Западной Моравии (West Moravian Power Company), и Борувка характеризовал его как «весьма талантливого и трудолюбивого (инженера)». Позже Саксел был казнен нацистами как «лицо иудейского происхождения».

³ Прочтите все от корки до корки в «Гиперболе с половиной» («Hyperbole and a Half»). Затем пойдите и купите эту книгу. И второй экземпляр для своего кота. Что такое? У вас нет кота? Что же вы за чудовище? Заведите кота и купите для него второй экземпляр «Гиперболы с половиной».

Остается только описать, как идентифицировать и добавить все надежные ребра в F . Предположим, что F содержит более одной компоненты, поскольку иначе все уже выполнено. Приведенная ниже подпрограмма формирует массив $safe[1..V]$ из надежных ребер, где $safe[i]$ – ребро минимального веса с одной конечной точкой в i -й компоненте F методом проверки прямым перебором каждого ребра в G . Для каждого ребра uv если u и v находятся в одной компоненте, то uv либо бесполезное, либо уже является ребром в F . В противном случае мы сравниваем вес uv с весами в $safe[comp(u)]$ и $safe[comp(v)]$ и обновляем элементы массива, если необходимо. После идентификации всех надежных ребер мы добавляем каждое ребро $safe[i]$ в F .

```

AddAllSafeEdges(E, F, count):
  for i ← 1 to count
    safe[i] ← Null
  for each edge uv ∈ E
    if comp(u) ≠ comp(v)
      if safe[comp(u)] = Null or w(uv) < w(safe[comp(u)])
        safe[comp(u)] ← uv
      if safe[comp(v)] = Null or w(uv) < w(safe[comp(v)])
        safe[comp(v)] ← uv
  for i ← 1 to count
    add safe[i] to F

```

Каждый вызов CountAndLabel выполняется за время $O(V)$, потому что лес F содержит не более $V - 1$ ребер. AddAllSafeEdges выполняется за время $O(V + E)$, поскольку мы тратим константное время на каждую вершину, каждое ребро в G и каждую компоненту F . Так как исходный граф связный, получаем $V \leq E + 1$. Из этого следует, что для каждой итерации цикла while алгоритма Борувки требуется время $O(E)$.

Каждая итерация сокращает число компонент в F как минимум с коэффициентом 2 – в наихудшем случае компоненты F объединяются в пары. Поскольку изначально F содержит V компонент, в цикле while выполняется не более $O(\log V)$ итераций. Мы приходим к выводу о том, что общее время выполнения алгоритма Борувки равно $O(E \log V)$.

Это тот самый алгоритм МОД, который вам нужен

Несмотря на относительно неясное происхождение, первые западные исследователи алгоритмов знали о существовании алгоритма Борувки, но отвергали его как «слишком сложный». В итоге, несмотря на простоту и эффективность, большинство книг по алгоритмам и структурам данных, к сожалению, даже не упоминает об алгоритме Борувки. Это упущение является серьезной ошибкой, так как алгоритм Борувки обладает несколькими преимуществами, отличающими его от прочих классических алгоритмов минимального остовного дерева (МОД).

- Алгоритм Борувки часто выполняется быстрее, чем $O(E \log V)$ в наихудшем случае. Число компонент в F можно существенно снизить с коэффициентом, большим 2, за одну итерацию, сократив число итераций до значения, меньшего, чем в наихудшем случае $\lceil \log_2 V \rceil$.
- Небольшая модификация алгоритма Борувки (в действительности более близкая к первоначальному варианту Борувки) позволяет реально обрабатывать за время $O(E)$ широкий класс любопытных графов, включая графы, которые можно изобразить на плоскости без пересечений ребер. В противоположность этому анализ времени для двух других алгоритмов применим для всех графов.
- Алгоритм Борувки поддерживает возможность значительного распараллеливания: на каждой итерации каждая компонента F может быть обработана в отдельном независимом потоке (thread). Из этого следует, что параллельное выполнение обеспечивает даже более высокую производительность на многоядерных или распределенных системах. В противоположность этому два других классических алгоритма МОД являются по своей сущности последовательными.
- Некоторые более современные алгоритмы минимального остовного дерева даже в наихудшем случае быстрее, чем описанные здесь классические алгоритмы. Все эти более быстрые алгоритмы являются обобщениями алгоритма Борувки.

Короче говоря, если вам вдруг потребуется реализация алгоритма минимального остовного дерева, используйте алгоритм Борувки. С другой стороны, если необходимо эффективно доказать свойства или характеристики минимальных остовных деревьев, то вы действительно должны знать еще и два описанных ниже алгоритма.

7.4. Алгоритм Ярника (Прима)

Описанный ниже самый старый алгоритм минимального остовного дерева впервые был описан чешским математиком Войцехом Ярником (Vojtěch Jarník) в 1929 г. в письме Борувке. Ярник опубликовал свой алгоритм в следующем году. Тот же алгоритм был независимо повторно разработан Йозефом Крускалом (Joseph Kruskal) в 1956 г., (вероятно) Робертом Примом (Robert Prim) в 1957 г., Гарри Лоберманом (Harry Loberman) и Арнольдом Вейнбергером (Arnold Weinberger) в 1957 г., наконец, Эдсгером Дейкстрой (Edsger Dijkstra) в 1958 г. Все они – Прим, Лоберман, Вейнбергер и Дейкстра – (со временем) узнали и даже цитировали работу Крускала, но, поскольку в той же работе Крускал также описал два других алгоритма минимального остовного дерева, именно этот алгоритм обычно называют алгоритмом Прима или иногда алгоритмом Прима–Дейкстры, даже несмотря на то, что в 1958 г. Дейкстра уже разработал другой алгоритм (неправильно) названный его именем.

В алгоритме Ярника промежуточный лес F содержит только одну нетривиальную компоненту T , а все прочие компоненты являются изолированными вершинами. Изначально T состоит из единственной произвольной вершины исходного графа. Алгоритм повторяет следующий шаг до тех пор, пока T не охватит весь граф:

Jarník: Повторное добавление в T надежного ребра из T .

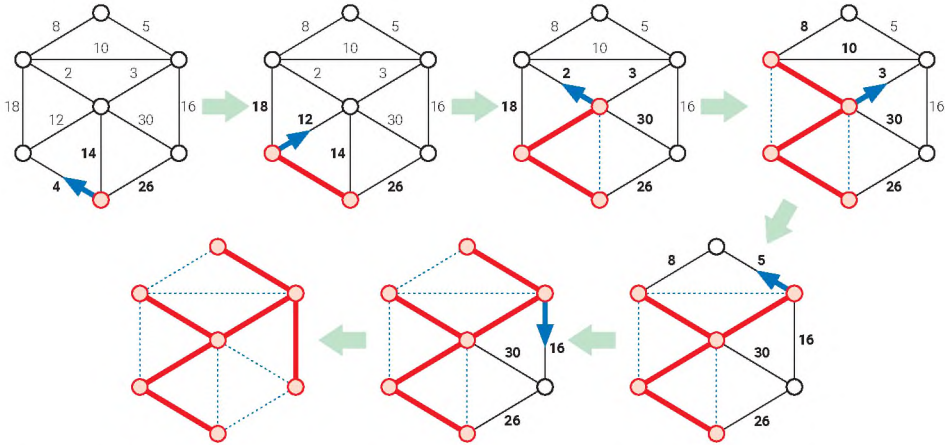


Рис. 7.4. Алгоритм Ярника выполняется на примере графа, начиная с нижней вершины. На этом этапе утолщенные красные ребра находятся в T , стрелки направлены вдоль надежных ребер T , а штриховые ребра являются бесполезными

Для реализации алгоритма Ярника мы сохраняем все ребра, примыкающие к T , в очереди с приоритетами. Когда мы извлекаем ребро с минимальным весом из очереди с приоритетами, то сначала проверяем, находятся ли обе его конечные вершины в T . Если не находятся, то мы добавляем это ребро в T , затем добавляем новые соседние ребра в очередь с приоритетами. Другими словами, алгоритм Ярника представляет собой вариант поиска по первому наилучшему совпадению, описанного в конце главы 5. Если внутренняя очередь с приоритетами реализуется с использованием стандартной двоичной кучи, то алгоритм Ярника выполняется за время $O(E \log E) = O(E \log V)$.

♥ Улучшенный алгоритм Ярника

Можно улучшить алгоритм Ярника, используя более сложную структуру данных очереди с приоритетами, называемую фибоначиевой кучей (Fibonacci heap), впервые описанной Майклом Фридманом (Michael Fredman) и Робертом Тарьяном (Robert Tarjan) в 1984 г. Как и двоичные кучи, фибоначиевые кучи поддерживают стандартные операции очереди с приоритетами Insert, ExtractMin и DecreaseKey. Но в отличие от стандарт-

ных двоичных куч, которые требуют времени $O(\log n)$ для каждой операции, фибоначиевы кучи поддерживают выполнение операций Insert и DecreaseKey за постоянное амортизированное время. Амортизированная стоимость операции ExtractMin остается равной $O(\log n)$.⁴

Для применения этой более быстрой структуры данных мы сохраняем вершины, а не ребра графа G в очереди с приоритетами, где приоритетом каждой вершины v является либо ребро с минимальным весом между v и растущим деревом T , либо ∞ , если такого ребра не существует. Мы можем вставить (Insert) все вершины в очередь с приоритетами в начале алгоритма, затем при добавлении нового ребра в T может потребоваться увеличение приоритетов некоторых соседних вершин.

Чтобы упростить описание, разделим алгоритм на две части (см. рис. 7.5). JarnikInit инициализирует очередь с приоритетами, JarnikLoop – это основной алгоритм. Входные данные состоят из вершин и ребер графа вместе с начальной вершиной s . Для каждой вершины v обрабатывается ее приоритет $priority(v)$ и смежное ребро $edge(v)$, такое что $w(edge(v)) = priority(v)$.

```
Jarnik(V, E, s):
  JarnikInit(V, E, s)
  JarnikLoop(V, E, s)
```

```
JarnikInit(V, E, s):
  for each vertex  $v \in V \setminus \{s\}$ 
    if  $vs \in E$ 
       $edge(v) \leftarrow vs$ 
       $priority(v) \leftarrow w(vs)$ 
    else
       $edge(v) \leftarrow \text{Null}$ 
       $priority(v) \leftarrow \infty$ 
  Insert( $v$ )
```

```
JarnikLoop(V, E, s):
   $T \leftarrow (\{s\}, \emptyset)$ 
  for  $i \leftarrow 1$  to  $|V| - 1$ 
     $v \leftarrow \text{ExtractMin}$ 
    add  $v$  and  $edge(v)$  to  $T$ 
    for each neighbor  $u$  of  $v$ 
      if  $u \notin T$  and  $priority(u) > w(uv)$ 
         $edge(u) \leftarrow uv$ 
        DecreaseKey( $u, w(uv)$ )
```

Рис. 7.5. Алгоритм Ярника для минимального остовного дерева готов к использованию с фибоначиевой кучей

Каждая операция Insert и ExtractMin вызывается $O(V)$ раз, по одному для каждой вершины, исключая s , а операция DecreaseKey вызывается $O(E)$ раз, не чаще двух раз для каждого ребра. Таким образом, если мы используем фибоначиеву кучу, то улучшенный алгоритм выполняется за время $O(E + V \log V)$, т. е. быстрее алгоритма Борушки, исключая случай $E = O(V)$.

⁴ Амортизированное время – это бухгалтерский прием, позволяющий игнорировать не слишком частые колебания во времени выполнения одной операции со структурой данных. Фибоначиева куча может выполнять любую смешанную последовательность операций I Insert, D DecreaseKey и X ExtractMin за время $O(I + D + X \log n)$ в наихудшем случае. Поэтому в среднем операции Insert и DecreaseKey требуют постоянного времени, а операция ExtractMin в среднем требует времени $O(\log n)$, но некоторые отдельные операции могут выполняться дольше в наихудшем случае. Амортизация использует статистическое усреднение по последовательности операций. Здесь не делается никаких предположений о случайности, о входных данных или об алгоритме.

Но на практике это улучшение редко бывает быстрее, чем простейшая реализация, использующая двоичную кучу, если только граф не является чрезвычайно большим и плотным. Алгоритмы с фибоначиевой кучей достаточно сложны, и скрытые константы в выражениях времени и пространства выполнения являются существенными – не чрезмерными, но определенно большими, чем скрытая константа 1 в граничном времени $O(\log n)$ для операций двоичной кучи.

7.5. Алгоритм Краскала

Последний алгоритм минимального остовного дерева, который мы будем рассматривать, был впервые описан Йозефом Краскалом (Joseph Kruskal) в 1956 г. в той же работе, где он заново открыл алгоритм Ярника. Краскал был вдохновлен «печатным переводом (невразумительного источника)» оригинального документа Борувки, который «бродил» по математическому факультету Принстонского университета. Краскал нашел алгоритм Борувки «излишне усложненным»⁵. Этот же алгоритм был повторно разработан в 1957 г. Гарри Лоберманом (Harry Loberman) и Арнольдом Вейнбергером (Arnold Weinberger), но каким-то образом удалось избежать его переименования.

Как и ранее рассмотренные алгоритмы минимального остовного дерева, алгоритм Краскала имеет удобное для запоминания однострочное описание:

Kruskal: Проверить все ребра по возрастанию веса;
если ребро надежное, то добавить его в F .

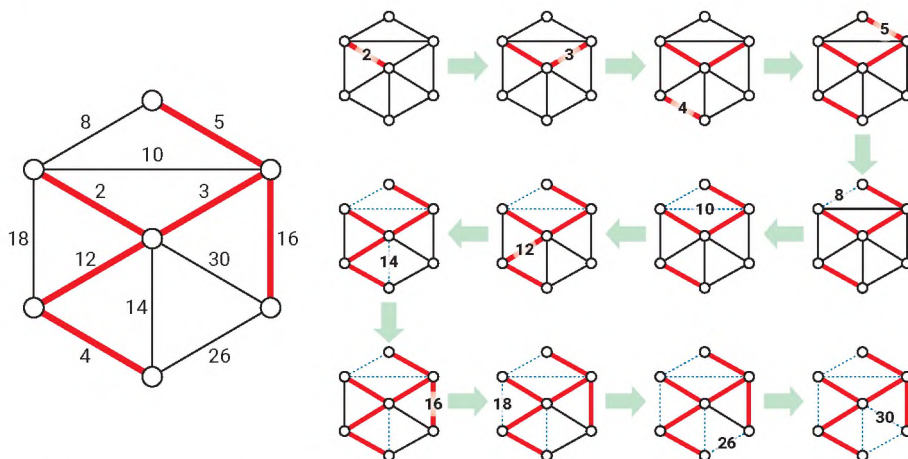


Рис. 7.6. Алгоритм Краскала работает на примере графа. Утолщенные красные ребра находятся в F , тонкие штриховые ребра – бесполезные

⁵ Справедливости ради, первая работа Борувки действительно была излишне усложнена, отчасти потому, что она была написана для математиков формальным языком (линейной) алгебры, а не на языке графов. Следующая работа Борувки, также опубликованная в 1927 г., но в электротехническом журнале, была написана простым языком для более широкой аудитории, особенно в ее современном изложении в наши дни. Краскал, очевидно, не знал о второй статье Борувки. Дурацкий «железный занавес».

Самый простой метод просмотра ребер в порядке возрастания веса – сортировка ребер по весу за время $O(E \log E)$, а затем применение простого цикла for к отсортированному списку ребер. Как мы вскоре увидим, эта предварительная сортировка является преобладающей компонентой времени выполнения алгоритма.

Поскольку мы исследуем ребра в порядке от самого малого до самого большого веса, любое исследуемое ребро является надежным, если и только если его конечные точки находятся в различных компонентах леса F . Предположим, что нам встретилось ребро e , соединяющее две компоненты A и B , но не являющееся надежным. Тогда обязательно должно существовать ребро e' с меньшим весом с ровно одной конечной точкой в A . Но это невозможно, потому что (по индукции) каждое ранее проверенное ребро имеет обе конечные точки в одной и той же компоненте леса F .

Как и в алгоритме Борувки, каждая вершина в F должна «знать», какая компонента F содержит ее. Но, в отличие от алгоритма Борувки, мы не вычисляем повторно метки всех компонент с нуля каждый раз, когда добавляем ребро. Вместо этого, когда две компоненты соединяются ребром, меньшая компонента наследует метку большей компоненты, т. е. мы выполняем обход меньшей компоненты (методом поиска в любом направлении). Этот обход требует времени $O(1)$ для каждой вершины в меньшей компоненте. При каждой смене метки компоненты в вершине компонента леса F , содержащая эту вершину, возрастает как минимум с коэффициентом 2, следовательно, метка каждой вершины изменяется не более $O(\log V)$ раз. Из этого следует, что общее время, затрачиваемое на обновление вершин, равно всего лишь $O(V \log V)$.

В более общем смысле алгоритм Краскала содержит часть вершин графа G в непересекающихся подмножествах (в нашем случае в компонентах леса F), используя структуру данных, которая поддерживает следующие операции:

- $\text{MakeSet}(v)$ – создание множества, содержащего только вершину v ;
- $\text{Find}(v)$ – возврат уникального идентификатора множества, содержащего v ;
- $\text{Union}(u, v)$ – замена множеств, содержащих u и v , на их объединение. (Эта операция уменьшает число множеств.)

Ниже приведено полное описание алгоритма Краскала на основании этих операций.

```
Kruskal( $V, E$ ):
  sort  $E$  by increasing weight
   $F \leftarrow (V, \emptyset)$ 
  for each vertex  $v \in V$ 
    MakeSet( $v$ )
  for  $i \leftarrow 1$  to  $|E|$ 
     $uv \leftarrow i$ th lightest edge in  $E$ 
    if  $\text{Find}(u) \neq \text{Find}(v)$ 
      Union( $u, v$ )
      add  $uv$  to  $F$ 
  return  $F$ 
```

После начальной сортировки алгоритм выполняет ровно V операций `MakeSet` (по одной для каждой вершины), $2E$ операций `Find` (по две для каждого ребра) и $V - 1$ операций `Union` (по одной для каждого ребра в минимальном остовном дереве). Мы только что описали структуру данных непересекающихся множеств (*disjoint-set data structure*), для которой операции `MakeSet` и `Find` требуют времени $O(1)$, а операция `Union` выполняется за амортизированное время $O(\log V)$. При использовании этой реализации общее время на обработку части множества равно $O(E + V \log V)$ ⁶.

Но следует напомнить, что нам уже необходимо время $O(E \log E) = O(E \log V)$ только лишь для сортировки ребер. Поскольку это время превышает время, затрачиваемое на обработку структуры данных `Union-Find`, общее время выполнения алгоритма Крускала равно $O(E \log V)$, в точности равное времени выполнения алгоритма Борувки или алгоритма Ярника с использованием обычной (не фибоначиевой) кучи.

Упражнения

1. Пусть $G = (V, E)$ – произвольный связный граф с взвешенными ребрами.
 - (а) Доказать, что для любого цикла в G минимальное остовное дерево G исключает ребро с максимальным весом из этого цикла.
 - (б) Доказать или опровергнуть: минимальное остовное дерево G включает ребро с минимальным весом в каждый цикл в G .
2. В этой главе мы предполагали, что никакие два ребра в исходном графе не имеют равные веса, из чего следует, что минимальное остовное дерево является неповторяющимся. В действительности менее строгое условие для весов ребер определяет неповторяемость минимального остовного дерева.
 - (а) Описать граф со взвешенными ребрами, который содержит неповторяющееся минимальное остовное дерево, даже если два ребра имеют равные веса.
 - (б) Доказать, что граф G со взвешенными ребрами содержит неповторяющееся минимальное остовное дерево, если и только если выполняются следующие условия:
 - для любого разделения вершин G на два подмножества ребро с минимальным весом с одной конечной точкой в каждом подмножестве является неповторяющимся;

⁶ Другая структура данных непересекающихся множеств, использующая стратегию объединения по рангу с уплотнением пути (*union by rank with path compression*), выполняет каждую операцию `Union` или `Find` за амортизированное время $O(\alpha(V))$, где α – обратная функция Аккермана, почти, но не всегда возвращающая константное значение. Если вам не нравится все время обращаться к «Википедии», просто считайте, что $\alpha(V)$ равно 4. При использовании этой реализации общее время на обработку части множества равно $O(E \alpha(V))$, что немного быстрее при больших значениях V и если значение E очень близко к V .

- ребро с максимальным весом в любом цикле G является неповторяющимся.
 - (с) Описать и проанализировать алгоритм, определяющий, содержит ли граф неповторяющееся минимальное остовное дерево.
3. Большинство классических алгоритмов минимального остовного дерева использует понятия «надежных» и «бесполезных» ребер, описанные выше в тексте, но существует и другая формулировка. Пусть G – взвешенный ненаправленный граф с различными весами ребер. Мы говорим, что ребро e является опасным (dangerous), если это самое длинное ребро в некотором цикле в G , и полезным (useful), если оно не лежит в любом цикле в G .
- (а) Доказать, что минимальное остовное дерево графа G содержит каждое полезное ребро.
 - (б) Доказать, что минимальное остовное дерево графа G не содержит любое опасное ребро.
 - (с) Описать и проанализировать эффективную реализацию следующего алгоритма, впервые описанного Йозефом Крускалом в той же работе 1956 г., в которой он предложил алгоритм Крускала. Рассматривать ребра графа G в убывающем порядке, и если ребро опасное, то удалять его из G . [Подсказка: этот алгоритм не столь быстр, как обычный алгоритм Крускала.]
4. (а) Описать и проанализировать алгоритм, вычисляющий остовное дерево максимального веса в заданном графе с взвешенными ребрами.
- (б) Множество ребер, разрезающих циклы (feedback edge set), в ненаправленном графе G – это подмножество ребер F , такое что каждый цикл в G содержит как минимум одно ребро из F . Другими словами, удаление каждого ребра в F делает граф G ациклическим. Описать и проанализировать быстрый алгоритм, вычисляющий множество ребер, разрезающих циклы, с минимальным весом для заданного графа с взвешенными ребрами.
5. Предположим, что задан ненаправленный граф G со взвешенными ребрами и минимальное остовное дерево T графа G .
- (а) Описать алгоритм обновления этого минимального остовного дерева, когда вес одного ребра e уменьшается.
 - (б) Описать алгоритм обновления этого минимального остовного дерева, когда вес одного ребра e увеличивается.

В обоих случаях входными данными для алгоритма являются ребро e и его новый вес. Алгоритм должен изменить T так, чтобы оно осталось минимальным остовным деревом. [Подсказка: рассмотрите отдельно случаи $e \in T$ и $e \notin T$.]

6. (а) Описать и проанализировать алгоритм поиска второго наименьшего остовного дерева заданного графа G , т. е. остовное дерево G с наименьшим общим весом, исключая минимальное остовное дерево.
- ♥(б) Описать и проанализировать эффективный алгоритм вычисления при заданном взвешенном ненаправленном графе G и целом числе k количество k остовных деревьев графа G с наименьшим весом.
7. Граф $G = (V, E)$ является плотным (dense), если $E = \Theta(V^2)$. Описать модификацию алгоритма минимального остовного дерева Ярника, которая выполняется за время $O(V^2)$ (независимо от E), когда исходный граф является плотным, с использованием только элементарных структур данных, в частности без использования фибоначиевых куч. Этот вариант алгоритма Ярника впервые был описан Эдсгером Дейкстрой (Edsger Dijkstra) в 1958 г.
8. Алгоритмы минимального остовного дерева часто формулируются с использованием операции, называемой *стягиванием ребра* (edge contraction). Для стягивания ребра uv мы вставляем новый узел, перенаправляем любое ребро, смежное с u или v (исключая uv), в этот новый узел, затем удаляем u и v . После стягивания может появиться несколько параллельных ребер между новым узлом и другими узлами графа, поэтому мы удаляем все такие ребра, кроме самого легкого, между любыми двумя узлами.

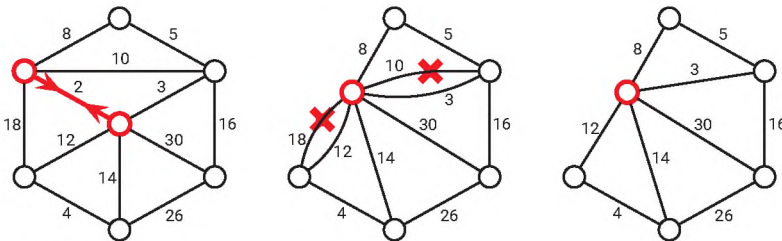


Рис. 7.7. Стягивание ребра и удаление избыточных параллельных ребер

Все три классических алгоритма минимального остовного дерева, описанные в этой главе, можно определенно выразить в терминах стягивания, как показано ниже. Все три алгоритма начинаются с создания чистой копии G' исходного графа G , а затем повторно выполняется операция стягивания надежных ребер в G' — минимальное остовное дерево состоит из этих стянутых ребер.

- **Возвѣтка:** пометить самое легкое ребро, исходящее из каждой вершины, выполнить операцию стягивания всех помеченных ребер, затем выполнить рекурсию.

- Jarník: повторно выполнять операцию стягивания самого легкого ребра, смежного с некоторой фиксированной корневой вершиной.
 - Kruskal: повторно выполнять операцию стягивания самого легкого ребра в графе.
- (a) Описать алгоритм выполнения единственного прохода алгоритма Борувки со стягиванием за время $O(V + 1)$. Исходный граф представлен в виде списка смежных вершин.
- (b) Рассмотреть алгоритм, который сначала выполняет k проходов алгоритма Борувки со стягиванием, а затем выполняет алгоритм Ярника (с использованием фибоначчиевой кучи) на полученном стянутом графе.
- i. Каково время выполнения этого смешанного алгоритма как функция от V, E и k ?
 - ii. Для каких значений k это время выполнения минимизировано? Каково итоговое время выполнения?
- (c) Семейство графов называется правильным (nice), если оно обладает следующими свойствами:
- стягивание ребра правильного графа порождает другой правильный граф;
 - каждый правильный граф с V вершинами содержит только $O(V)$ ребер.

Например, плоские графы – графы, которые можно изобразить на плоскости без пересекающихся ребер, – являются правильными. Стягивание любого ребра плоского графа создает меньший плоский граф, а формула Эйлера предполагает, что каждый плоский граф с V вершинами содержит не более $3V - 6$ ребер.

Доказать, что алгоритм Борувки со стягиванием вычисляет минимальное остовное дерево любого правильного графа за время $O(V)$.

9. Рассмотрим путь между двумя вершинами s и t в ненаправленном взвешенном графе G . Шириной (width) этого пути является минимальный вес каждого ребра в пути. Узкое место (bottleneck distance) между s и t – это ширина самого широкого пути из s в t . (Если не существует путей из s в t , то узким местом является $-\infty$. С другой стороны, узким местом в пути из вершины s в саму себя является ∞ .)
- (a) Доказать, что максимальное остовное дерево графа G содержит самые широкие пути между каждой парой вершин.
- (b) Описать алгоритм решения следующей задачи за время $O(V + E)$: задан ненаправленный взвешенный граф G , две вершины s и t и вес W . Узкое место между s и t не превышает W ?

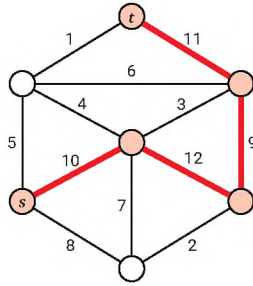


Рис. 7.8. Узкое место между s и t равно 9

(с) Предположим, что B – узкое место между s и t .

- i. Доказать, что удаление любого ребра с весом, меньшим B , не изменяет узкое место между s и t .
- ii. Доказать, что стягивание любого ребра с весом, большим B , не изменяет узкое место между s и t . (Если операция стягивания создает параллельные ребра, то нужно удалить их, за исключением самого тяжелого ребра между каждой парой узлов.)

♥(d) Описать алгоритм вычисления пути с минимальным узким местом между s и t за время $O(V + E)$. [Подсказка: начните с поиска ребра со средним весом в графе G .]

10. Алгоритм Борувки можно переформулировать для использования стандартной структуры данных непересекающихся множеств для идентификации надежных ребер, как и алгоритм Крускала, вместо явного подсчета и пометки компонент растущего остовного леса F на каждой итерации.

В этом варианте каждая компонента леса F представлена деревом *up-tree*, каждая вершина v хранит указатель на своего родителя $\text{parent}(v)$ или на саму себя, если v является корнем этого дерева *up-tree*. Подпрограмма $\text{Find}(v)$ возвращает корень дерева *up-tree* вершины v , но также применяет сжатие пути, переписывая все указатели на родителей из v в корень, чтобы они указывали прямо на корень, для ускорения будущих операций Find .⁷ Подпрограмма Union объединяет два дерева *up-tree* в одно, преобразуя один из двух корневых узлов в родителя другого.⁸

```
Find(v):
  if parent(v) = v
    return v
  else
     $\bar{v} \leftarrow \text{Find}(\text{parent}(v))$ 
    parent(v)  $\leftarrow \bar{v}$ 
    return  $\bar{v}$ 
```

```
Union(u, v):
   $\bar{u} \leftarrow \text{Find}(u)$ 
   $\bar{v} \leftarrow \text{Find}(v)$ 
  either
    parent( $\bar{u}$ )  $\leftarrow \bar{v}$ 
  or
    parent( $\bar{v}$ )  $\leftarrow \bar{u}$ 
```

⁷ Сжатие пути – это одна из форм мемоизации.

⁸ Обычно Union реализуется более тщательно, чтобы гарантировать, что корень большего или старшего дерева *up-tree* не изменялся, но эти подробности здесь не имеют значения.

В модифицированной версии алгоритма Борувки в дополнение к указателям на родителей корневая вершина $\bar{v}$ каждой компоненты леса F поддерживает ребро $\text{safe}(\bar{v})$, которое (в конце FindSafeEdges) является самым легким ребром с одной конечной точкой в каждой компоненте.

```

FindSafeEdges(V, E):
  for each vertex  $v \in V$ 
     $\text{safe}(v) \leftarrow \text{Null}$ 
  found  $\leftarrow \text{False}$ 
  for each edge  $uv \in E$ 
     $\bar{u} \leftarrow \text{Find}(u)$ 
     $\bar{v} \leftarrow \text{Find}(v)$ 
    if  $\bar{u} \neq \bar{v}$ 
      if  $\text{safe}(\bar{u}) = \text{Null}$  or  $w(uv) < w(\text{safe}(\bar{u}))$ 
         $\text{safe}(\bar{u}) \leftarrow uv$ 
      if  $\text{safe}(\bar{v}) = \text{Null}$  or  $w(uv) < w(\text{safe}(\bar{v}))$ 
         $\text{safe}(\bar{v}) \leftarrow uv$ 
    found  $\leftarrow \text{True}$ 
  return found

```

```

AddSafeEdges(V, E, F):
  for each vertex  $v \in V$ 
    if  $\text{safe}(v) \neq \text{Null}$ 
       $xy \leftarrow \text{safe}(v)$ 
      if  $\text{Find}(x) \neq \text{Find}(y)$ 
        Union( $x, y$ )
        add  $xy$  to  $F$ 

```

```

Borůvka(V, E):
   $F = \emptyset$ 
  for each vertex  $v \in V$ 
     $\text{parent}(v) \leftarrow v$ 
  while FindSafeEdges(V, E)
    AddSafeEdges(V, E, F)
  return F

```

Доказать, что каждый вызов FindSafeEdges и AddSafeEdges требует только времени $O(E)$. [Подсказка: какова глубина деревьев up-tree , когда завершается FindSafeEdges?] Из этого следует, что приведенный вариант алгоритма Borůvka также выполняется за время $O(E \log V)$.

Глава 8

Кратчайшие пути

Я изучаю Библию, собирая яблоки. Сначала я встряхиваю все дерево, чтобы упало самое спелое. Затем я взбираюсь на дерево и встряхиваю каждую толстую ветку, затем каждую ветку, затем каждую тонкую веточку, а потом заглядываю под каждый лист.

— приписывается Мартину Лютеру (Martin Luther) (около 1500 г.)

Жизнь — это развитие, и чем дальше мы идем, тем большую истину мы можем постичь. Понимание того, что находится у нас на пороге, — лучшая подготовка к пониманию того, что находится за его пределами.

— приписывается Гипатии Александрийской (Hypatia of Alexandria) (около 400 г.)
в книге Элберта Хаббарда (Elbert Hubbard) «Маленькие путешествия дома великих учителей» («Little Journeys to the Homes of Great Teachers») (1908 г.)

Ваш разум ответит на большинство вопросов, если вы научитесь расслабляться и ждать ответа. Вот в электронный мозг вы вводите, к примеру, вопрос, а потом сидите себе и ждете...

— Эдвард Форест Мур (Edward F. Moore),
«Кратчайший путь через лабиринт» («The Shortest Path Through a Maze») (1959 г.)

Предположим, что задан взвешенный направленный граф $G = (V, E, w)$ с двумя особыми вершинами, и мы должны найти кратчайший путь из вершины-источника s в целевую вершину t . То есть нужно найти направленный путь P , начинающийся в s и заканчивающийся в t , который минимизирует функцию

$$w(P) := \sum_{u \rightarrow v \in P} w(u \rightarrow v).$$

Например, если мне нужно ответить на вопрос «Каков самый быстрый путь для поездки на автомобиле от моей старой квартиры в Шампейн (Иллинойс) до старой квартиры моей жены в Коламбусе (Огайо)?», то я могу воспользоваться графом, вершинами которого являются города, ребрами —

дороги, веса представляют время проезда, s – Шампейн, а t – Коламбус.¹ Граф является направленным, потому что времена проезда по одной и той же дороге могут отличаться для различных направлений. (Когда-то на шоссе I-70 был участок, контролируемый радаром замера скорости, прямо к востоку от границы Индиана/Огайо, но только для трафика, направленного на восток.)

8.1. Деревья кратчайшего пути

Почти каждый известный алгоритм вычисления кратчайших путей из одной вершины в другую в действительности решает (по большей части) следующую более общую задачу поиска кратчайшего пути из одной вершины графа во все остальные (single source shortest path – SSSP): найти кратчайшие пути из вершины-источника s в каждую другую вершину того же графа. Эта задача обычно решается методом поиска дерева кратчайшего пути (shortest path tree) с корнем в s , которое содержит все требуемые кратчайшие пути.

Нетрудно заметить, что если кратчайшие пути не повторяются, то они образуют дерево, потому что любой отрезок кратчайшего пути сам по себе является кратчайшим путем. Если существует несколько кратчайших путей к одним и тем же вершинам, то мы всегда можем выбрать один кратчайший путь к каждой вершине, так что объединение этих путей является деревом. Если существуют кратчайшие пути из s в две вершины u и v , которые расходятся, потом сходятся, затем снова расходятся, то мы можем изменить один из этих путей без изменения его длины так, чтобы эти два пути расходились только один раз.

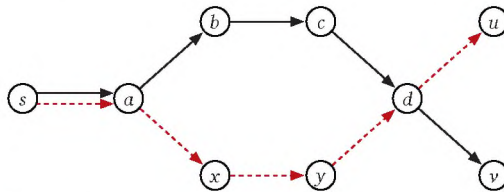


Рис. 8.1. Если $s \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow v$ (сплошные стрелки)

и $s \rightarrow a \rightarrow x \rightarrow y \rightarrow d \rightarrow u$ (штриховые стрелки) являются кратчайшими путями, то $s \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow u$ (по верхнему краю) также является кратчайшим путем

Деревья кратчайшего пути и минимальные остовные деревья представляют собой весьма различные создания, хотя оба являются оптимальными остовными деревьями. Деревья кратчайшего пути имеют корень и направленность, минимальные остовные деревья корня не имеют и являются ненаправленными. Деревья кратчайшего пути наиболее естественно

¹ На запад к церкви, на север по проспекту, на восток по шоссе I-74, на юг по шоссе I-465, на восток по скоростной автомагистрали к аэропорту, на север по шоссе I-65, на восток по шоссе I-70, на север к Грандвью, на восток по дороге № 5, на север к Оулентенджи Ривер, на восток к Додриджу, на север к Хай, на запад к Келсо, на юг к Нейл. В зависимости от дорожного трафика. Сейчас мы живем в Эрбана (Иллинойс).

определяются для направленных графов, минимальные остовные деревья наиболее естественно определяются для ненаправленных графов. Если веса ребер различны, то существует только одно минимальное остовное дерево, но каждая вершина-источник предполагает отличное от других дерево кратчайшего пути, а кроме того, для каждого дерева кратчайшего пути возможно использование различных множеств ребер из минимального остовного дерева.

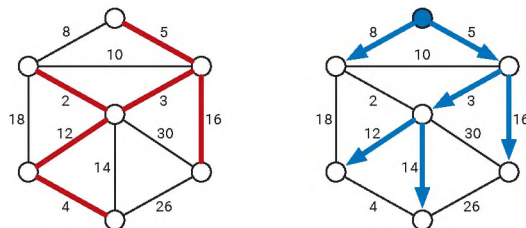


Рис. 8.2. Минимальное остовное дерево и дерево кратчайшего пути для одного и того же ненаправленного графа

♥ 8.2. Отрицательные ребра

Для большинства задач поиска кратчайшего пути, в которых веса ребер соответствуют расстоянию, длине или времени, естественно предположить, что все веса ребер являются неотрицательными или даже положительными числами. Но для многих приложений алгоритмов поиска кратчайшего пути вполне естественно рассмотрение ребер с отрицательными весами. Например, вес ребра может представлять стоимость перемещения из одной вершины в другую, поэтому ребра с отрицательным весом представляют перемещения с отрицательной стоимостью или, что равнозначно, перемещения, которые приносят прибыль.

Отрицательные ребра — источник постоянных неприятностей в большинстве задач поиска кратчайшего пути, потому что при наличии отрицательного цикла может подразумеваться, что кратчайшие пути, возможно, не вполне точно определены. Точнее говоря, кратчайший путь из s в t существует, если и только если существует как минимум один путь из s в t , но не существует пути из s в t , соприкасающийся с отрицательным циклом. Для любого пути из s в t , соприкасающегося с отрицательным циклом, существует более короткий путь из s в t , который проходит по этому циклу еще один раз.² Таким образом, если хотя бы один путь из s в t соприкасается с отрицательным циклом, то кратчайшего пути из s в t не существует.

² Технически здесь мы должны обсуждать кратчайшие блуждания (shortest walks), а не кратчайшие пути, но неправильное использование терминологии — это стандарт. Если s может достичь t , то обязательно должен существовать кратчайший простой путь из s в t , это лишь NP-трудная задача вычисления (при наличии отрицательных циклов) простой редукцией задачи поиска гамильтонова пути. С другой стороны, если существует кратчайшее блуждание из s в t , то это блуждание непременно должно быть простым путем, следовательно, — кратчайшим простым путем из s в t . О, черт!

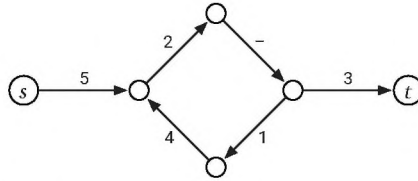


Рис. 8.3. Здесь нет кратчайшего пути из s в t

Кроме того, поскольку мы должны учитывать ребра с отрицательными весами, в этой главе явно рассматриваются только направленные графы. Все описываемые здесь алгоритмы также работают и для ненаправленных графов с весьма тривиальными изменениями, если и только если отрицательные ребра запрещены. Корректная обработка отрицательных ребер в ненаправленных графах – это значительно более тонкая работа. Мы не можем просто заменить каждое ненаправленное ребро парой направленных ребер, потому что это преобразовало бы любое отрицательное ребро в короткий отрицательный цикл. Отрезки ненаправленного кратчайшего пути, содержащие отрицательное ребро, не обязательно являются кратчайшими путями, следовательно, множество всех ненаправленных кратчайших путей из одной вершины-источника может не определять дерево, даже если эти кратчайшие пути являются неповторяющимися.

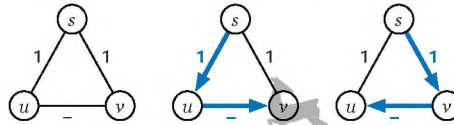


Рис. 8.4. Ненаправленный граф, в котором кратчайшие пути из s являются неповторяющимися, но не определяют дерево

Полный курс лечения ненаправленных графов с отрицательными ребрами выходит за пределы тематики этой книги. Я только лишь отмечу для тех, кто всегда ищет ответы в Google, что единственный кратчайший путь в ненаправленном графе с отрицательными ребрами можно вычислить за время $O(VE + V^2 \log V)$, применив редукцию к алгоритму построения паросочетания с максимальным весом.

8.3. Единственный алгоритм SSSP

Как и для алгоритмов обхода графа и минимальных остовных деревьев, многие разнообразные алгоритмы SSSP можно описать как особые случаи одного обобщенного алгоритма, впервые предложенного Лестером Фордом (Lester Ford) в 1956 г. и независимо от него описанного Джорджем Данцигом (George Dantzig) в 1957 г.³, а также Джорджем Минти (George Minty)

³ В частности, Данциг показал, что задачу поиска кратчайшего пути можно переформулировать как задачу линейного программирования, а затем описал интерпретацию собственного симплекс-метода в терминах исходного графа. Описание Данцига (по существу) равнозначно релаксационной стратегии Форда.

в 1958 г. Каждая вершина v в графе хранит два значения, которые (индуктивно) описывают пробный кратчайший путь из s в v :

- $dist(v)$ — длина пробного кратчайшего пути $s \rightsquigarrow v$ или ∞ , если такого пути нет;
- $pred(v)$ — предок вершины v в пробном кратчайшем пути $s \rightsquigarrow v$ или $Null$, если такой вершины нет.

Указатели на предков автоматически определяют дерево пробного кратчайшего пути с корнем в s , эти указатели в точности те же, что и указатели на родителей в нашем обобщенном алгоритме обхода графа. В начале алгоритма мы инициализируем расстояния и предков, как показано ниже:

```
InitSSSP(s):
  dist(s) ← 0
  pred(s) ← Null
  for all vertices v ≠ s
    dist(v) ← ∞
    pred(v) ← Null
```

Во время выполнения алгоритма ребро $u \rightarrow v$ является напряженным (tense), если $dist(u) + w(u \rightarrow v) < dist(v)$. Если ребро $u \rightarrow v$ напряженное, то пробный кратчайший путь $s \rightsquigarrow v$ явно некорректный, потому что путь $s \rightsquigarrow u \rightarrow v$ короче. Мы можем скорректировать (или по меньшей мере улучшить) это очевидное преувеличение, уменьшив напряжение ребра, как показано ниже:

```
Relax(u→v):
  dist(v) ← dist(u) + w(u→v)
  pred(v) ← u
```

Теперь, когда все подготовлено, обобщенный алгоритм Форда приобретает вид простого однострочного описания:

**Повторно ослаблять напряженные ребра до тех пор,
пока не останется напряженных ребер.**

```
FordSSSP(s):
  InitSSSP(s)
  while there is at least one tense edge
    Relax any tense edge
```

<<пока существует хотя бы одно напряженное ребро>>
<<ослабить напряжение любого напряженного ребра>>

Если FordSSSP в итоге завершается (потому что больше не существует напряженных ребер), то указатели на предков корректно определяют де-

рево кратчайшего пути, и каждое значение $dist(v)$ — это реальное расстояние кратчайшего пути из s в v . В частности, если s не может достичь v , то $dist(v) = \infty$, а если любой отрицательный цикл недостижим из s , то алгоритм никогда не завершается.

Корректность обобщенного алгоритма Форда следует из приведенной ниже последовательности простых утверждений.

1. В любой момент выполнения алгоритма для каждой вершины v расстояние $dist(v)$ равно либо ∞ , либо длине блуждания из s в v . Это утверждение можно доказать по индукции для числа операций ослабления напряжения.
2. Если граф не содержит отрицательных циклов, то $dist(v)$ равно либо ∞ , либо длине некоторого простого пути из s в v . В частности, если $dist(v)$ равно длине блуждания из s в v , содержащего направленный цикл, то этот цикл обязательно должен иметь отрицательную длину. Это утверждение предполагает, что если G не содержит отрицательных циклов, то алгоритм ослабления напряжения в итоге останавливается, потому что существует только конечное число простых путей в G .
3. Если ни одно из ребер в G не является напряженным, то для каждой вершины v расстояние $dist(v)$ равно длине пути предка $s \rightarrow \dots \rightarrow pred(pred(v)) \rightarrow pred(v) \rightarrow v$. В частности, если v нарушает это условие, а его предок $pred(v)$ не нарушает, то ребро $pred(v) \rightarrow v$ является напряженным.
4. Если ни одно из ребер в G не является напряженным, то для каждой вершины v путь из ребер предков $s \rightarrow \dots \rightarrow pred(pred(v)) \rightarrow pred(v) \rightarrow v$ в действительности является кратчайшим путем из s в v . В частности, если v нарушает это условие, а его предок u в некотором кратчайшем пути не нарушает, то ребро $u \rightarrow v$ является напряженным. Это утверждение также предполагает, что если G содержит отрицательный цикл, то некоторое ребро всегда является напряженным, поэтому обобщенный алгоритм никогда не остановится.

До сих пор я ничего не говорил о том, как находить напряженные ребра, или к какому (каким) из напряженных ребер применять операцию ослабления напряжения, если таковых более одного. Как и для поиска в любом направлении, существует несколько различных вариантов обобщенного алгоритма ослабления напряжения Форда. Но в отличие от поиска в любом направлении эффективность и корректность каждой стратегии поиска зависит от структуры исходного графа.

В остальной части этой главы рассматривается четыре наиболее общих варианта алгоритма Форда, каждый из которых представляет собой наилучший вариант выбора для различных классов исходных графов. Я оставляю все прочие подробности общего доказательства корректности в качестве упражнений, а вместо этого приведу (более информативные,

самодостаточные) доказательства корректности для каждого из этих четырех конкретных алгоритмов.

8.4. Невзвешенные графы: поиск в ширину

В простейшем особом случае задачи поиска кратчайшего пути все ребра имеют вес 1, а длина пути – это просто число ребер. Этот особый случай можно решить одной из разновидностей нашего алгоритма обхода графа под названием поиск в ширину (breadth-first search). Алгоритм поиска в ширину часто приписывают Эдварду Муру (Edward Moore), который описал его в 1957 г. (как «алгоритм А») как первый опубликованный метод поиска кратчайшего пути в лабиринте⁴. Особенно в контексте схем разводки СБИС и планирования пути робота алгоритм поиска в ширину иногда приписывают Цзинь Ян Ли (Chin Yang Lee), который описал несколько приложений «алгоритма А» Мура (с корректным указанием авторских прав Мура) в 1961 г. Но в 1945 г., более чем за десять лет до исследования лабиринтов Муром Конрад Цузе (Konrad Zuse) описал реализацию поиска в ширину как метод подсчета и пометки компонент несвязного графа.⁶

Алгоритм поиска в ширину поддерживает очередь типа «первым вошел, первым вышел» (FIFO) для вершин, которая изначально содержит только вершину-источник s . На каждой итерации алгоритм извлекает (Pull) вершину u из головы очереди и проверяет каждое исходящее из нее ребро $u \rightarrow v$. Когда алгоритм обнаруживает исходящее напряженное ребро $u \rightarrow v$, он ослабляет напряжение этого ребра и помещает (Push) вершину v в очередь. Алгоритм завершается, когда очередь становится пустой.

⁴ Мур был мотивирован слабостью робота «Theseus» (Тесей), созданного Клодом Шенноном (Claude Shannon) для решения лабиринтов. Шеннон спроектировал и построил робота в 1950 г. (Тесей использовал мемоизированную версию поиска в глубину, реализованную с помощью электромеханических реле — почти наверняка это была первая реализация поиска в глубину в графах.) Мур говорил: «Когда эта машина использовалась в лабиринте, для которого существовало более одного решения, посетитель спрашивал, почему она не создана так, чтобы всегда находить кратчайший путь. Шеннон и я, каждый по своему, пытались найти экономные методы такого решения, найденного машиной. Шеннон нашел несколько методов, подходящих для аналоговых вычислений⁵, а я получил эти алгоритмы.

⁵ Аналоговые методы вычисления кратчайших путей в лабиринтах были предложены с использованием шарикоподшипников, потока жидкости/плазмы, волновых химических реакций, хемотропизма, сетей резисторов, электрических схем с LED-индикаторами, сетей мемристоров, тлеющих разрядов в микрожидкостных чипах, растений в процессе роста, слизистых грибов (плесени), амёб, муравьев, пчел, круглых червей и туристов.

⁶ Конрад Цузе был одним из первооткрывателей ИТ и ВТ, он спроектировал и сконструировал свой первый программируемый компьютер (позже названный Z1) в конце 1930-х гг. из металлических полос и стержней в гостиной своих родителей. Компьютер Z1 и его оригинальные чертежи были уничтожены при налете британской авиации в 1944 г. В кандидатской диссертации Цузе в 1945 г. описывает самый первый язык программирования высокого уровня, названный Планкалькюль (Planckalkül). Первый полным примером программы на Планкалькюль в диссертации Цузе была реализация поиска в ширину для подсчета компонент вместе с описанием на псевдокоде и пошаговой иллюстрацией трассировки выполнения алгоритма на несвязном графе с восемью вершинами. Из-за падения нацистского режима Цузе не смог представить и защитить свою кандидатскую диссертацию, и Планкалькюль оставался неопубликованным до 1972 г. Первый компилятор языка Планкалькюль был в полной мере реализован в 1975 г. Йохимом Хохманном (Joachim Hohmann).


```

BFS(s):
  InitSSSP(s)
  Push(s)
  while the queue is not empty    «пока очередь не пуста»
    u ← Pull()
    for all edges u → v
      if dist(v) > dist(u) + 1    «если ребро u → v напряженное»
        dist(v) ← dist(u) + 1
        pred(v) ← u             «ослабить напряжение u → v»

```

Поиск в ширину несколько проще анализировать, если разделить процесс его выполнения на этапы, введя воображаемый маркер (token). Перед извлечением (Pull) любых вершин мы помещаем (Push) маркер в очередь. Текущий этап завершается, когда мы извлекаем (Pull) этот маркер из очереди, а следующий этап начинается, когда мы снова помещаем (Push) маркер в очередь. Таким образом, первый этап состоит исключительно из просмотра вершины-источника s . Алгоритм завершается, когда очередь содержит только маркер. Этот измененный алгоритм показан на рис. 8.5, а на рис. 8.6 показан пример его работы. Еще раз особо отмечу, что эти изменения внесены исключительно для удобства анализа, так что с маркером или без него алгоритм помещает (Push) и извлекает (Pull) вершины в одном и том же порядке, просматривает ребра в одинаковом порядке и выводит в точности те же расстояния и тех же предков.

```

BFSWithToken(s):
  InitSSSP(s)
  Push(s)
  Push(※)                      «начало первого этапа»
  while the queue contains at least one vertex
    u ← Pull( )
    if u = ※
      Push(※)                  «начало следующего этапа»
    else
      for all edges u → v
        if dist(v) > dist(u) + 1 «если ребро u → v напряженное»
          dist(v) ← dist(u) + 1 «ослабить напряжение u → v»
          pred(v) ← u
          Push(v)

```

Рис. 8.5. Поиск в ширину с маркером окончания этапа (※).

Строки, выделенные полужирным красным шрифтом, предназначены только для анализа

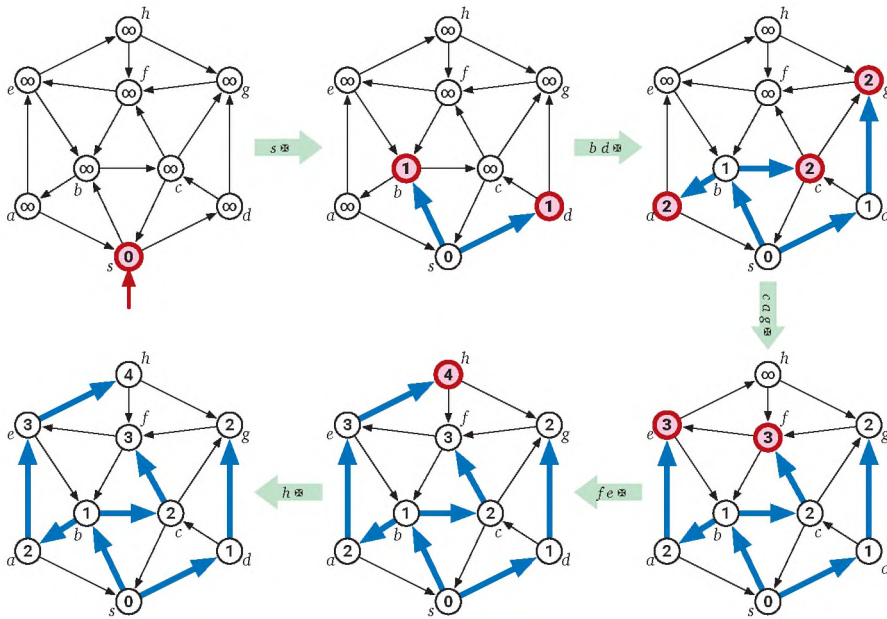


Рис. 8.6. Полная схема выполнения алгоритма поиска в ширину в направленном графе. Вершины извлекаются из очереди в порядке $s \neq b d \neq c a g \neq f e \neq h \neq \neq$, где $\neq$ — маркер конца этапа. Выделенные толстой линией вершины находятся в очереди в конце каждого этапа. Выделенные утолщенными линиями ребра описывают рост дерева кратчайшего пути

Позволю себе особо отметить, что в приведенной ниже лемме $dist(v)$ — это просто переменная, поддерживаемая этим алгоритмом. Пока $dist(v)$ интуитивно представляет пробное расстояние кратчайшего пути, мы не можем (пока) предполагать, что $dist(v)$ действительно равно истинному расстоянию кратчайшего пути из s в v . Не беспокойтесь, мы достигнем цели.

Лемма 8.1. Для каждого целого числа $i \geq 0$ и каждой вершины v в конце i -го этапа либо $dist(v) = \infty$, либо $dist(v) \leq i$, и вершина v находится в очереди, если и только если $dist(v) = i$.

Доказательство. Доказательство выполняется методом индукции по i . Базовый случай $i = 0$ очевиден: в начале первого этапа («в конце нулевого этапа») очередь содержит только начальную вершину s и маркер $\neq$, а $InitSSSP$ просто устанавливает $dist(s) \leftarrow 0$ и $dist(v) \leftarrow \infty$ для всех $v \neq s$.

Теперь определим целое число $i > 0$. По индуктивному предположению в начале i -го этапа очередь содержит все вершины u с $dist(u) = i - 1$ после маркера $\neq$. Другими словами, очередь выглядит так:

$\rightarrow \neq i-1 \ i-1 \ \dots \ i-1 \rightarrow$

Таким образом, прежде чем извлечь (Pull) маркер $\star$ из очереди, завершая i -й этап, мы извлекаем (Pull) каждую вершину u с $\text{dist}(u) = i - 1$. Для каждой такой вершины u мы рассматриваем каждое исходящее ребро $u \rightarrow v$. Если ребро $u \rightarrow v$ напряженное, мы устанавливаем $\text{dist}(v) \leftarrow \text{dist}(u) + 1$, так что $\text{dist}(v) = i$, а затем немедленно помещаем (Push) v в очередь. Только эти операции присваивания меток расстояниям выполняются на i -ом этапе. Следовательно, по индукции на протяжении всего i -го этапа очередь содержит некоторые вершины с меткой расстояния $i - 1$ после маркера, следующего за несколькими вершинами с меткой расстояния i :

$$\rightarrow i \dots i \star i - 1 \dots i - 1 \rightarrow$$

По существу, непосредственно перед завершением i -го этапа очередь содержит маркер, следующий за несколькими вершинами с меткой расстояния i .

$$\rightarrow i i \dots i \star \rightarrow$$

Кроме того, вершина v появляется в этой завершающей очереди, если и только если значение $\text{dist}(v)$ было изменено во время i -го этапа. Следовательно, в конце i -го этапа очередь содержит только вершину v с $\text{dist}(v) = i$. $\square$

Лемма 8.1 предполагает, что в основном теле алгоритма BFS выполняется присваивание меток расстояний в неубывающем порядке. С другой стороны, метка расстояния $\text{dist}(v)$ каждой вершины v никогда не увеличивается. Из этого следует, что для каждой вершины v строка $\text{dist}(v) \leftarrow \text{dist}(u) + 1$ выполняется не более одного раза во время этапа $\text{dist}(v)$. Аналогично:

- каждый указатель на предка $\text{pred}(v)$ изменяется не более одного раза во время этапа $\text{dist}(v)$;
- каждая вершина v помещается (Push) в очередь не более одного раза во время этапа $\text{dist}(v)$;
- каждая вершина u извлекается (Pull) из очереди не более одного раза во время этапа $\text{dist}(u) + 1$;
- для каждого ребра $u \rightarrow v$ сравнение $\text{dist}(v) > \text{dist}(u) + 1$ выполняется не более одного раза во время этапа $\text{dist}(u) + 1$.

Все вместе эти высказывания предполагают, что поиск в ширину выполняется за время $O(V + E)$. Интуитивно мы можем считать вершины в очереди «фронтом волны», монотонно распространяющейся в направлении от вершины-источника s , проходя каждую вершину и ребро графа не более одного раза. Эта аналогия распространения фронта волны была предложена Цзинь Ян Ли (Chin Yang Lee) уже в 1961 г. под впечатлением визуализаций, сгенерированных его реализацией алгоритма А Мура.

Эти высказывания также предполагают, что мы можем заменить условие $\text{if } \text{dist}(v) > \text{dist}(u) + 1$ на (вероятно) более простую проверку $\text{if } \text{dist}(v) = \infty$. Тогда расстояния играют ту же роль, что и метки, поддерживаемые другими алгоритмами обхода графа, которые гарантируют, что каждая вершина посещается только один раз. Более конкретно: вершина «помечена», если и только если ее метка расстояния конечна.

Но нам все еще нужно доказать, что конечные метки расстояния являются корректными.

Теорема 8.2. Когда алгоритм поиска в ширину завершается, $\text{dist}(v)$ – это длина кратчайшего пути в графе G из s в v для каждой вершины v .

Доказательство. Определим произвольную вершину v и рассмотрим произвольный путь $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_\ell$ в графе G , где $v_0 = s$ и $v_\ell = v$. Я утверждаю, что $\text{dist}(v_j) \leq j$ для каждого индекса j , в частности $\text{dist}(v) \leq \ell$. Мы можем доказать это утверждение методом индукции по j , как показано ниже:

- очевидно, что $\text{dist}(v_0) = \text{dist}(s) = 0$;
- для любого индекса $j > 0$ по индуктивному предположению $\text{dist}(v_{j-1}) \leq j - 1$. Сразу после извлечения (Pull) вершины v_{j-1} из очереди либо уже $\text{dist}(v_j) \leq \text{dist}(v_{j-1}) + 1$, либо мы устанавливаем $\text{dist}(v_j) \rightarrow \text{dist}(v_{j-1}) + 1$. В любом случае получаем $\text{dist}(v_j) \leq \text{dist}(v_{j-1}) + 1 \leq j$.

Мы только что доказали, что $\text{dist}(v)$ не больше длины произвольного кратчайшего пути из s в v . Из этого следует, что $\text{dist}(v)$ не больше длины кратчайшего пути из s в v .

Простое доказательство по индукции предполагает, что $\text{dist}(v)$ является длиной пути предка $s \rightarrow \dots \rightarrow \text{pred}(\text{pred}(v)) \rightarrow \text{pred}(v) \rightarrow v$, следовательно, это расстояние обязательно должно быть кратчайшим путем. $\square$

8.5. Направленный ациклический граф: поиск в глубину

Кратчайшие пути также легко вычисляются в направленных ациклических графах, даже если их ребра взвешенные, и в частности, даже если некоторые ребра имеют отрицательный вес. (Не следует беспокоиться об отрицательных циклах, потому что по определению направленные ациклические графы не содержат никаких циклов) В действительности это абсолютно стандартный алгоритм динамического программирования.

Пусть G – направленный граф с взвешенными ребрами и пусть s – фиксированная начальная вершина. Для любой вершины v пусть $\text{dist}(v)$ обозначает длину кратчайшего пути в G из s в v . Этой функции соответствует следующее рекуррентное выражение:

$$\text{dist}(v) = \begin{cases} 0 & \text{, если } v = s; \\ \min_{u \rightarrow v} (\text{dist}(u) + w(u \rightarrow v)) & \text{иначе.} \end{cases}$$

В действительности это тождество соблюдается для всех направленных графов, но оно является рекуррентным только для направленных ациклических графов. Если бы исходный граф G содержал цикл, то рекурсивное вычисление этой функции привело бы к бесконечному циклу, но поскольку G – НАГ, каждый рекурсивный вызов посещает предыдущие вершины в топологическом порядке.

Граф зависимостей для этого рекуррентного выражения является обратным исходному графу G : подзадача $dist(v)$ зависит от $dist(u)$, если и только если $u \rightarrow v$ является ребром в G . Таким образом, мы вычисляем каждое расстояние за время $O(V + E)$, выполняя поиск в глубину в графе, обратном G и рассматривая вершины в обратном порядке обхода. Равнозначно мы можем рассматривать вершины в исходном графе G в топологическом порядке, как показано на рис. 8.7.

```
DagSSSP(s):
  for all vertices v in topological order
    if v = s
      dist(v) ← 0
    else
      dist(v) ← ∞
      for all edges u → v
        if dist(v) > dist(u) + w(uv)  «если ребро u → v напряженное»
          dist(v) ← dist(u) + w(uv)  «ослабить напряжение u → v»
```

Рис. 8.7. Вычисление кратчайших путей в НАГ с использованием динамического программирования

Полученный в результате алгоритм динамического программирования является еще одним примером обобщенного алгоритма Форда ослабления напряжения ребер. Чтобы сделать эту связь еще более ясной, мы можем вынести инициализацию $dist(v)$ за пределы основного цикла и добавить вычисление указателей на предков, как показано на рас. 8.8.

```
DagSSSP(s):
  InitSSSP(s)
  for all vertices v in topological order
    for all edges u → v
      if u → v is tense
        Relax(uv)
```

Рис. 8.8. Вычисление кратчайших путей в НАГ с использованием алгоритма Форда. (Это тот же самый алгоритм)

На рис. 8.9 показана схема работы этого алгоритма.

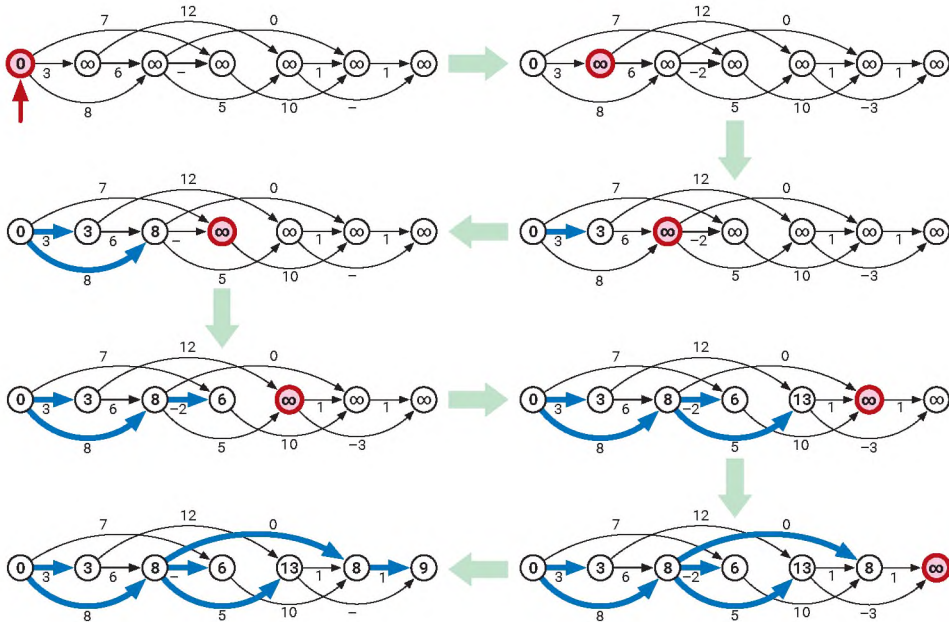


Рис. 8.9. Вычисление кратчайших путей в НАГ с помощью ослабления напряжения входящих ребер в топологическом порядке. На каждой итерации утолщенные ребра обозначают предков, а утолщенные вершины обозначают состояние непосредственно перед просмотром. Сравните с рис. 8.10

Алгоритм DagSSSP отличается от алгоритма поиска в ширину и других вариантов релаксационной стратегии Форда в одном минимальном аспекте. Когда другие алгоритмы поиска кратчайшего пути рассматривают вершину, они пытаются ослабить напряжение каждого из исходящих из нее ребер, интуитивно распространяя фронт волны в направлении от источника, тогда как DagSSSP пытается ослабить напряжение каждого из входящих ребер для каждой вершины, интуитивно растягивая фронт волны вперед.

Но если мы изменяем алгоритм DagSSSP, чтобы ослабить напряжение исходящих ребер вместо входящих, то получим другой алгоритм, который вычисляет кратчайшие пути в направленных ациклических графах за время $O(V + E)$ и в большей степени похож на другие наши алгоритмы поиска кратчайшего пути.

```

PushDagSSSP(s):
  InitSSSP(s)
  for all vertices u in topological order
        <<для всех вершин u в топологическом порядке>>
    for all outgoing edges  $u \rightarrow v$ 
          <<для всех исходящих ребер  $u \rightarrow v$ >>
      if  $u \rightarrow v$  is tense
        Relax( $u \rightarrow v$ )
  
```


На рис. 8.10 показана схема выполнения этого измененного алгоритма на том же графе, что и на рис. 8.9. Корректность алгоритма PushDagSSSP следует непосредственно из корректности обобщенной релаксационной стратегии Форда, но нетрудно доказать корректность напрямую методом индукции для вершин в топологическом порядке.

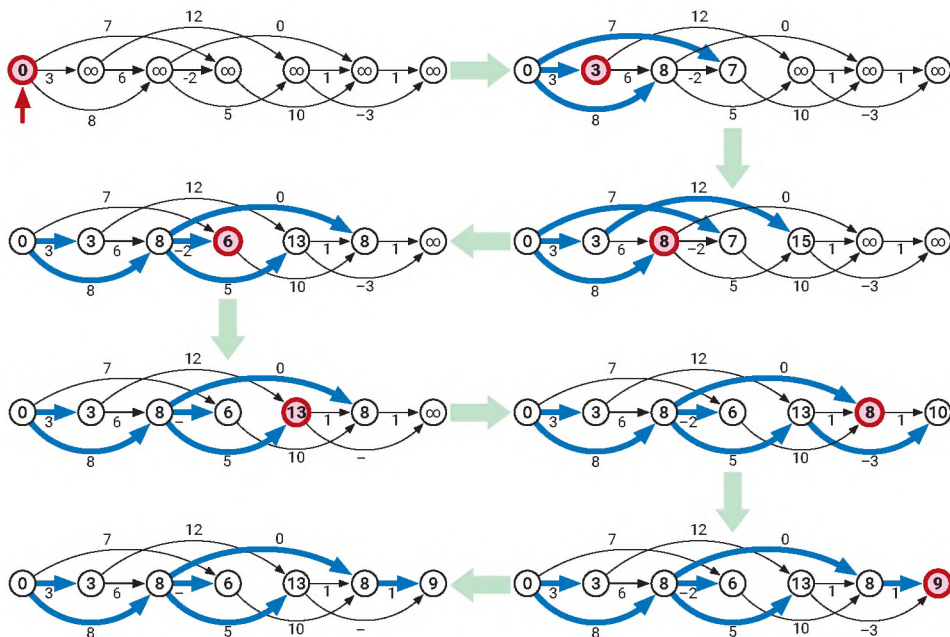


Рис. 8.10. Вычисление кратчайших путей в НАГ с помощью ослабления напряжений исходящих ребер в топологическом порядке. На каждой итерации утолщенные ребра обозначают предков, а утолщенные вершины обозначают состояние непосредственно перед просмотром. Сравните с рис. 8.9

8.6. Поиск по первому наилучшему совпадению: алгоритм Дейкстры

Если в алгоритме поиска в ширину заменить очередь «первым вошел, первым вышел» (FIFO) на очередь с приоритетами, где ключом вершины является ее пробное расстояние $dist(v)$, то получим алгоритм, «опубликованный» впервые в 1957 г. группой исследователей из Кейсовского технологического института (Кливленд, Огайо, США), возглавляемой Майклом Лейзореком (Michael Leyzorek), в годовом отчете по проекту для Исследовательского центра вооружений (Combat Development Department) на испытательном полигоне электронной техники армии США (US Army Electronic Proving Ground). Тот же алгоритм был независимо разработан Эдсгером Дейкстрой (Edsger Dijkstra) в 1956 г. (но не публиковался до 1959 г.), потом

Джорджем Минти (George Minty) чуть ранее 1960 г., затем Питером Уайтингом (Peter Whiting) и Джоном Хиллером (John Hillier) в 1960 г. Почти такой же алгоритм был также описан Джорджем Данцигом (George Dantzig) в 1958 г. Хотя некоторые ранние источники называют его алгоритмом Минти, эта методика сейчас повсеместно известна как алгоритм Дейкстры в полном соответствии с законом Стиглера.⁷ Псевдокод этого алгоритма показан на рис. 8.11.

```

Dijkstra(s):
  InitSSSP(s)
  Insert(s, 0)
  while the priority queue is not empty
    u ← ExtractMin( )
    for all edges u → v
      if u → v is tense
        Relax(u → v)
      if v is in the priority queue
        DecreaseKey(v, dist(v))
      else
        Insert(v, dist(v))

```

Рис. 8.11. Алгоритм Дейкстры

Простое доказательство методом индукции предполагает, что на протяжении всего времени выполнения этого алгоритма ребро $u \rightarrow v$ является напряженным, если и только если вершина u либо находится в очереди с приоритетами, либо является вершиной, которая извлечена (Extract) позже всех (в текущий момент) из очереди с приоритетами. Таким образом, алгоритм Дейкстры является вариантом обобщенной стратегии Форда, который предполагает, что кратчайшие пути вычисляются корректно при отсутствии отрицательных циклов в графе G .

Отсутствие отрицательных ребер

Алгоритм Дейкстры особенно хорошо ведет себя, если исходный граф не содержит ребер с отрицательным весом. При этом условии алгоритм интуитивно распространяет фронт волны в направлении от вершины-источника s , проходя по вершинам в возрастающем порядке их расстояний от s , как и при поиске в ширину. На рис. 8.12 показана схема выполнения этого алгоритма.

⁷ Я буду соблюдать это общепринятое соглашение, несмотря на историческую неточность, отчасти из-за того, что считаю, что вряд ли кто-то захочет читать об «алгоритме Лейзорека–Грея–Джонсона–Лэдьо–Микера–Петри–Сайца–Данцига–Дейкстры–Минти–Уайтинга–Хиллера», и отчасти из-за того, что статьи, которые в действительности не были опубликованы, не считаются.

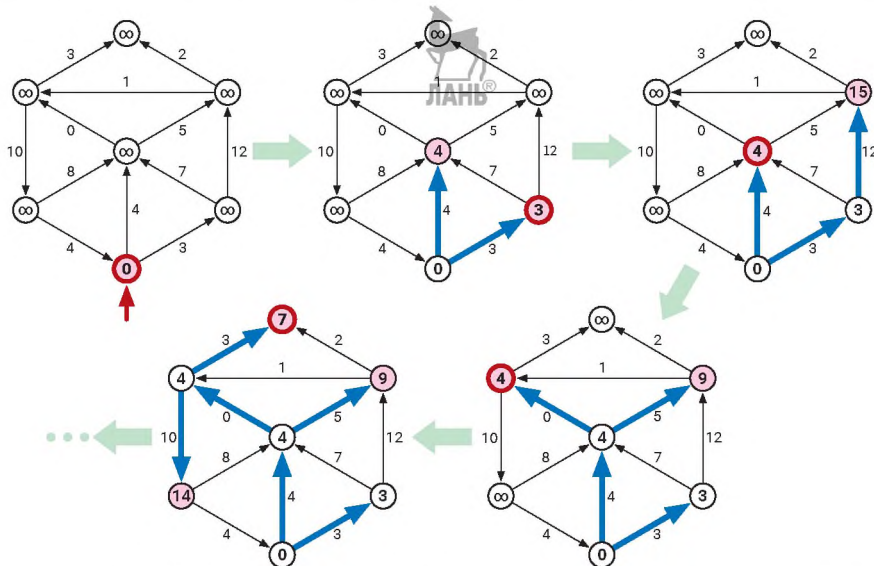


Рис. 8.12. Первые четыре итерации алгоритма Дейкстры на графе без отрицательных ребер. На каждой итерации утолщенные ребра обозначают предков, затененные вершины находятся в очереди с приоритетами, а утолщенные вершины находятся в состоянии непосредственно перед просмотром. Остальные итерации не изменяют ни расстояния, ни дерево кратчайшего пути

Мы можем вывести самодостаточное доказательство корректности алгоритма Дейкстры для этого случая, формализуя интуитивное представление фронта волны. Для каждого целого числа i пусть u_i обозначает вершину, возвращаемую i -м вызовом подпрограммы `ExtractMin`, и пусть d_i – значение $\text{dist}(u_i)$ сразу после этого извлечения (`Extract`). В частности, имеем $u_1 = s$ и $d_1 = 0$. В этот момент мы не можем предполагать, что вершины u_i различны, теоретически, одна и та же вершина может быть извлечена более одного раза.

Лемма 8.3. Если граф G не содержит ребер с отрицательным весом, то для всех $i < j$ мы имеем $d_i \leq d_j$.

Доказательство. Предположим, что граф G не содержит ребер с отрицательным весом. Определим произвольный индекс i . Для доказательства леммы достаточно доказать, что $d_{i+1} \geq d_i$. Необходимо рассмотреть два случая:

- если G содержит ребро $u_i \rightarrow u_{i+1}$ и для этого ребра ослаблено напряжение во время i -й итерации основного цикла, то в конце i -й итерации имеем $\text{dist}(u_{i+1}) = \text{dist}(u_i) + w(u_i \rightarrow u_{i+1}) \geq \text{dist}(u_i)$, потому что веса всех ребер неотрицательные;
- иначе в начале i -й итерации вершина u_{i+1} уже обязательно должна находиться в очереди с приоритетами и непременно должна иметь

приоритет $dist(u_{i+1}) \geq dist(u_i)$, потому что u_i – это вершина, возвращенная процедурой ExtractMin. Кроме того, $dist(u_{i+1})$ не изменяется во время i -й итерации.

В обоих случаях мы приходим к выводу, что $d_{i+1} \geq d_i$. Теперь лемма логически вытекает непосредственно из индукции по i . $\square$

Лемма 8.4. Если граф G не содержит ребер с отрицательным весом, то каждая вершина в G извлекается (Extract) из очереди с приоритетами не более одного раза.

Доказательство. Предположим, что v извлекается (Extract) более одного раза. А именно предположим, что v извлекается на i -й итерации основного цикла, повторно вставляется (Insert) в очередь во время j -й итерации, затем повторно извлекается на k -й итерации при некоторых индексах $i < j < k$. Тогда по форме записи предыдущего доказательства имеем $v = u_i = u_k$.

Метка расстояния $dist(v)$ никогда не увеличивается. Кроме того, $dist(v)$ строго уменьшается во время j -й итерации непосредственно перед повторной вставкой v в очередь. Из этого следует, что $d_i > d_k$. Следовательно, по предыдущей лемме G содержит как минимум одно ребро с отрицательным весом. $\square$

Лемма 8.4 непосредственно предполагает, что каждая вершина просматривается не более одного раза, следовательно, ослабление напряжения каждого ребра выполняется не более одного раза. Но в отличие от поиска в ширину каждая метка расстояния $dist(v)$ может изменяться несколько раз. В первый раз $dist(v)$ изменяется с ∞ , когда мы вставляем (Insert) v в очередь с приоритетами, после этого каждое изменение $dist(v)$ сопровождается вызовом DecreaseKey. После извлечения (Extract) v из очереди с приоритетами ее метка расстояния никогда не изменяется.

Оставшаяся часть доказательства корректности почти идентична поиску в ширину.

Теорема 8.5. Если граф G не содержит ребер с отрицательным весом, при завершении алгоритма Dijkstra $dist(v)$ является длиной кратчайшего пути в G из s в v для каждой вершины v .

Доказательство. Определим произвольную вершину v и рассмотрим произвольный путь $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_\ell$ в графе G , где $v_0 = s$ и $v_\ell = v$. Для любого индекса j пусть L_j обозначает длину отрезка пути $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_j$. Докажем методом индукции, что $dist(v_j) \leq L_j$ для всех j .

- Очевидно, что $dist(v_0) = dist(s) = 0 = L_0$.
- Для любого индекса $j > 0$ по индуктивному предположению $dist(v_{j-1}) \leq L_{j-1}$. Сразу после извлечения (Pull) вершины v_{j-1} из очереди либо уже $dist(v_j) \leq dist(v_{j-1}) + w(v_{j-1} \rightarrow v_j)$, либо мы устанавливаем $dist(v_j) \leftarrow dist(v_{j-1}) + w(v_{j-1} \rightarrow v_j)$.

В обоих случаях получаем:

$$\text{dist}(v_j) \leq \text{dist}(v_{j-1}) + w(v_{j-1} \rightarrow v_j) \leq L_{j-1} + w(v_{j-1} \rightarrow v_j) = L_j.$$

Мы только что доказали, что $\text{dist}(v)$ – это не более, чем длина каждого пути из s в v . Из этого следует, что $\text{dist}(v)$ – это не более, чем длина кратчайшего пути из s в v .

С другой стороны, аналогичное индуктивное доказательство предполагает, что $\text{dist}(v)$ – длина пути предков $s \rightarrow \dots \rightarrow \text{pred}(\text{pred}(v)) \rightarrow \text{pred}(v) \rightarrow v$. □

Остается только связать это со временем выполнения алгоритма. В целом алгоритм Dijkstra выполняет не более E операций DecreaseKey и не более V операций Insert и ExtractMin. Таким образом, если мы реализуем внутреннюю очередь с приоритетами с использованием стандартной двоичной кучи, которая поддерживает выполнение каждой операции за время $O(\log V)$, то алгоритм Dijkstra выполняется **за время $O(E \log V)$** ⁸.

Если нам заранее известно, что исходные графы никогда не будут содержать отрицательных ребер, то можно немного упростить алгоритм Дейкстры, вставив (Insert) каждый индекс в очередь с приоритетами на этапе инициализации, а затем только лишь вызывая DecreaseKey в основном цикле, как показано на рис. 8.13. Это версия алгоритма Дейкстры, представленная в большинстве книг по алгоритмам, в «Википедии» и даже в самой статье Дейкстры. Кроме того, это та версия алгоритма Дейкстры, которую я описал как «поиск по первому наилучшему совпадению» в главе 5.

```
NonnegativeDijkstra(s):
  InitSSSP(s)
  for all vertices v
    Insert(v, dist(v))
  while the priority queue is not empty
    u ← ExtractMin( )
    for all edges u → v
      if u → v is tense
        Relax(uv)
        DecreaseKey(v, dist(v))
```

Рис. 8.13. Слегка упрощенный алгоритм Дейкстры для графов без отрицательных ребер. Отличия от основного алгоритма Дейкстры выделены красным цветом и полужирным шрифтом

⁸ В статьях об алгоритмах поиска кратчайшего пути 1950-х гг. никогда не упоминается очередь с приоритетами. Дейкстра предложил просмотр прямым перебором всех вершин на фронте волны на каждой итерации. Исходный алгоритм Дейкстры выполняется за время $O(V^2)$, что действительно быстрее, чем реализация с двоичной кучей при $E = \Omega(V^2)$. Минти предложил просмотр прямым перебором всех ребер $u \rightarrow v$, таких, что расстояние $\text{dist}(u)$ конечно, но $\text{dist}(v)$ нет, следовательно, исходный алгоритм Минти выполняется за время $O(VE)$. Использование очереди с приоритетами, реализованной как двоичная куча, для получения нелинейного времени выполнения было предложено Дональдом Джонсоном (Donald Johnson) в 1977 г. Время выполнения можно улучшить до $O(E + V \log V)$, используя более сложную структуру данных очереди с приоритетами под названием фибоначчиева куча. Существуют и более быстрые алгоритмы с применением еще более изощренных очередей с приоритетами для особого случая ребер с целочисленными весами.

♥ Отрицательные ребра

Но алгоритм `NonnegativeDijkstra` некорректно вычисляет кратчайшие пути в графах с отрицательными ребрами. Кроме того, даже если веса всех ребер положительны, `NonnegativeDijkstra` не быстрее алгоритма `Dijkstra` (как в теории, так и на практике). По обеим этим причинам я считаю, что `Dijkstra` в большей степени заслуживает названия «алгоритм Дейкстры», чем `NonnegativeDijkstra`. Даже сам Эдсгер Дейкстра согласился бы, что корректный алгоритм, который иногда (а на практике редко) работает медленно, лучше, чем быстрый алгоритм, который работает не всегда.

К сожалению, когда исходный граф содержит отрицательные ребра, знакомая нам интуитивная аналогия с «распространением фронта волны» перестает быть точной. Одна и та же вершина может быть извлечена (`Extract`) несколько раз, напряжение одного и того же ребра может быть ослаблено несколько раз, а расстояния, возможно, не будут просматриваться в порядке возрастания. На рис. 8.15 показан пример выполнения алгоритма, в котором верхняя левая вершина извлекается шесть раз, а для трех верхних ребер напряжение ослабляется по два раза для каждого.

Для графов без отрицательных циклов, но и без других ограничений по весам ребер время выполнения алгоритма `Dijkstra` в наихудшем случае действительно является экспоненциальным. На рис. 8.14 показано чрезвычайно простое семейство графов (связанное с именами Дугласа Шира (Douglas Shier) и Кристофа Вицгалля (Cristoph Witzgall)), в которых алгоритм `Dijkstra` вынужден выполнять $\Theta(2^{V/2})$ операций ослабления напряжения ребер.⁹ Более сложное семейство графов (которое я оставляю читателям в качестве упражнения для самостоятельного выполнения) заставляет выполнить $\Theta(2^V)$ операций ослабления напряжения – это наихудший из возможных случаев. Но на практике алгоритм Дейкстры обычно выполняется быстро даже для графов с отрицательными ребрами.

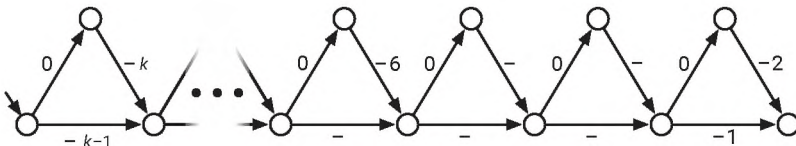
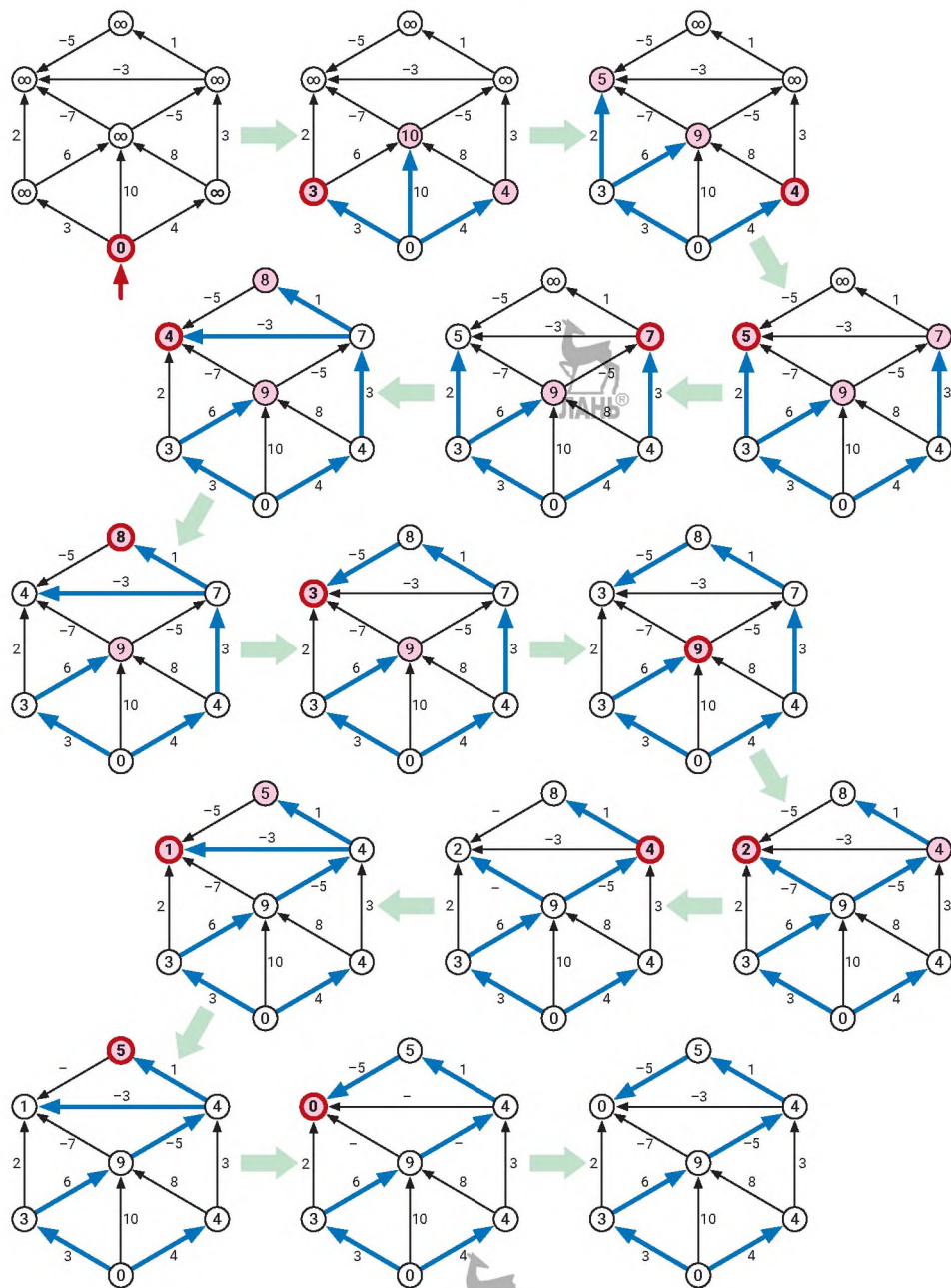


Рис. 8.14. Направленный граф с отрицательными ребрами, в котором алгоритм `Dijkstra` вынужден выполняться за экспоненциальное время

⁹ Удивительно, что пример Шира и Вицгалля — это НАГ всего лишь с $O(V)$ ребрами, в котором предполагается, что кратчайшие пути можно вычислить лишь за время $O(V)$, даже если мы все еще не заметили, что зигзагообразный путь по верхним ребрам — это дерево кратчайшего пути.



8.7. Ослабление напряжения всех ребер: алгоритм Беллмана–Форда

Самая простая реализация обобщенного алгоритма Форда для вычисления кратчайшего пути впервые была представлена в виде чернового проекта Альфонсо Шимбелем (Alfonso Shimbel) в 1954 г., описана более подробно Эдвардом Муром (Edward Moore) в 1957 г. и независимо повторно разработана Максом Вудбери (Max Woodbury) и Джорджем Данцигом (George Dantzig) в 1957 г., Ричардом Беллманом (Richard Bellman) в 1958 г. и Джорджем Минти (George Minty) в 1958 г. (Ни Вудбери и Данциг, ни Минти не публиковали свои алгоритмы.) В полном соответствии с законом эпонимии Стиглера этот алгоритм почти всегда известен как алгоритм Беллмана–Форда¹⁰, потому что Беллман открыто использовал формулировку Форда 1956 г. для ребер с ослабленным напряжением, хотя некоторые авторы ссылаются на «алгоритм Беллмана–Калаба» (Bellamn-Kalaba)¹¹, а в нескольких более ранних источниках упоминается «алгоритм Беллмана–Шимбеля».

Алгоритм Шимбеля/Мура/Вудбери-Данцига/Беллмана–Форда/Калабы/Минти/Брош¹² можно обобщить одной строкой:

Bellman-Ford: Ослабить напряжение **ВСЕХ** напряженных ребер, затем применить рекурсию.

```

BellmanFord(s)
  InitSSSP(s)
  while there is at least one tense edge
    for every edge  $u \rightarrow v$ 
      if  $u \rightarrow v$  is tense
        Relax( $u \rightarrow v$ )
  
```

Приведенная ниже лемма – ключ к доказательству корректности и эффективности алгоритма Беллмана–Форда. Для каждой вершины v и неотрицательного целого числа i пусть $\text{dist}_{\leq i}(v)$ обозначает длину кратчайшего блуждания в графе G из s в v , состоящего из не более чем i ребер. В частности, $\text{dist}_{\leq 0}(s) = 0$ и $\text{dist}_{\leq 0}(v) = \infty$ для всех $v \neq s$.

¹⁰ Я буду следовать этому общепринятому соглашению, несмотря на историческую неточность, отчасти потому, что не думаю, что кто-то захочет читать об «алгоритме Шимбеля/Мура/Вудбери–Данцига/Беллмана–Форда/Калаба/Минти», отчасти потому, что мне надоели люди, глядящие на меня с усмешкой, когда я говорю об «алгоритме Шимбеля».

¹¹ Это имя по большей части упоминается в связи со ссылкой на монографию Ричарда Беллмана и Роберта Калабы (Robert Kalaba) 1965 г. по динамическому программированию и теории управления, в которой описан алгоритм Беллмана. Беллман и Калаба в 1960 г. также опубликовали расширение алгоритма Беллмана, которое вычисляет k -е кратчайшие пути для любой константы k .

¹² И снова рекомендую читать полностью «Преувеличение с половиной» (Hyperbole and a Half). А потом заведите еще одного кота, чтобы можно было купить еще один экземпляр этой книги.

Лемма 8.6. Для каждой вершины v и неотрицательного целого числа i после i итераций основного цикла алгоритма BellmanFord мы имеем $\text{dist}(v) \leq \text{dist}_{\leq i}(v)$.

Доказательство. Доказательство выполняется методом индукции по i . Базовый случай $i = 0$ тривиален, поэтому предполагаем $i > 0$. Определим вершину v и пусть W – кратчайшее блуждание из s в v , состоящий из не более чем i ребер (со случайным разрывом связей). По определению W имеет длину $\text{dist}_{\leq i}(v)$. Необходимо рассмотреть два случая:

- предположим, что W не содержит ребер. Тогда W обязательно должно быть тривиальным блужданием из s в s , поэтому $v = s$ и $\text{dist}_{\leq i}(s) = 0$. Мы устанавливаем $\text{dist}(s) \leftarrow 0$ в подпрограмме InitSSSP, а $\text{dist}(s)$, возможно, никогда не увеличивается, поэтому мы всегда имеем $\text{dist}(s) \leq 0$;
- иначе пусть $u \rightarrow v$ – самое последнее ребро в блуждании W . По индукционному предположению после $i - 1$ итераций $\text{dist}(u) \leq \text{dist}_{\leq i-1}(u)$. Во время i -й итерации внешнего цикла, когда мы рассматриваем ребро $u \rightarrow v$ во внутреннем цикле, либо уже $\text{dist}(v) < \text{dist}(u) + w(u \rightarrow v)$, либо мы устанавливаем $\text{dist}(v) \leftarrow \text{dist}(u) + w(u \rightarrow v)$. В обоих случаях мы имеем $\text{dist}(v) \leq \text{dist}_{\leq i-1}(u) + w(u \rightarrow v) = \text{dist}_{\leq i}(v)$. Как обычно, $\text{dist}(v)$ не может увеличиваться (хотя $\text{dist}(v)$ может и уменьшаться, прежде чем завершится i -я итерация внешнего цикла).

В обоих случаях мы приходим к выводу, что $\text{dist}(v) \leq \text{dist}_{\leq i}(v)$ в конце i -й итерации. □

Если исходный граф не содержит отрицательных циклов, то кратчайшее блуждание из s в любую другую вершину является простым путем с не более чем $V - 1$ ребрами. Из этого следует, что алгоритм BellmanFord останавливается с корректными расстояниями кратчайшего пути после не более чем $V - 1$ итераций. Другими словами, если любое ребро остается напряженным после $V - 1$ итераций, то исходный граф обязательно должен содержать отрицательный цикл. Следовательно, мы можем переписать алгоритм более точно, как показано ниже:

```

BellmanFord(s):
  InitSSSP(s)
  repeat V - 1 times
    for every edge  $u \rightarrow v$ 
      if  $u \rightarrow v$  is tense
        Relax( $u \rightarrow v$ )
  for every edge  $u \rightarrow v$ 
    if  $u \rightarrow v$  is tense
      return "Negative cycle!"    <<"Отрицательный цикл!">>

```

Каждая итерация внутреннего цикла очевидно требует времени $O(E)$, поэтому в целом алгоритм выполняется за время $O(VE)$. Таким образом,

алгоритм Беллмана–Форда эффективен всегда, даже если граф содержит отрицательные ребра, а в действительности, даже если граф содержит отрицательные циклы.

Но если веса всех ребер неотрицательны, то алгоритм Дейкстры работает быстрее, по крайней мере, в наихудшем случае. (На практике алгоритм Дейкстры часто быстрее, чем алгоритм Беллмана–Форда, даже для графов с отрицательными ребрами.)

Улучшение Мура

Ни Мур, ни Беллман не описывали алгоритм Беллмана–Форда в той форме, которую я показал здесь. Мур представил свою версию этого алгоритма (алгоритм D) в той же статье, в которой предлагался алгоритм поиска в ширину (алгоритм A) для невзвешенных графов. Несомненно, эти два алгоритма почти идентичны. Несмотря на то что алгоритм Мура в наихудшем случае имеет то же время выполнения $O(VE)$, что и алгоритм `BellmanFord`, зачастую он значительно быстрее работает на практике, предположительно потому, что избегает проверки ребер, которые «очевидно» не являются напряженными.

Мур вывел свой алгоритм поиска взвешенного кратчайшего пути, создав две модификации алгоритма поиска в ширину. Во-первых, заменяется каждая «+1» на «+ $w(u \rightarrow v)$ » во внутреннем цикле, чтобы принимать в рассмотрение веса ребер. Во-вторых, проверяется, находится ли уже вершина в FIFO-очереди перед ее вставкой (`Insert`), так что очередь всегда содержит не более одной копии каждой вершины.¹³

Следуя нашему предыдущему анализу алгоритма поиска в ширину, я введу маркер $*$ для разделения процесса выполнения алгоритма на два этапа. Как и при поиске в ширину, каждый этап начинается, когда маркер записывается (`Push`) в очередь, а заканчивается, когда маркер снова извлекается (`Pull`) из очереди. Как и при поиске в ширину, алгоритм завершается, когда очередь содержит только маркер. Итоговый алгоритм показан на рис. 8.16.

Поскольку очередь содержит не более одной копии каждой вершины в любой момент времени, каждая вершина извлекается (`Pull`) из очереди не более одного раза на каждом этапе, следовательно, каждое ребро $u \rightarrow v$ проверяется на напряженность не более одного раза на каждом этапе. Кроме того, для каждого ребра, являющегося напряженным в начале этапа, ослабляется напряжение на том же этапе. (Некоторые ребра, которые становятся напряженными на том же этапе, также могут ослабить напряжение на этом этапе, а некоторые ослабленные ребра могут снова стать напряженными опять же на этом этапе.) Таким образом, алгоритм `Moore` можно рассматривать как улучшение алгоритма `BellmanFord`, в котором используется очередь для сопровождения напряженных ребер, а не проверка каждого ребра прямым перебором. По существу, аналогичное индуктивное доказательство устанавливает следующий аналог леммы 8.6.

¹³ Алгоритм Мура остается корректным без этой проверки, но не временная граница $O(VE)$.


```

Moore(s):
  InitSSSP(s)
  Push(s)
  Push(∞)      «начало первого этапа»
  while the queue contains at least one vertex
    u ← Pull( )
    if u = ∞
      Push(∞)   «начало следующего этапа»
    else
      for all edges u → v
        if u → v is tense
          Relax(u → v)
          if v is not already in the queue
            Push(v)

```

Рис. 8.16. Алгоритм Мура для поиска кратчайшего пути. Строки, выделенные красным полужирным шрифтом, вводят маркер ∞ только для анализа

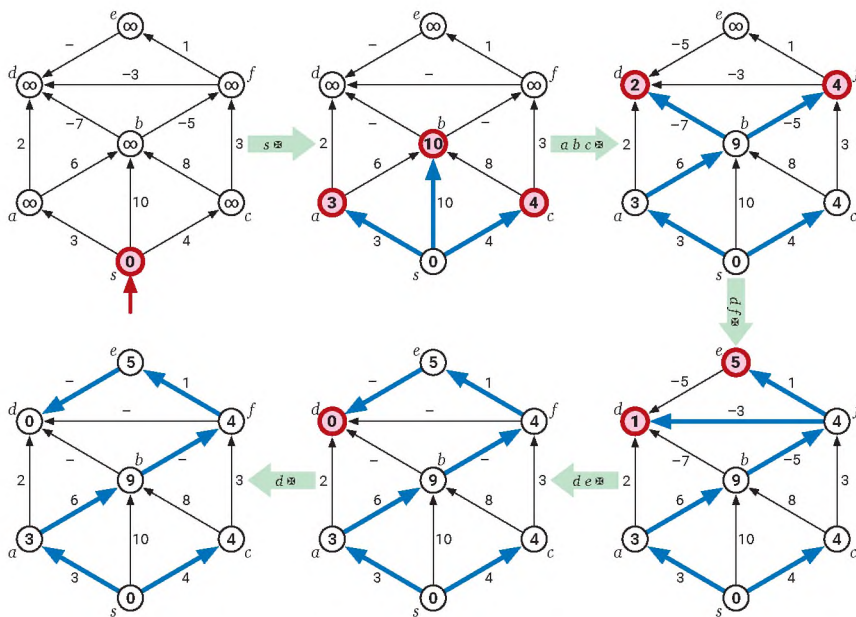


Рис. 8.17. Полная схема выполнения алгоритма Мура в направленном графе с отрицательными ребрами. Узлы извлекаются из очереди в следующем порядке: $s \infty abc \infty df \infty de \infty d \infty \infty$, где ∞ – маркер конца этапа. В начале каждого этапа утолщенные ребра обозначают предков, а затененные вершины находятся в очереди с приоритетами. Сравните с рис. 8.6 и 8.15

Лемма 8.7. Для каждой вершины v и неотрицательного целого числа i после выполнения i этапов алгоритма Moore мы имеем $\text{dist}(v) \leq \text{dist}_{\leq i}(v)$.

Следовательно, если исходный граф не содержит отрицательных циклов, то алгоритм Moore останавливается после не более $V - 1$ этапов. На каждом этапе мы просматриваем каждую вершину не более одного раза, поэтому ослабляем напряжение каждого ребра не более одного раза, так что время выполнения одного этапа в наихудшем случае равно $O(E)$. Таким образом, общее время выполнения алгоритма Moore равно $O(VE)$. Но на практике алгоритм Moore часто вычисляет кратчайшие пути значительно быстрее алгоритма BellmanFord, потому что просматривает ребро $u \rightarrow v$, только если $\text{dist}(u)$ было изменено на предыдущем этапе.

Если исходный граф содержит отрицательный цикл, то алгоритм Moore никогда не остановится. К счастью, как и алгоритм BellmanFord, легко изменить алгоритм Мура так, чтобы он сообщал о наличии отрицательных циклов, если они существуют. Возможно, самой простой модификацией является действительное сопровождение маркера и подсчет числа извлечений (Pull) маркера из очереди. Тогда исходный граф содержит отрицательный цикл, если и только если очередь остается не пустой сразу после извлечения из нее маркера в $(V - 1)$ -й раз.

Формулировка с использованием динамического программирования

Как и почти все, что связано с его именем в этой области, Ричард Беллман (Richard Bellman) создал алгоритм поиска кратчайшего пути «Беллмана–Форда» с помощью динамического программирования. Как обычно, мы должны начать с рекурсивного определения расстояний кратчайшего пути. Есть искушение воспользоваться тем же тождеством, что применялось ранее для направленных ациклических графов:

$$\text{dist}(v) = \begin{cases} 0 & , \text{ если } v = s; \\ \min_{u \rightarrow v} (\text{dist}(u) + w(u \rightarrow v)) & \text{ иначе.} \end{cases}$$

К сожалению, если исходный граф не является НАГ, то это рекуррентное выражение не работает. Предположим, что исходный граф содержит направленный цикл $u \rightarrow v \rightarrow w \rightarrow u$. Для вычисления $\text{dist}(w)$ сначала необходимо вычислить $\text{dist}(v)$, а чтобы вычислить $\text{dist}(v)$, сначала требуется вычисление $\text{dist}(u)$, но для вычисления $\text{dist}(u)$ предварительно нужно вычислить $\text{dist}(w)$. Если исходный граф содержит какие-либо направленные циклы, то мы застреваем в бесконечном цикле.

Для того чтобы рекуррентное выражение стало верным, необходимо добавить в функцию расстояния дополнительный структурный параметр, который монотонно уменьшается при каждом рекурсивном вызове и позво-

ляет определить, что вычисление функции становится тривиальным при достижении параметром значения 0. В качестве такого дополнительного параметра Беллман выбрал максимальное число ребер¹⁴.

Как и в нашем предыдущем анализе, пусть $dist_{\leq i}(v)$ обозначает длину кратчайшего блуждания из s в v , состоящего из не более чем i ребер. Беллман заметил, что эта функция подчиняется следующему уравнению Беллмана рекуррентному выражению:

$$dist_{\leq i}(v) = \begin{cases} 0 & , \text{ если } i = 0 \text{ и } v = s, \\ \infty & , \text{ если } i = 0 \text{ и } v \neq s, \\ \min \left\{ dist_{\leq i-1}(v), \min_{u \rightarrow v} (dist_{\leq i-1}(u) + w(u \rightarrow v)) \right\} & \text{ иначе.} \end{cases}$$

Предположим, что граф не содержит отрицательных циклов, тогда наша цель – вычисление $dist_{\leq V-1}(v)$ для каждой вершины v . В этом рекуррентном выражении присутствует очевидное вычисление методом динамического программирования, где $dist[i, v]$ сохраняет значение $dist_{\leq i}(v)$. Корректность итоговых расстояний кратчайшего пути следует из корректности этого рекуррентного выражения, и очевидно, что время выполнения равно $O(VE)$. Именно так Беллман представил свой алгоритм поиска кратчайшего пути.

```

BellmanFordDP(s):
  dist[0, s] ← 0
  for every vertex v ≠ s
    dist[0, v] ← ∞
  for i ← 1 to V - 1
    for every vertex v
      dist[i, v] ← dist[i - 1, v]
    for every edge u → v
      if dist[i, v] > dist[i - 1, u] + w(u → v)
        dist[i, v] ← dist[i - 1, u] + w(u → v)

```

Мы можем преобразовать этот алгоритм динамического программирования в нашу исходную формулировку алгоритма BellmanFord с помощью короткой последовательности минимальных оптимизаций. Во-первых, каждая итерация внешнего цикла рассматривает каждое ребро $u \rightarrow v$ ровно один раз, но порядок рассмотрения этих ребер в действительности не имеет значения. Следовательно, мы можем безопасно удалить один уровень выравнивания (вправо) из последних трех строк (псевдокода). Измененный алгоритм может рассматривать ребра в другом порядке, но при этом остается корректным вычисление $dist_{\leq i}(v)$ для всех i и v .

¹⁴ Как мы увидим в следующей главе, это не единственный разумный выбор.

```

BellmanFordDP2(s):
  dist[0, s] ← 0
  for every vertex v ≠ s
    dist[0, v] ← ∞
  for i ← 1 to V - 1
    for every vertex v
      dist[i, v] ← dist[i - 1, v]
    for every edge u → v
      if dist[i, v] > dist[i - 1, u] + w(u → v)
        dist[i, v] ← dist[i - 1, u] + w(u → v)

```

Далее мы изменяем индексы в последних двух строках: с $i - 1$ на i . Это изменение может привести к тому, что расстояния $dist[i, v]$ приходят к истинным расстояниям кратчайшего пути быстрее, чем раньше, но алгоритм корректно вычисляет истинные расстояния кратчайшего пути. Вместо $dist[i, v] = dist_{\leq i}(v)$ теперь имеем $dist[i, v] \leq dist_{\leq i}(v)$ для всех i и v в полном соответствии с леммами 8.6 и 8.7.

```

BellmanFordDP2(s):
  dist[0, s] ← 0
  for every vertex v ≠ s
    dist[0, v] ← ∞
  for i ← 1 to V - 1
    for every vertex v
      dist[i, v] ← dist[i - 1, v]
    for every edge u → v
      if dist[i, v] > dist[i, u] + w(u → v)    <<не i - 1!!>>
        dist[i, v] ← dist[i, u] + w(u → v)    <<не i - 1!!>>

```

Но этот алгоритм слегка неразумен. На i -й итерации внешнего цикла мы сначала копируем $(i-1)$ -ю строку массива $dist[\cdot, \cdot]$ в i -ю строку, а затем изменяем элементы i -й строки. В действительности здесь двумерный массив вообще не нужен – итерация по индексу i абсолютно избыточна. В нашем завершающем изменении мы работаем только с одномерным массивом пробных расстояний.

```

BellmanFordFinal(s):
  dist[s] ← 0
  for every vertex v ≠ s
    dist[v] ← ∞
  for i ← 1 to V - 1
    for every edge u → v
      if dist[v] > dist[u] + w(u → v)
        dist[v] ← dist[u] + w(u → v)

```


Этот окончательный алгоритм динамического программирования почти идентичен нашей исходной формулировке алгоритма BellmanFord. В первых трех строках инициализируются расстояния кратчайшего пути, а в последних двух строках ослабляется напряжение ребра $u \rightarrow v$, если оно напряженное. В алгоритме BellmanFordFinal отсутствуют только два функциональных свойства нашей предыдущей формулировки: не поддерживаются указатели на предков и не определяются отрицательные циклы. К счастью, эти функциональные свойства добавляются без затруднений.

Упражнения

0. Пусть G – направленный граф с произвольными весами ребер (веса могут быть положительными, отрицательными или нулевыми), возможно, с отрицательными циклами, и пусть 1 – произвольная вершина графа G .
 - (а) Предположим, что каждая вершина v хранит число $dist(v)$ (но не указатели на предков). Описать и проанализировать алгоритм, определяющий, является ли $dist(v)$ расстоянием кратчайшего пути из s в v для любой вершины v .
 - (б) Теперь предположим, что каждая вершина $v \neq s$ хранит указатель $pred(v)$ на другую вершину в графе G (но не расстояния). Описать и проанализировать алгоритм, выясняющий, определяют ли эти указатели на предков дерево кратчайшего пути из одной вершины-источника с корнем в s .
1. Дерево с циклами (looped tree) — это взвешенный направленный граф, созданный из двоичного дерева добавлением ребра из каждого листа обратно в корень. Каждое ребро имеет неотрицательный вес.

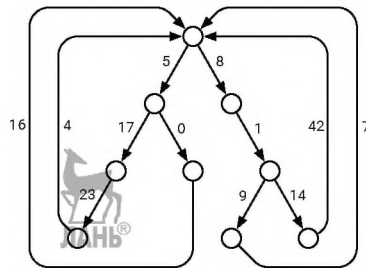


Рис. 8.18. Дерево с циклами

- (а) Сколько вычислений в алгоритме Дейкстры должно потребоваться для определения кратчайшего пути между двумя вершинами u и v в дереве с циклами с n узлами?
 - (б) Описать и проанализировать более быстрый алгоритм.
2. Предположим, что задан направленный граф G с взвешенными ребрами и двумя вершинами s и t .

- (a) Описать и проанализировать алгоритм поиска кратчайшего пути из s в t , если ровно одно ребро в G имеет отрицательный вес. [Подсказка: измените алгоритм Дейкстры. Или не изменяйте.]
 - (b) Описать и проанализировать алгоритм поиска кратчайшего пути из s в t , если ровно k ребер в G имеют отрицательный вес. Каково время выполнения вашего алгоритма в зависимости от k ?
3. Предположим, что задан ненаправленный граф G , в котором каждая вершина имеет положительный вес.
- (a) Описать и проанализировать алгоритм поиска остовного дерева графа G с минимальным общим весом. (Общий вес остовного дерева – это сумма весов его вершин.)
 - (b) Описать и проанализировать алгоритм поиска пути в графе G из одной заданной вершины s в другую заданную вершину t с минимальным общим весом. (Общий вес пути – это сумма весов его вершин.)
- [Подсказка: одна из этих задач является тривиальной.]
4. Для любого ребра e в любом графе G пусть $G \setminus e$ обозначает граф, полученный удалением e из G . Предположим, что задан граф G и две вершины s и t . Задача поиска путей замены (replacement paths) предлагает нам вычислить расстояние кратчайшего пути из s в t в графе $G \setminus e$ для каждого ребра e в графе G . Выводом является массив из E расстояний, по одному для каждого ребра графа G .
- (a) Предположим, что G — направленный граф, и кратчайший путь из вершины s в вершину t проходит через каждую вершину графа G . Описать алгоритм решения этого особого случая задачи поиска путей замены за время $O(E \log V)$.
 - ♥(b) Описать алгоритм решения задачи поиска путей замены для произвольных ненаправленных графов за время $O(E \log V)$.
- В обеих подзадачах вы можете предположить, что веса всех ребер являются неотрицательными. [Подсказка: если мы удаляем ребро из исходного кратчайшего пути, то как перекрываются старый и новый кратчайший путь?]
5. Пусть $G = (V, E)$ — связный направленный граф с неотрицательными весами ребер, пусть s и t — вершины графа G , и пусть H — подграф G , полученный удалением некоторых ребер. Предположим, что необходимо снова вставить ровно одно ребро из G обратно в подграф H , так, чтобы кратчайший путь из s в t в полученном графе являлся самым коротким из всех возможных. Описать и проанализировать алгоритм, который выбирает наилучшее ребро для повторной вставки за время $O(E \log V)$.

6. (a) Описать и проанализировать модификацию алгоритма Беллмана–Форда, которая действительно возвращает отрицательный цикл, если любой такой цикл достижим из s , или дерево кратчайшего пути при отсутствии такого цикла. Измененный алгоритм также должен выполняться за время $O(VE)$.
- (b) Описать и проанализировать модификацию алгоритма Беллмана–Форда, которая вычисляет корректные расстояния кратчайшего пути из s в каждую другую вершину исходного графа, даже если граф содержит отрицательные циклы. В частности, если любое блуждание из s в v содержит отрицательный цикл, то алгоритм должен завершаться с $dist(v) = -\infty$, иначе $dist(v)$ должно содержать длину кратчайшего пути из s в v . Этот модифицированный алгоритм должен сохранить время выполнения $O(VE)$.
- ♥(c) Повторить решение частей (a) и (b), но для обобщенного алгоритма Форда ослабления напряжения ребер. Вы можете предположить, что немодифицированный алгоритм останавливается через $O(2^V)$ шагов, если нет отрицательных циклов. Модифицированный алгоритм также должен выполняться за время $O(2^V)$.
7. Рассмотрим еще более свободный вариант обобщенного алгоритма Форда ослабления напряжения ребер, показанный ниже:

```

FellmanBored(s):
  InitSSSP(s)
  for i ← 1 to whatever, man, I don't care
    «for i ← 1 до чего угодно, дружище, мне все равно»
    ei ← any edge in G          «ei ← любое ребро в G»
    if ei is tense               «если ei – напряженное ребро»
      Relax(ei)                 «ослабить напряжение ребра ei»

```

Доказать, что если алгоритм `FellmanBored` просматривает ребра любого блуждания W , начиная с s , в порядке вдоль W , то последняя метка расстояния в W не превышает длину W . Более формально: если ребра любого блуждания $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_\ell$, где $v_0 = s$, определяют последовательность ребер $e_1, e_2, e_3, \dots$, просмотренных алгоритмом `FellmanBored`, то мы имеем $dist(v_\ell) \leq \sum_{i=1}^{\ell} w(v_{i-1} \rightarrow v_i)$. [Подсказка: это свойство почти проще доказать, чем верно сформулировать.]

8. В этой задаче рассматривается несколько способов обнаружения отрицательных циклов с использованием обобщенного алгоритма Форда ослабления напряженных ребер.
- (a) Доказать, что если $pred(s)$ непрерывно изменяется после `InitSSSP`, то исходный граф содержит отрицательный цикл, проходящий через s .

- (b) Показать, что $pred(s)$ может никогда не изменяться после InitSSSP, даже если исходный граф содержит отрицательный цикл, проходящий через s .
 - (c) Пусть P обозначает текущий граф ребер-предков $pred(v) \rightarrow v$ и пусть X обозначает множество всех текущих напряженных ребер. Оба эти множества расширяются в процессе выполнения алгоритма. Доказать, что исходный граф не содержит отрицательных циклов, если и только если $P \cup X$ всегда является НАГ.
 - (d) Пусть R обозначает множество всех ребер, напряжение которых было ослаблено до настоящего момента. Это множество растет в процессе выполнения алгоритма. Доказать, что исходный граф не содержит отрицательных циклов, если и только если R всегда является НАГ.
- ♥9. Доказать, что алгоритм Дейкстры выполняет $\Omega(2^V)$ операций ослабления напряжения ребер в наихудшем случае, когда для ребер разрешены отрицательные веса, даже если внутренний граф является ациклическим. В частности, для каждого положительного целого числа n создать НАГ G_n с n вершинами и взвешенными ребрами, такой что алгоритм Дейкстры вызывает Relax $\Omega(2^n)$ раз, если G_n является исходным графом. [Подсказка: двоичный счетчик.]
- ♥10. Доказать, что обобщенный алгоритм Форда ослабления напряжения ребер (следовательно, и алгоритм Дейкстры) останавливается после выполнения не более чем $O(2^V)$ операций ослабления напряжения ребер, если только исходный граф не содержит отрицательный цикл. [Подсказка: см. задачу 8(d).]
11. Предположим, что задан направленный граф G , в котором каждое ребро имеет отрицательный вес, и вершина-источник s . Описать и проанализировать эффективный алгоритм вычисления расстояний кратчайших путей из s в каждую другую вершину графа G . В частности, для каждой вершины t :
- если t недостижима из s , то алгоритм должен возвращать $dist(t) = \infty$;
 - если граф G содержит цикл, достижимый из s , и t достижима из этого цикла, то расстояние кратчайшего пути из s в t является не точно определенным, потому что существуют пути (формально: блуждания) из s в t произвольно большой отрицательной длины. В этом случае алгоритм должен возвращать $dist(t) = -\infty$;
 - если ни одно из двух предыдущих условий не выполнено, то алгоритм должен возвращать корректное расстояние кратчайшего пути из s в t .
12. Несмотря на то что обычно мы говорим о конкретном кратчайшем пути между двумя узлами, один и тот же граф может содержать не-

сколько путей минимальной длины с одинаковыми конечными точками. Даже для взвешенных графов часто желательно выбрать путь минимального веса с наименьшим числом ребер – назовем это наилучшим путем (best path) из s в t . Предположим, что задан направленный граф G с положительными весами ребер и вершина-источник s в графе G . Описать и проанализировать алгоритм вычисления наилучших путей в G из s в каждую другую вершину.

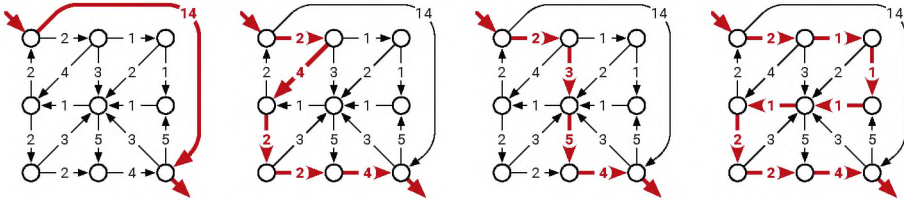


Рис. 8.19. Четыре (из многих) кратчайших пути равной длины. Первый путь является «наилучшим» кратчайшим путем

13. Описать и проанализировать алгоритм, определяющий число кратчайших путей из вершины-источника s в целевую вершину t в произвольно направленном графе G со взвешенными ребрами. Вы можете предположить, что веса всех ребер положительные и что все необходимые арифметические операции выполняются за время $O(1)$. [Подсказка: вычислите расстояния кратчайших путей из s в каждую другую вершину. Отбросьте все ребра, которые не могут являться частью кратчайшего пути из s в другую вершину. Что осталось?]

14. Вы только что обнаружили своего лучшего друга по начальной школе в Twitbook. Вы оба хотите встретиться как можно быстрее, но живете в разных городах, расположенных далеко друг от друга. Чтобы минимизировать время поездки, вы договорились встретиться в промежуточном городе, затем вы одновременно запрыгнули в свои машины и начали движение навстречу друг другу. Но где в точности вы должны встретиться?

Задан взвешенный граф $G = (V, E)$, в котором вершины V представляют города, а ребра E – дороги, напрямую соединяющие эти города. Каждое ребро e имеет вес $w(e)$, равный времени, требуемому для проезда между двумя городами. Также задана вершина p , представляющая начальную локацию, и вершина q , представляющая начальную локацию вашего друга.

Описать и проанализировать алгоритм поиска целевой вершины t , которая позволяет вам как можно быстрее встретиться со своим другом.

15. Вы получили работу в качестве велогонщика в проекте Gigggle Highway View, который должен будет предоставлять изображения на

уровне дороги для всей национальной системы скоростных шоссе США. В пилотном проекте вам предлагается проехать на велосипеде новейшей модели Gigggle Highway-View Fixed-Gear Carbon-Fiber Bicycle из «Гигглплекса» в Портленде (шт. Орегон) в «Гигглсберг» в Вильямсбурге, Бруклин, Нью-Йорк.

Вы безнадежный кофеман, но, как и большинство сотрудников Gigggle, также являетесь снобом по отношению к кофе: пьете только напитков из отдельно прожаренных, отобранных вручную, приобретенных у производителя, выращенных без химии на специальных крытых плантациях в одном месте кофейных зерен, – эспрессо, без добавления молока и сахара, большое спасибо. После каждой порции эспрессо вы можете преодолеть на велосипеде до L миль, прежде чем начнется приступ головной боли, вызванный недостатком кофеина. Компания Gigggle любезно предоставляет вам карту США в виде ненаправленного графа G , вершины которого представляют кофейни, в которых продается эспрессо из отдельно прожаренных, отобранных вручную, приобретенных у производителя, выращенных без химии на специальных крытых плантациях в одном месте кофейных зерен, а ребра – скоростные шоссе, соединяющие эти кофейни. Каждое ребро e помечено значением длины $\ell(e)$ соответствующих участков скоростных шоссе. Разумеется, требуемые порции эспрессо находятся в обоих офисах Gigggle, представленных двумя особыми вершинами s и t в графе G .

- (a) Описать и проанализировать алгоритм, определяющий, возможно ли проехать на велосипеде из Гигглплекса в Гигглсберг без появления головной боли из-за недостатка кофеина.
- (b) Вы обнаружили, что, надев более дорогую мягкую фетровую шляпу – федору (fedora), можно увеличить расстояние L , которое вы способны проехать на велосипеде между приемом порций эспрессо. Описать и проанализировать алгоритм поиска минимального значения L , которое позволяет проехать на велосипеде из Гигглплекса в Гигглсберг без появления головной боли из-за недостатка кофеина.
- (c) Когда вы сообщили своему руководителю (ее недавно перевербовали в Gigggle из конкурирующей компании Ünter), что такой маршрут проезда невозможен, она попросила разрешения взглянуть на вашу карту. «О, я поняла, в чем проблема! На этой карте нет кофеен Старбакс (Starbucks)». Поскольку вы посмотрели с ужасом, она вручила вам обновленный граф G' , в который включены вершины для каждой локации Старбакс в США, заботливо помеченные зеленым цветом (Starbucks Green, Pantone® 3425 C).

Описать и проанализировать алгоритм поиска минимального числа кофеен Старбакс, которые вы обязательно должны посе-

тить для проезда на велосипеде из Гигглплекса в Гигглсберг без появления головной боли из-за недостатка кофеина. Более формально: алгоритм должен найти минимальное число зеленых вершин в любом пути в графе G' из s в t , использующем только ребра, длина которых не превышает L .

16. Предположим, что задан направленный граф $G = (V, E)$ с неотрицательными взвешенными ребрами и двумя вершинами s и t . Описать и проанализировать алгоритм поиска кратчайшего блуждания в G из s в t (возможно, с повторяющимися вершинами и/или ребрами), число ребер в котором нацело делится на 3.

Например, если задан граф, показанный на рис. 8.20, с обозначенными вершинами s и t , в котором все ребра имеют вес 1, то ваш алгоритм должен вернуть 6, т. е. длину блуждания $s \rightarrow w \rightarrow y \rightarrow x \rightarrow s \rightarrow w \rightarrow t$, равную 6.

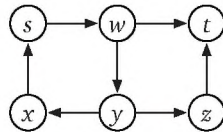


Рис. 8.20. Пример графа для алгоритма поиска кратчайшего блуждания

17. Предположим, что задан направленный граф G с неотрицательными взвешенными ребрами, в котором некоторые ребра красные, а остальные – синие. Описать алгоритм поиска кратчайшего блуждания в G из одной вершины s в другую вершину t , в которой никакие три следующие друг за другом ребра не имеют одинаковый цвет. То есть если блуждание содержит два красных ребра, следующих друг за другом, то следующее ребро обязательно должно быть синим, а если блуждание содержит два синих ребра, следующих друг за другом, то следующее ребро непременно должно быть красным.

Например, если в качестве исходного задан граф, показанный на рис. 8.21, в котором каждое красное ребро имеет вес 1, а каждое синее ребро – вес 2, то ваш алгоритм должен вернуть целое число 9, потому что кратчайшее допустимое блуждание из s в t : $s \rightarrow a \rightarrow b \Rightarrow d \rightarrow c \Rightarrow a \rightarrow b \rightarrow c$.

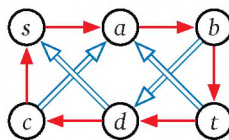


Рис. 8.21. Пример графа с разноцветными ребрами разного веса для алгоритма поиска кратчайшего блуждания

18. Рассмотрим направленный граф G , в котором каждое ребро имеет неотрицательный вес и окрашено в красный, белый или синий цвет. Блуждание в G называется блужданием флага Франции, если содержит последовательность цветов ребер: красный, белый, синий, красный, белый, синий и т. д. Более формально: блуждание $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k$ является блужданием флага Франции, если для каждого целого числа i ребро $v_i \rightarrow v_{i+1}$ красное, если $i \bmod 3 = 0$, белое, если $i \bmod 3 = 1$, и синее, если $i \bmod 3 = 2$.

Описать алгоритм поиска кратчайших блужданий флага Франции из одной начальной вершины s в каждую другую вершину в графе G .

19. Существует n галактик, соединенных m межгалактическими телепортационными маршрутами. Каждый маршрут телепортации соединяет две галактики, и по нему можно путешествовать в обоих направлениях. Кроме того, с каждым маршрутом телепортации связана стоимость $c(e)$ долл., где $c(e)$ – положительное целое число. Маршрут телепортации можно использовать несколько раз, но оплачивать его необходимо при каждом использовании.

Джуди хочет путешествовать из галактики s в галактику t как можно дешевле. Но ей необходимо, чтобы общая стоимость была кратна 5 долл., потому что носить с собой сдачу мелочью неудобно.

- (a) Описать и проанализировать алгоритм вычисления минимальной общей стоимости путешествия из галактики s в галактику t с учетом ограничения по кратности общей стоимости 5 долл.
 - (b) Решить задачу из части (a), но теперь предположить, что у Джуди есть купон, позволяющий ей бесплатно использовать ровно один маршрут телепортации.
20. После переезда в новый город вы решили выбрать пешеходный маршрут от дома до нового офиса. Для обеспечения достаточной ежедневной физической нагрузки этот маршрут обязательно должен состоять из подъема на возвышенность (для тренировки), за которым следует спуск с возвышенности (для заминки), или только из подъема, или только из спуска. (Домой вы возвращаетесь тем же маршрутом, так что так или иначе потренируетесь.) Но вам также необходим кратчайший путь, удовлетворяющий эти условия, чтобы вы действительно приходили на работу вовремя.
- Исходные данные состоят из ненаправленного графа G , вершины которого представляют перекрестки, а ребра – участки дорог, а также начальная вершина s и целевая вершина t . С каждой вершиной v связано значение $h(v)$ – высота этого перекрестка над уровнем моря, а с каждым ребром uv связано значение $\ell(uv)$ – длина этого участка дороги.

- (а) Описать и проанализировать алгоритм поиска кратчайшего блуждания с подъемами и спусками из s в t . Предположить, что все высоты вершин различны.
- (б) Теперь предположим, что некоторые или все вершины могут иметь равные высоты. Описать и проанализировать алгоритм поиска кратчайшего блуждания «подъем, потом спуск» из s в t . Можно использовать ребра без уклона («ровные») в частях «подъема» и «спуска» вашего блуждания.
- (с) Наконец, предположим, вы обнаружили, что не существует пути из s в t с необходимой вам структурой. Описать и проанализировать алгоритм поиска пути из s в t , который чередует участки «подъема» и «спуска» столько раз, сколько возможно, и имеет минимальную длину из всех таких путей.
21. После окончания университета Шам-Пубанана (Sham-Poobanana University) вас приняли на работу в Aerophobes-Я-Us – ведущее транспортное агентство для людей, которые ненавидят летать самолетами. Ваша работа заключается в создании системы помощи клиентам в планировании воздушных перелетов из одного города в другой. Все клиенты боятся летать (как, впрочем, и аэропортов), поэтому любой планируемый вами маршрут должен быть как можно более коротким. Вам известны все времена отправления и прибытия всех авиатранспортных средств на планете.
- Предположим, что один из ваших клиентов желает совершить перелет из города X в город Y . Описать алгоритм поиска последовательности авиарейсов, которая минимизирует общее время в полете – длину интервала времени от начального отправления (взлета) до конечного прибытия (посадки), включая время в промежуточных аэропортах при ожидании очередного авиарейса.
22. В упражнении 20 из главы 5 вы проектировали алгоритм, который определял, является ли разрешимым заданный остроугольный лабиринт. В этой задаче вы будете проектировать алгоритмы поиска кратчайшего блуждания в заданном остроугольном лабиринте при двух различных определениях понятия «длина».

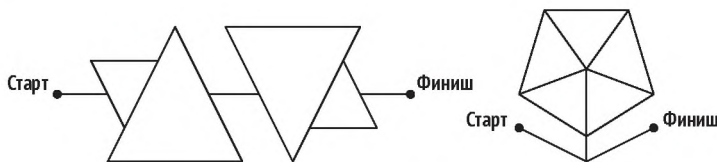


Рис. 8.22. Пройти оба лабиринта, показанные здесь, прослеживая путь от старта до финиша, который содержит только острые углы

Входные данные: связный ненаправленный граф G , вершинами которого являются точки на плоскости, а ребрами – отрезки прямых. Ребра не пересекаются нигде, кроме конечных точек. Например, изображение буквы X должно содержать пять вершин и четыре ребра, первый лабиринт на рис. 8.22 (слева) содержит 14 вершин и 21 ребро. Также заданы две особые вершины: *Старт* и *Финиш*.

Блуждание от *Старта* к *Финишу* в графе G разрешено, если оно содержит только острые углы, или более формально: для любых двух последовательных ребер $u \rightarrow v \rightarrow w$ либо $\angle u \rightarrow v \rightarrow w = \pi$, либо $0 < \angle u \rightarrow v \rightarrow w < \pi/2$. Предположим, что вы можете определить за время $O(1)$, является ли угол между двумя заданными отрезками развернутым, тупым, прямым или острым.

- (a) Описать алгоритм вычисления разрешенного пути от *Старта* к *Финишу*, который проходит минимально возможное число отрезков. (Если блуждание проходит один и тот же отрезок дважды, то этот отрезок учитывается дважды.)
- (b) Описать алгоритм вычисления разрешенного пути от *Старта* к *Финишу*, который совершает минимально возможное число поворотов. [Подсказка: это не та же задача, что в части (a).]
- (c) Описать алгоритм вычисления разрешенного пути от *Старта* к *Финишу*, который имеет минимально возможное общее евклидово расстояние. (Предположим, что можете также вычислить длину любого отрезка за время $O(1)$.)

23. После сложнейшего экзамена в середине семестра в See-Bull Center по определению ложных новостей (Fake News Detection) вы решили поехать домой на автобусе. Поскольку вы планируете поездку заранее, то имеете расписание, в котором перечислены времена и места расположения каждой остановки каждого автобуса в Шам-Пубанана (Sham-Poobanana). К сожалению, ни один из автобусов не останавливается ни около See-Bull Center, ни около вашего дома, поэтому вы непременно должны пересестись в другой автобус как минимум один раз. Существует ровно b различных автобусов. Каждый автобус начинает движение в 12:00:01 дня, делает ровно n остановок и завершает работу на маршруте в 11:59:59 вечера. Автобусы всегда следуют точно по расписанию, а у вас есть точные часы. Наконец, вы слишком устали, чтобы ходить от остановки к остановке.

- (a) Описать и проанализировать алгоритм, определяющий последовательность автобусных рейсов, которые доставят вас домой как можно раньше. Ваша цель – минимизировать время прибытия, а не общее время, затраченное на поездку.
- (b) О нет! Экзамен в середине семестра пришелся на Хеллоуин, и улицы заполнили зомби. Районное управление общественным транспортом Шам-Пубанана не располагает средствами для вы-

деления дополнительных автобусов или для установки защищенных от зомби автобусных остановок, тем более всего на одну ночь в году. Описать и проанализировать алгоритм, определяющий последовательность автобусных рейсов, который минимизирует ваше общее время ожидания на автобусных остановках. При этом не имеет значения, насколько поздно вы доберетесь до дома или сколько времени вы проведете в автобусах. (Предположим, что вы можете подождать в See-Bull Center до тех пор, пока около него не остановится первый требуемый автобус.)

24. В первое утро после возвращения с прекрасных весенних каникул Алиса обнаружила, что ее автомобиль не заводится, так что ей придется добираться на занятия в университет Шам-Пубанана общественным транспортом. У Алисы есть полное расписание движения пассажирского транспорта в округе Пубанана. Маршруты автобусов в расписании изображены в виде направленного графа G , вершины которого представляют автобусные остановки, а ребра – участки автобусных маршрутов между остановками. Для каждого ребра $u \rightarrow v$ в расписании указаны три положительных действительных числа:

- $\ell(u \rightarrow v)$ — длина участка автобусного маршрута от остановки u до остановки v (в мин);
- $f(u \rightarrow v)$ — первое время (в мин после 12 часов дня), в которое автобус отъезжает с остановки u к остановке v ;
- $\Delta(u \rightarrow v)$ — время между последовательными отправлениями от остановки u к остановке v (в мин).

Таким образом, первый автобус на этом маршруте отъезжает с остановки u во время $f(u \rightarrow v)$ и прибывает на остановку v во время $f(u \rightarrow v) + \ell(u \rightarrow v)$, второй автобус покидает u во время $f(u \rightarrow v) + \Delta(u \rightarrow v)$ и прибывает на остановку v во время $f(u \rightarrow v) + \Delta(u \rightarrow v) + \ell(u \rightarrow v)$, третий автобус отправляется от u во время $f(u \rightarrow v) + 2 \cdot \Delta(u \rightarrow v)$ и останавливается на v во время $f(u \rightarrow v) + 2 \cdot \Delta(u \rightarrow v) + \ell(u \rightarrow v)$ и т. д.

Алиса хочет выехать с остановки s (около ее дома) в определенное время и прибыть на остановку t (See-Bull Center) как можно быстрее. Если Алиса приезжает на остановку одного автобуса точно в то время, когда другой автобус отъезжает по расписанию, то она может успеть пересесть во второй автобус. Поскольку Алиса – студентка ШПУ, то проезд в автобусе для нее бесплатный, поэтому ей все равно, сколько раз придется пересаживаться в другие автобусы.

Описать и проанализировать алгоритм поиска самого раннего времени достижения конечной цели Алисы. Входные данные состоят из направленного графа $G = (V, E)$, вершин s и t , значений $\ell(e)$, $f(e)$, $\Delta(e)$ для каждого ребра $e \in E$ и начального времени поездки Алисы (в мин после 12 ч дня).

[Подсказка: в этом редко встречающемся варианте задачи, возможно, проще изменить алгоритм, чем исходный граф.]

25. Малдер и Скалли вычислили для каждой дороги в США точную вероятность того, что водитель, проезжающий по той или иной дороге, не будет похищен инопланетянами. Агенту Малдеру необходимо проехать из Лэнгли (Вирджиния) в Зону 51 (Невада). Какой маршрут должен выбрать Малдер, чтобы шанс его похищения был минимальным?

Более формально: задан направленный граф $G = (V, E)$, в котором каждому ребру e соответствует независимая вероятность безопасного проезда $p(e)$. Общая безопасность пути равна произведению вероятностей безопасности его ребер. Разработать и проанализировать алгоритм определения самого безопасного пути из заданной начальной вершины s в заданную целевую вершину t . Вы можете предположить, что все необходимые арифметические операции можно выполнить за время $O(1)$.

Например, при значениях вероятностей, показанных на рис. 8.23, если Малдер пытается проехать напрямую из Лэнгли в Зону 51, то имеет 50%-ный шанс доехать до цели без похищения. Если Малдер едет с остановкой в Мемфисе, то шанс безопасного проезда равен $0.7 \times 0.9 = 63\%$. Если Малдер выбирает маршрут с остановкой в Мемфисе, а затем в Лас Вегасе, то шанс быть похищенным становится равным $1 - 0.7 \times 0.1 \times 0.5 = 96.5\%$. (Именно так они встретили Элвиса, как вам известно.)

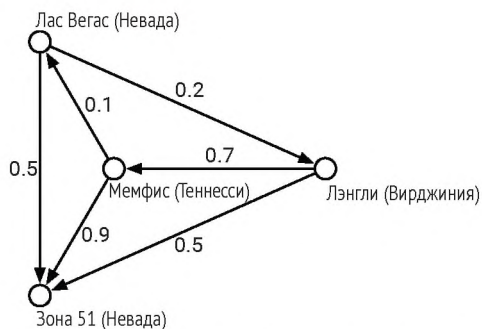


Рис. 8.23. Пример графа для выбора безопасного пути

- ♣26. Во время ночевки в коротком походе по национальному парку Солнечная Долина ваш беспокойный сон был прерван криком. Когда вы выползли из палатки, чтобы узнать, что случилось, то увидели перепуганного зрителя парка, выбегающего из леса. Он был весь в крови, а на его груди был прикреплен мятый клочок бумаги. Он добежал до палатки и задышавшись проговорил «Беги... пока... тебя...»,

сунул клочок бумаги вам в руку и упал на землю. Проверив его пульс, вы поняли, что смотритель мертв.

Вы взглянули на клочок бумаги и поняли, что это карта парка, изображенная в виде ненаправленного графа, в котором вершины представляют межевые знаки в парке, а ребра – тропы между этими знаками. (Тропы начинаются и заканчиваются у межевых знаков и не пересекаются.) В одной из вершин вы узнали свое текущее местоположение. Несколько вершин на границе карты помечены надписью EXIT (Выход).

При более подробном рассмотрении вы заметили, что кто-то (возможно, несчастный погибший смотритель парка) написал действительные числа от 0 до 1 около каждой вершины и каждого ребра. Примечание, нацарапанное на обороте карты, сообщает, что число около ребра – это вероятность встретить вампира на соответствующей тропе, а число около вершины – вероятность встретить вампира у соответствующего межевого знака. (Вампиры не могут оставаться в компании другого вампира, поэтому вы не встретите больше одного вампира на одной тропе и у одного межевого знака.) Примечание также предупреждает, что если вы сойдете с помеченных троп, то вас ждет медленная и мучительная смерть.

Вы взглянули на тело у ваших ног. Да, его смерть определенно была мучительной. Но постойте, тело, кажется, содрогнулось? А его зубы стали длиннее? Вы проткнули колом от палатки сердце смотрителя, превратившегося в нежить, после чего разумно решили немедленно покинуть парк, и как можно быстрее.

Описать и проанализировать эффективный алгоритм поиска пути из вашего текущего местоположения в любой узел с пометкой EXIT (Выход), такой что суммарное ожидаемое число вампиров, встречающихся на этом пути, было бы по возможности минимальным. Будьте внимательны: необходимо учитывать вероятности и около вершин, и около ребер. [Подсказка: даже без вероятностей на вершинах это не та же задача, что предыдущая.]

Глава 9

Кратчайшие пути между всеми парами вершин в графе

Ибо большое дерево вырастает из маленького,
девятиэтажная башня начинает строиться из горстки земли,
путешествие в тысячу ли начинается с одного шага.¹

— Лао-цзы (Lao-Tzu), «Книга пути и достоинства» («Tao Te Ching» – «Дао де цзин»),
глава 64 (VI в. до н. э.), [на англ. яз. перевел Джеймс Легг (James Legge) (1891 г.)]

И я прошел бы пять сотен миль,
Потом прошел бы еще пятьсот,
Чтобы стать тем, кто, пройдя тысячу миль,
Упадет у твоего порога.

— The Proclaimers, «I'm Gonna Be (500 Miles)», альбом «Sunshine on Leith» (2001 г.)

Еще немного... Еще чуть-чуть...

— Красный ведущий (Red Leader) (Гарвен Дрейс) [Дрю Хенли (Drew Henley)],
«Звездные войны» («Star Wars») (1977 г.)

9.1. Введение

В предыдущей главе мы обсудили несколько алгоритмов поиска кратчайших путей из одной вершины-источника s в каждую другую вершину графа, формируя дерево кратчайшего пути с корнем в s . Это дерево кратчайшего пути определяет два свойства для каждого узла v в графе:

- $dist(v)$ – длину кратчайшего пути из s в v ;
- $pred(v)$ – вторую с конца вершину в кратчайшем пути из s в v .

В этой главе мы рассматриваем более общую задачу поиска кратчайших путей между всеми парами вершин в графе (all pairs shortest path – APSP), в которой требуется найти кратчайший путь из каждого возможного источ-

¹ Перевод Яна Хин-шуна <http://tao-te-ching.ru/>.

ника в каждую возможную цель. Для каждой пары вершин u и v необходимо вычислить следующую информацию: ▸

- $dist(u, v)$ – длину кратчайшего пути из u в v ;
- $pred(u, v)$ – вторую с конца вершину в кратчайшем пути из u в v .

Эти интуитивные определения исключают несколько граничных случаев, которые мы уже полностью рассмотрели в предыдущей главе.

- Если не существует пути из u в v , то не существует и кратчайшего пути из u в v . В этом случае мы определяем $dist(u, v) = \infty$ и $pred(u, v) = \text{Null}$.
- Если существует отрицательный цикл между u и v , то существуют пути² из u в v с произвольной отрицательной длиной. В этом случае мы определяем $dist(u, v) = -\infty$ и $pred(u, v) = \text{Null}$.
- Наконец, если u не лежит в отрицательном цикле, то кратчайший путь из u в саму себя не содержит ребер, следовательно, не имеет последнего ребра. В этом случае мы определяем $dist(u, u) = 0$ и $pred(u, u) = \text{Null}$.

Требуемым выводом задачи поиска кратчайших путей между всеми парами вершин в графе является пара массивов $V \times V$ – один содержит все V^2 длин кратчайших путей³, другой – все V^2 предшественников. В этой главе я почти полностью сосредоточил внимание на вычислении массива расстояний. Массив предшественников, из которого мы можем определить действительные кратчайшие пути, можно вычислить с минимальными изменениями (подсказка).

9.2. Множество отдельных источников

Самое очевидное решение задачи поиска кратчайших путей между всеми парами вершин в графе – выполнение алгоритма поиска кратчайшего пути из одной вершины во все остальные V раз, по одному для каждой возможной вершины-источника. В частности, для заполнения одномерного подмассива $dist[s, \cdot]$ мы вызываем алгоритм одного источника, начиная с вершины-источника s .

```
ObviousAPSP(V, E, w):
  for every vertex s
    dist[s, ·] ← SSSP(V, E, w, s)
```

Время выполнения этого алгоритма очевидно зависит от того, какой алгоритм поиска кратчайших путей из одной вершины мы используем. Как и

² Формально маршруты.

³ В те времена, когда еще использовались карты дорог, напечатанные на бумаге, а поиск на них выполнялся вручную, достаточно часто такие карты включали треугольные «таблицы расстояний». Например, чтобы определить расстояние от Шампейна до Коламбуса, вы должны были найти строку с меткой «Champaign» и столбец с меткой «Columbus».

в случае с одним источником, существует четыре естественных варианта, зависящих от структуры графа и весов его ребер:

- если ребра графа не имеют весов, то поиск в ширину дает нам общее время выполнения $O(VE) = O(V^5)$;
- если граф ациклический, то просмотр вершин в топологическом порядке также дает нам общее время выполнения $O(VE) = O(V^5)$;
- если веса всех ребер неотрицательны, то алгоритм Дейкстры дает нам время выполнения $O(VE \log V) = O(V^5 \log V)^4$;
- наконец, в общем случае алгоритм Беллмана–Форда дает нам асимптотическое время выполнения $O(V^2E) = O(V^4)$.

9.3. Изменение весов

Отрицательные ребра существенно замедляют нас, но можно ли избавиться от них? У многих людей возникает простая идея: увеличить веса всех ребер на одинаковую величину, чтобы все веса стали положительными и можно было бы использовать алгоритм Дейкстры вместо алгоритма Беллмана–Форда. К сожалению, эта простая идея не работает, интуитивно потому, что наши два понимания «длины» несовместимы – пути с большим числом ребер могут иметь меньший общий вес по сравнению с путями с меньшим числом ребер. Если мы увеличиваем веса всех ребер на одинаковую величину, то пути с большим числом ребер становятся длиннее быстрее, чем пути с меньшим числом ребер, в результате кратчайший путь между двумя вершинами может измениться.

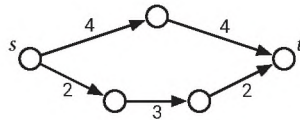


Рис. 9.1. Увеличение весов всех ребер на 2 изменяет кратчайший путь из s в t

Но существует более хитроумный метод изменения весов ребер, который сохраняет кратчайшие пути. Этот метод изменения весов ребер часто приписывают Дональду Джонсону (Donald Johnson), который описал его практическое применение к алгоритмам поиска кратчайшего пути в 1973 г. Но в действительности Джонсон указал источник этого метода – статью 1972 г. Джека Эдмондса (Jack Edmonds) и Ричарда Карпа (Richard Karp). Тот же метод также был описан Нобуаки Томицава (Nobuaki Tomizawa) в 1971 г. и в несколько другой форме Делбертом Фалкерсоном (Delbert Fulkerson) в 1961 г.

⁴ И снова если мы заменим двоичную кучу в нашей реализации алгоритма Дейкстры на неотсортированный массив, то общее время выполнения станет равным $O(V^3)$ (не имеет значения, сколько ребер содержит граф), а если мы заменим двоичную кучу на фибоначиеву кучу, то время выполнения уменьшится до $O(V(E + V \log V)) = O(VE + V^2 \log V) = O(V^3)$.

Предположим, что с каждой вершиной v связана стоимость $\pi(v)$, которая может быть положительной, отрицательной или нулевой. Мы можем определить новую весовую функцию w' , как показано ниже:

$$w'(u \rightarrow v) = \pi(u) + w(u \rightarrow v) - \pi(v).$$

Для интуитивного понимания представим себе, что когда мы покидаем вершину u , то должны заплатить за выход стоимость $\pi(u)$, а при входе в вершину v мы получаем $\pi(v)$ как подарок за вход.

Нетрудно видеть, что кратчайшие пути с новой весовой функцией w' в точности те же самые, что и кратчайшие пути с первоначальной весовой функцией w . В действительности для любого пути $u \rightarrow v$ из одной вершины u в другую вершину v имеем:

$$w'(u \rightsquigarrow v) = \pi(u) + w(u \rightsquigarrow v) - \pi(v).$$

Мы платим $\pi(u)$ за выход плюс исходный вес этого пути минус подарок за вход $\pi(v)$. В каждой промежуточной вершине x этого пути мы получаем $\pi(x)$ как подарок за вход, но затем сразу же возвращаем этот подарок как плату за выход. Поскольку все пути из u в v изменяют длину в точности на одинаковую величину, кратчайший путь из u в v не изменяется. (Длины путей между различными парами вершин, возможно, изменяются на различные величины, поэтому, возможно, их порядок также изменяется.)

9.4. Алгоритм Джонсона

Алгоритм Джонсона для поиска кратчайших путей между всеми парами вершин в графе вычисляет стоимость $\pi(v)$ для каждой вершины так, что новый вес каждого ребра является неотрицательным, а затем вычисляет кратчайшие пути с учетом этих новых весов, используя алгоритм Дейкстры.

Сначала предположим, что исходный граф содержит вершину s , из которой можно достичь все прочие вершины. Алгоритм Джонсона вычисляет кратчайшие пути из s в другие вершины, используя алгоритм Беллмана–Форда (для которого не имеет значения, если веса ребер отрицательные), а затем изменяет веса в графе, применяя функцию стоимости $\pi(v) = \text{dist}(s, v)$. Новый вес каждого ребра равен

$$w'(u \rightarrow v) = \text{dist}(s, u) + w(u \rightarrow v) - \text{dist}(s, v).$$

Эти новые веса неотрицательные, потому что алгоритм Беллмана–Форда остановился. Напомню, что ребро $u \rightarrow v$ является напряженным, если $\text{dist}(s, u) + w(u \rightarrow v) < \text{dist}(s, v)$, и что алгоритмы поиска кратчайшего пути из одной вершины во все остальные устраняют все напряженные ребра. (Если алгоритм Беллмана–Форда обнаруживает отрицательный цикл, то алгоритм Джонсона аварийно останавливается, потому что кратчайшие пути в таком случае не определены.)

Если не существует подходящей вершины s , из которой можно достичь все прочие вершины, то вне зависимости от того, где начинается выполне-

ние алгоритма Беллмана–Форда, некоторые из полученных в результате стоимостей вершин будут бесконечными. Для устранения этой проблемы мы всегда добавляем новую вершину s в исходный граф с ребрами нулевого веса из s в другие вершины, но без ребер, ведущих обратно в s . Такое добавление не изменяет кратчайшие пути между любой парой исходных вершин, потому что не существует путей в s .

Полный псевдокод алгоритма Джонсона показан на рис. 9.2. Время выполнения этого алгоритма определяется в основном обращениями к алгоритму Дейкстры. В частности, мы тратим время $O(VE)$, выполняя алгоритм BellmanFord один раз, время $O(VE \log V)$, запуская алгоритм Dijkstra 1 раз, и время $O(V + E)$, выполняя прочие вычислительные операции. Таким образом, общее время выполнения равно $O(VE \log V) = O(V^3 \log V)$ ⁵. Отрицательные ребра больше нас не замедляют!

```

JohnsonAPSP(V, E, w) :
  «Добавление искусственного источника»
  add a new vertex s
  for every vertex v
    add a new edge s → v
    w(s → v) ← 0
  «Вычисление стоимостей вершин»
  dist[s, ·] ← BellmanFord(V, E, w, s)
  if BellmanFord found a negative cycle
    fail gracefully
  «Изменение весов ребер»
  for every edge u → v ∈ E
    w'(u → v) ← dist[s, u] + w(u → v) - dist[s, v]
  «Вычисление длин кратчайших путей после изменения весов»
  for every vertex u
    dist'[u, ·] ← Dijkstra(V, E, w', u)
  «Вычисление исходных длин кратчайших путей»
  for every vertex u
    for every vertex v
      dist[u, v] ← dist'[u, v] - dist[s, u] + dist[s, v]

```

Рис. 9.2. Алгоритм Джонсона для поиска кратчайших путей между всеми парами вершин в графе

9.5. Динамическое программирование

Задачу поиска кратчайших путей между всеми парами вершин в графе также можно решить напрямую, используя динамическое программирование вместо вызова алгоритма поиска кратчайшего пути из одной вершины во все остальные. Для плотных (dense) графов, где $E = \Omega(V^2)$, метод динамического программирования в итоге позволяет получить алгоритм, который

⁵ ...подразумевая по умолчанию реализацию с двоичной кучей; см. предыдущую сноску.

проще и (немного) быстрее алгоритма Джонсона. В остальной части этой главы я буду подразумевать, что исходный граф не содержит отрицательных циклов.

Как обычно, для алгоритмов динамического программирования сначала требуется рекуррентное выражение. Как и в случае с одним источником, «очевидное» рекурсивное определение,

$$\text{dist}(u, v) = \begin{cases} 0 & , \text{ если } u = v; \\ \min_{x \rightarrow v} (\text{dist}(u, x) + w(x \rightarrow v)) & \text{ иначе.} \end{cases},$$

работает, только если исходным графом является НАГ, а любые направленные циклы вводят это рекуррентное выражение в бесконечный цикл.

Мы можем разорвать этот бесконечный цикл, вводя дополнительный параметр, в точности как это было сделано для алгоритма Беллмана–Форда; пусть $\text{dist}(u, v, \ell)$ обозначает длину кратчайшего пути из u в v , в котором используется не более ℓ ребер. Кратчайший путь между любыми двумя вершинами проходит не более $V - 1$ ребер, поэтому истинное расстояние кратчайшего пути равно $\text{dist}(u, v, V - 1)$. Рекуррентное выражение Беллмана для одного источника сразу же адаптируется к этому условию:

$$\text{dist}(u, v, \ell) = \begin{cases} 0 & , \text{ если } \ell = 0 \text{ и } u = v, \\ \infty & , \text{ если } \ell = 0 \text{ и } u \neq v, \\ \min \left\{ \begin{array}{l} \text{dist}(u, v, \ell - 1) \\ \min_{x \rightarrow v} (\text{dist}(u, x, \ell - 1) + w(x \rightarrow v)) \end{array} \right\} & \text{ иначе.} \end{cases}$$

Преобразование этого рекуррентного выражения в алгоритм динамического программирования выполняется просто. Получившийся в итоге алгоритм выполняется за время $O(V^2E) = O(V^4)$.

```
ShimbelAPSP(V, E, w):
  for all vertices u
    for all vertices v
      if u = v
        dist[u, v, 0] ← 0
      else
        dist[u, v, 0] ← ∞
    for ℓ ← 1 to V - 1
      for all vertices u
        for all vertices v ≠ u
          dist[u, v, ℓ] ← dist[u, v, ℓ - 1]
          for all edges x → v
            if dist[u, v, ℓ] > dist[u, x, ℓ - 1] + w(x → v)
              dist[u, v, ℓ] ← dist[u, x, ℓ - 1] + w(x → v)
```

Набросок этого алгоритма впервые был представлен Альфонсо Шимбелем (Alfonso Shimbel) в 1954 г.⁶ Как и в формулировке Беллмана алгоритма Беллмана–Форда, нам не нужен внутренний цикл по вершинам v или индекс итерации ℓ . Улучшенный алгоритм показан ниже.

```

AllPairsBellmanFord(V, E, w):
  for all vertices u
    for all vertices v
      if u = v
        dist[u, v] ← 0
      else
        dist[u, v] ← ∞
  for ℓ ← 1 to V - 1
    for all vertices u
      for all edges x → v
        if dist[u, v] > dist[u, x] + w(x → v)
          dist[u, v] ← dist[u, x] + w(x → v)

```

С учетом того, как мы к нему пришли, нас не должно удивлять, что полученный в итоге алгоритм в точности тот же самый, что и V накладывающихся выполнений алгоритма Беллмана–Форда, каждое с другой вершиной-источником. В частности, для всех вершин u и v после ℓ -й итерации основного цикла for $\text{dist}[u, v]$ не превышает длину кратчайшего пути из u в v , содержащего не более ℓ ребер.

9.6. Разделяй и властвуй

Но мы можем применить более существенное усовершенствование, предложенное Майклом Фишером (Michael Fischer) и Альбертом Майером (Albert Meyer) в 1971 г. Рекуррентное выражение Беллмана разделяет кратчайший путь на чуть более короткий путь и отдельное ребро при рассмотрении всех возможных предков целевой вершины. Вместо этого разделим кратчайшие пути на два более коротких кратчайших пути в средней вершине. Такой подход дает нам другое рекуррентное выражение для той же функции $\text{dist}(u, v, \ell)$. Здесь мы должны остановиться в базовом случае $\ell = 1$ вместо $\ell = 0$, потому что путь не более чем с одним ребром не имеет «средней» вершины. Для небольшого упрощения этого рекуррентного выражения определим $w(v \rightarrow v) = 0$ для каждой вершины v :

$$\text{dist}(u, v, \ell) = \begin{cases} w(u \rightarrow v) & , \text{ если } \ell = 1; \\ \min_x (\text{dist}(u, x, \ell/2) + \text{dist}(x, v, \ell/2)) & \text{ иначе.} \end{cases}$$

⁶ Шимбель предполагал, что входными данными являлась полная матрица расстояний $V \times V$, поэтому его первоначальный алгоритм в действительности выполнялся за время $O(V^4)$ вне зависимости от того, сколько ребер содержал граф.

В указанной форме это рекуррентное выражение работает, только если ℓ является степенью 2, так как иначе мы могли бы попытаться найти кратчайший путь из (не более чем) дробного числа ребер. Но в действительности это не проблема, $\text{dist}(u, v, \ell)$ – это истинное расстояние кратчайшего пути из u в v для всех $\ell \geq V - 1$, в частности, мы можем использовать $\ell = \lceil 2 \lg V \rceil < 2V$.

И в этом случае решение методом динамического программирования реализуется просто. Еще до выписывания алгоритма мы можем сказать, что его время выполнения равно $O(V^3 \log V)$, – мы должны рассмотреть V возможных значений u, v и x , но только $\lceil \lg V \rceil$ возможных значений ℓ . В приведенном ниже псевдокоде алгоритма Фишера и Майера элемент массива $\text{dist}[u, v, i]$ хранит значение $\text{dist}(u, v, 2^i)$.

```

FischerMeyerAPSP(V, E, w):
  for all vertices u
    for all vertices v
      dist[u, v, 0] ← w(u → v)
  for i ← 1 to lg V                                <<ℓ = 2i>>
    for all vertices u
      for all vertices v
        dist[u, v, i] ← ∞
        for all vertices x
          if dist[u, v, i] > dist[u, x, i - 1] + dist[x, v, i - 1]
            dist[u, v, i] ← dist[u, x, i - 1] + dist[x, v, i - 1]

```

В отличие от наших предыдущих алгоритмов FischerMeyerAPSP – это не то же самое, что V обращений к какому-либо алгоритму поиска кратчайшего пути из одной вершины во все остальные. В частности, самый внутренний цикл не просто ослабляет напряжение ребер. Как бы то ни было, мы все еще можем удалить последнее измерение в таблице, используя $\text{dist}[u, v]$ везде вместо $\text{dist}[u, v, i]$, как это было сделано в алгоритме Беллмана–Форда и в нашем предыдущем алгоритме динамического программирования, – это сокращает потребление памяти с $O(V^3)$ до $O(V^2)$. Этот более отшлифованный алгоритм был описан Майклом Лейзореком (Michael Leyzorek) и др. в 1957 г. в той же статье, где эти авторы описывают алгоритм Дейкстры.

```

LeyzorekAPSP(V, E, w):
  for all vertices u
    for all vertices v
      dist[u, v] ← w(u → v)
  for i ← 1 to lg V                                <<ℓ = 2i>>
    for all vertices u
      for all vertices v
        for all vertices x
          if dist[u, v] > dist[u, x] + dist[x, v]
            dist[u, v] ← dist[u, x] + dist[x, v]

```

9.7. Странное умножение матриц

Существует весьма тесная связь (впервые обнаруженная Шимбелем и позже независимо Беллманом) между вычислением кратчайших путей в направленном графе и вычислением степеней квадратных матриц. Сравните приведенный ниже алгоритм возведения в квадрат матрицы A размером $n \times n$ с внутренним циклом алгоритма FischerMeyerAPSP. (Я немного изменил нотацию во втором алгоритме, чтобы схожесть была более явной.)

```
MatrixSquare(A):
  for i ← 1 to n
    for j ← 1 to n
      A'[i, j] ← 0
      for k ← 1 to n
        A'[i, j] ← A'[i, j] + A[i, k] · A[k, j]
```

```
FischerMeyerInnerLoop(D):
  for all vertices u
    for all vertices v
      D'[u, v] ← ∞
      for all vertices x
        D'[u, v] ← min{D'[u, v], D[u, x] + D[x, v]}
```

Единственное различие между этими двумя алгоритмами состоит в том, что второй алгоритм использует сложение вместо умножения и функцию минимизации вместо сложения. По этой причине внутренний цикл в алгоритме поиска кратчайшего пути иногда называют «минимум-плюс» (min-plus), или «умножением расстояний» (distance), или «странным» (funny) умножением матриц.

Наш более медленный алгоритм ShimbelAPSP является стандартным итеративным алгоритмом вычисления $(V-1)$ -й минимум-плюс-степени матрицы весов w . Первая группа циклов формирует единичную матрицу (матрицу тождественного преобразования) минимум-плюс с нулями на главной диагонали и ∞ в остальных позициях, а каждая итерация второго основного цикла вычисляет следующую минимум-плюс-степень. FischerMeyerAPSP заменяет этот итеративный метод вычисления степеней на повторяющееся возведение в квадрат, точно так же, как мы видели в конце главы 1. Здесь мы снова наблюдаем влияние древних египетских вязателей веревок (*ἀρπυδοπλάται*).

Существуют более быстрые алгоритмы «разделяй и властвуй» для (стандартного) умножения матриц, подобных алгоритму Карацубы для умножения целых чисел. Первый такой алгоритм, описанный Фолькером Штрассеном (Volker Strassen) в 1969 г., сводит задачу умножения двух матриц $n \times n$ к семи вариантам умножения двух матриц $n/2 \times n/2$. Алгоритм Штрассена выполняется за время $O(n^{\lg 7}) = O(n^{2.807355})$. Алгоритм Штрассена многократно

совершенствовался в течение последних пятидесяти лет, и к 2018 г. самый быстрый известный алгоритм умножения матриц выполняется за время $O(n^{2.372864})^7$. К сожалению, *все* эти более быстрые алгоритмы используют вычитание, и какого-либо «странного» эквивалента вычитанию нет. (Какая операция является обратной операции $\min$?) Поэтому, по крайней мере, для общих графов не существует очевидного способа ускорения внутреннего цикла в наших алгоритмах динамического программирования.

Но «отсутствие очевидного» не означает «невозможно»! В действительности существует несколько существенно более быстрых алгоритмов для *особых случаев* задачи поиска кратчайших путей между всеми парами вершин в графе. Одним из наилучших является простой рандомизированный алгоритм, разработанный в 1991 г. Цви Галилом (Zvi Galil) и Одедом Марголитом (Oded Margalit) и упрощенный в 1992 г. Раймундом Зайделем (Raimund Seidel). Алгоритм вычисляет *длины* кратчайших путей между всеми парами вершин в *невзвешенных, ненаправленных* графах за *ожидаемое* время $O(M(V) \log V)$, где $M(n) = O(n^{2.372864})$ – время, требуемое для (серьезного) умножения двух целочисленных матриц $n \times n$.⁸ Методика Галила, Маргалита и Зайделя была расширена для детерминированного вычисления действительных кратчайших путей в направленных графах с малыми целочисленными весами ребер за строго субкубическое время.

С другой стороны, несмотря на значительный прогресс в области задач с малыми целочисленными весами, никто не знает, как вычислить кратчайшие пути между всеми парами вершин в графе для общих весов ребер за время $O(V^{2.999999})$ для любого числа девяток. Более того, существуют некоторые свидетельства того, что такой алгоритм невозможен. Возможно, «отсутствие очевидного» все-таки в итоге означает «невозможно».

9.8. Алгоритм (Клини–Роя–)Флойда–Уоршелла(–Ингермана)

Наш быстрый алгоритм динамического программирования в наихудшем случае отстает на множитель $O(\log V)$ от стандартной реализации алгоритма Джонсона. Другая формулировка задачи поиска кратчайших путей, которая удаляет этот логарифмический множитель, была предложена дважды в 1962 г.: сначала Робертом Флойдом (Robert Floyd), затем независимо от него Питером Ингерманом (Peter Ingerman) – оба немного обобщили алгоритм Стивена Уоршелла (Stephen Warshall), опубликованный ранее в

⁷ Определение минимального времени, требуемого для умножения двух произвольных матриц $n \times n$, – давно существующая нерешенная задача. Многие верят, что существует пока еще не найденный алгоритм, который выполняется за время $O(n^{2+\epsilon})$ для любого $\epsilon > 0$ или даже за время $O(n^2)$.

⁸ Raimund Seidel. «On the all-pairs-shortest-path problem in unweighted undirected graphs». Journal of Computer and System Sciences, 51(3):400–403, 1995. Это одна из немногих статей об алгоритмах, где (по крайней мере, в версии для конференции 1992 г.) алгоритм полностью описан и проанализирован в аннотации к этой статье; см. также: Noga Alon, Zvi Galil, Oded Margalit*. «On the exponent of the all pairs shortest path problem». Journal of Computer and System Sciences 54(2):255–262, 1997.

том же году. В действительности алгоритм Уоршелла еще раньше был разработан Бернардом Роем (Bernard Roy) в 1959 г., а внутренний рекурсивный шаблон был использован Стивенем Клини (Stephen Kleene)⁹ в 1951 г.

Подход Уоршелла (а также Роя и Клини) заключался в использовании другого третьего параметра в рекуррентном выражении динамического программирования. Вместо рассмотрения путей с ограниченным числом ребер они рассматривали пути, которые могут проходить только через конкретные вершины. Здесь «проходить через» означает «входить и выходить»; например, путь $w \rightarrow x \rightarrow y \rightarrow z$ начинается в вершине w , проходит через x и y , затем завершается в z .

Перенумеруем вершины произвольными числами от 1 до V . Для каждой пары вершин u и v и каждого целого числа r мы определяем путь $\pi(u, v, r)$ следующим образом:

$\pi(u, v, r)$ – это кратчайший путь (если такой существует) из u в v , который проходит только через вершины с номерами, не превышающими r .

В частности, $\pi(u, v, V)$ – это истинный кратчайший путь из u в v . Клини, Рой и Уоршелл заметили, что эти пути имеют простую рекурсивную структуру.

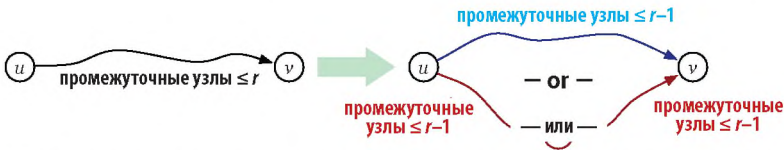


Рис. 9.3. Рекурсивная структура ограниченного кратчайшего пути $\pi(u, v, r)$

- Путь $\pi(u, v, 0)$ не может проходить через какие-либо промежуточные вершины, поэтому он обязательно должен быть ребром (если такое существует) из u в v .
- Для любого целого числа $r > 0$ кратчайший путь $\pi(u, v, r)$ либо проходит через вершину r , либо не проходит.
 - ♦ Если $\pi(u, v, r)$ проходит через вершину r , то он состоит из участка пути из u в r , за которым следует участок пути из r в v . Оба эти участка пути проходят только через вершины, номера которых не превышают $r - 1$. Кроме того, эти участки пути настолько коротки, насколько это возможно при заданном здесь ограничении. Следовательно, этими двумя участками пути обязательно должны быть $\pi(u, r, r - 1)$ и $\pi(r, v, r - 1)$.

⁹ Фамилия Клеене правильно произносится «клеини» (он сам именно так ее произносил), а не «клин», или «клини», или «клайна», или «димаджжо». [В русскоязычной литературе укоренилось неверное написание и произношение «Клини» еще со времен СССР в связи с изданием переводов его книг именно под такой фамилией. Здесь эта «традиция» сохранена. – Прим. перев.] Клини описал индуктивное доказательство того, что каждому конечному автомату соответствует эквивалентное регулярное выражение. Индуктивный шаблон Клини по существу идентичен рекуррентному выражению Флойда–Уоршелла.

- ♦ С другой стороны, если путь $\pi(u, v, r)$ не проходит через вершину r , то он проходит только через вершины с номерами, не превышающими $r - 1$, и это обязательно должен быть кратчайший путь с учетом заданного здесь ограничения. Так что в этом случае мы непременно получаем $\pi(u, v, r) = \pi(u, v, r - 1)$.

Теперь пусть $\text{dist}(u, v, r)$ обозначает длину пути $\pi(u, v, r)$. Рекурсивная структура пути $\pi(u, v, r)$ сразу же позволяет вывести следующее рекуррентное выражение:

$$\text{dist}(u, v, r) = \begin{cases} w(u \rightarrow v) & , \text{ если } r = 0, \\ \min \left\{ \begin{array}{l} \text{dist}(u, v, r - 1) \\ \min(\text{dist}(u, r, r - 1) + \text{dist}(r, v, r - 1)) \end{array} \right\} & \text{ иначе.} \end{cases}$$

Наша цель – вычисление $\text{dist}(u, v, V)$ для всех вершин u и v . И в этом случае приведенное выше рекуррентное выражение можно вычислить с помощью простого алгоритма динамического программирования за время $O(V^3)$.

```

KleeneAPSP(V, E, w):
  for all vertices u
    for all vertices v
      dist[u, v, 0] ← w(u → v)
  for r ← 1 to V
    for all vertices u
      for all vertices v
        if dist[u, v, r - 1] < dist[u, r, r - 1] + dist[r, v, r - 1]
          dist[u, v, r] ← dist[u, v, r - 1]
        else
          dist[u, v, r] ← dist[u, r, r - 1] + dist[r, v, r - 1]

```

Как и во всех наших предыдущих алгоритмах динамического программирования для поиска кратчайших путей, мы можем упростить алгоритм KleeneAPSP, удалив третье измерение из таблицы мемоизации. Кроме того, поскольку мы выбираем нумерацию вершин произвольно, нет причины для явной ссылки на них в псевдокоде. В итоге мы приходим к усовершенствованию Флойда для алгоритма Уоршелла:

```

FloydWarshall(V, E, w):
  for all vertices u
    for all vertices v
      dist[u, v] ← w(u → v)
  for all vertices r
    for all vertices u
      for all vertices v
        if dist[u, v] > dist[u, r] + dist[r, v]
          dist[u, v] ← dist[u, r] + dist[r, v]

```

Интересно сравнить алгоритм FloydWarshall с нашим предыдущим, не-много более медленным алгоритмом динамического программирования LeyzorekAPSP. Вместо $O(\log V)$ проходов через все тройки вершин FloydWarshall требует единственного прохода, но только потому, что использует другой порядок вложенности трех циклов!

Упражнения

1. (a) Описать модификацию алгоритма LeyzorekAPSP, которая возвращает массив указателей на предшественников в дополнение к массиву длин кратчайших путей, но продолжает выполняться за время $O(V^3 \log V)$.
 (b) Описать модификацию алгоритма FloydWarshall, которая возвращает массив указателей на предшественников в дополнение к массиву длин кратчайших путей, но продолжает выполняться за время $O(V^3)$.
2. Все алгоритмы, рассмотренные в этой главе, не работают, если граф содержит отрицательный цикл. Алгоритм Джонсона обнаруживает отрицательный цикл на этапе инициализации (через алгоритм Беллмана–Форда) и завершается аварийно. Алгоритмы динамического программирования просто возвращают некорректные результаты. Но все эти алгоритмы можно модифицировать так, чтобы они возвращали корректные расстояния кратчайших путей даже при наличии отрицательного цикла. Более конкретно – для вершин u и v :
 - если u не может достичь v , то алгоритм должен вернуть $dist[u, v] = \infty$;
 - если u может достичь отрицательный цикл, из которого можно достичь v , то алгоритм должен вернуть $dist[u, v] = -\infty$;
 - иначе существует кратчайший путь из u в v , поэтому алгоритм должен вернуть его длину.
 (a) Описать, как нужно модифицировать алгоритм Джонсона, чтобы он возвращал корректные длины кратчайших путей, даже если граф содержит отрицательные циклы.
 (b) Описать, как нужно модифицировать алгоритм LeyzorekAPSP, чтобы он возвращал корректные длины кратчайших путей, даже если граф содержит отрицательные циклы.
 (c) Описать, как нужно модифицировать алгоритм Флойда–Уоршелла, чтобы он возвращал корректные длины кратчайших путей, даже если граф содержит отрицательные циклы.
3. Алгоритмы, описанные в этой главе, также можно модифицировать, чтобы они возвращали явное описание некоторого отрицательного цикла в исходном графе G , если такой цикл существует, а не только

сообщали о наличии или отсутствии отрицательного цикла в графе G .

- (а) Описать, как нужно модифицировать алгоритм Джонсона, чтобы он возвращал либо массив всех длин кратчайших путей, либо отрицательный цикл.
- (б) Описать, как нужно модифицировать алгоритм LeyzorekAPSP, чтобы он возвращал либо массив всех длин кратчайших путей, либо отрицательный цикл.
- (с) Описать, как нужно модифицировать алгоритм Флойда–Уоршелла, чтобы он возвращал либо массив всех длин кратчайших путей, либо отрицательный цикл.

Во всех случаях, если исходный граф содержит более одного отрицательного цикла, алгоритмы могут произвольно выбрать один из них.

4. Пусть $G = (V, E)$ – направленный граф с взвешенными ребрами. Веса ребер могут быть положительными, отрицательными или нулевыми, но отрицательные циклы отсутствуют.

- (а) Описать эффективный алгоритм, который либо находит цикл нулевой длины в графе G , либо корректно сообщает, что такого цикла не существует.
- (б) Описать эффективный алгоритм, который создает подграф H из графа G со следующими свойствами:
 - каждая вершина графа G является вершиной подграфа H ;
 - каждый направленный цикл в H имеет длину 0;
 - каждый направленный цикл длиной 0 в G также является циклом в H .

В частности, если в графе G нет циклов нулевой длины, то в подграфе H нет ребер.

5. Пусть $G = (V, E)$ – направленный граф с взвешенными ребрами. Веса ребер могут быть положительными, отрицательными или нулевыми. Предположим, что вершины G разделены на k непересекающихся подмножеств $V_1, V_2, \dots, V_k$, т. е. каждая вершина G принадлежит ровно одному подмножеству V_i . Для каждого i и j пусть $\delta(i, j)$ обозначает минимальную длину кратчайшего пути между вершинами в V_i и вершинами в V_j :

$$\delta(i, j) = \min\{\text{dist}(v_i, v_j) \mid v_i \in V_i \text{ и } v_j \in V_j\}.$$

Описать алгоритм вычисления $\delta(i, j)$ для всех i и j . Для получения максимальной оценки алгоритм должен выполняться за время $O(VE + kV \log V)$.

6. В этой задаче требуется узнать, как вы, да именно вы, можете получить работу на Уолл-стрит и вызвать всеобщий экономический коллапс! Биржевой арбитраж – это схема зарабатывания денег, которая использует преимущество в разнице при обмене валют. В частности, предположим, что 1 долл. США стоит 120 японских йен, одна йена стоит 0.01 евро, а один евро стоит 1.2 долл. США. Далее трейдер, начинающий с 1 долл., может сконвертировать свои деньги из долларов в йены, затем из йен в евро, наконец, из евро обратно в доллары, получив в итоге 1.44 долл.! Цикл обмена валют $\$ \rightarrow \text{¥} \rightarrow \text{€} \rightarrow \$$ называется *циклом арбитражных операций*. Разумеется, поиск и использование циклов арбитражных операций раньше корректировки курсов валют требует экстремально быстрых алгоритмов.

Предположим, что n различных валют обмениваются на вашей валютной бирже. Вам задана матрица $Exch[1..n, 1..n]$ курсов обмена между каждой парой валют: для каждого i и j единицу валюты i можно обменять на $Exch[i, j]$ единиц валюты j . (Здесь не предполагается, что $Exch[i, j] \cdot Exch[j, i] = 1$.)

- Описать алгоритм, который возвращает массив $MaxAmt[1..n]$, где $MaxAmt[i]$ – максимальная сумма валюты i , которую вы можете получить при обмене, начав с одной единицы валюты 1, предполагая отсутствие циклов арбитражных операций.
 - Описать алгоритм, определяющий, создает ли цикл арбитражных операций заданная матрица обменных курсов валют.
 - Изменить алгоритм из части (b) так, чтобы он возвращал цикл арбитражных операций, если такой цикл существует.
7. Морти нужно извлечь стабилизированный отвес (plumbus) из лабиринта Клакспайр (Clackspire Labyrinth). Морти должен войти в лабиринт, используя межпространственный портал-пушку Рика, пройти лабиринт до отвеса, затем перенести отвес через лабиринт к флибу (fibleb), чтобы стабилизировать его, наконец, доставить стабилизированный отвес обратно к входному portalу, чтобы вернуться домой. Отвесы стабилизируются флибовым соком, который любой флиб выделяет сразу после его выдергивания из лунки. Нестабилизованный отвес взорвется, если нести его больше 137 флинков (flink) от исходного места хранения. Запах в лабиринте Клакспайр отвратительный, поэтому Морти нужно потратить как можно меньше времени на пребывание в лабиринте.

Рик дал Морти подробную карту лабиринта Клакспайр, которая состоит из направленного графа $G = (V, E)$ с неотрицательными весами ребер (обозначающими расстояния во флинках) вместе с двумя непересекающимися подмножествами $P \subset V$ и $F \subset V$, обозначающие соответственно места хранения отвесов и лунки флибов. Морти не-

обходимо идентифицировать начальную вершину s , место хранения отвеса $p \in P$ и лунку флиба $f \in F$ такие, чтобы длина кратчайшего пути из p в f не превышала 137 флинков, а длина кратчайшего маршрута $s \rightsquigarrow p \rightsquigarrow f \rightsquigarrow s$ была самой короткой.

Описать и проанализировать алго(burp)ритм re(burp)шения задачи Морти. Вы можете предположить, что Морти действительно может добиться успеха.

8. Пусть $G = (V, E)$ – направленный граф с взвешенными ребрами. Веса ребер могут быть положительными, отрицательными или нулевыми.
 - (а) Как мы могли бы удалить произвольную вершину v из этого графа без изменения длины кратчайшего пути между любой другой парой вершин? Описать алгоритм, который создает направленный граф $G' = (V \setminus \{v\}, E')$ со взвешенными ребрами, такой, что длина кратчайшего пути между любой парой вершин в G' равна длине кратчайшего пути между теми же двумя вершинами в G , за время $O(V^2)$.
 - (б) Теперь предположим, что у нас уже вычислены все длины кратчайших путей в G' . Описать алгоритм вычисления длин кратчайших путей в исходном графе G из v в каждую другую вершину и из каждой другой вершины в v , и все это за время $O(V^2)$.
 - (с) Объединить части (а) и (б) в еще один алгоритм поиска кратчайшего пути из одной вершины во все остальные, который выполняется за время $O(V^3)$. (Итоговый алгоритм почти тот же, что и алгоритм Флойда–Уоршелла.)
9. Предположим, что A и B – булевы матрицы $n \times n$. Булево, оно же И-ИЛИ-произведение A и B , является матрицей C $n \times n$, определяемой следующим образом:

$$C[i, j] := \bigvee_k (A[i, k] \wedge B[k, j]).$$

- (а) Свести булево умножение матриц к минимум-плюс-умножению матриц. То есть при заданной подпрограмме `MinPlusMultiply`, которая вычисляет минимум-плюс-произведение двух матриц $n \times n$ за время $T(n)$, описать и проанализировать алгоритм `BooleanMatrixMultiply`, выполняющий умножение двух булевых матриц за время $O(T(n))$.
- (б) Свести умножение булевых матриц к стандартному умножению матриц. То есть при заданной подпрограмме `MatrixMultiply`,

которая вычисляет стандартное произведение двух матриц $n \times n$ за время $T(n)$, описать и проанализировать алгоритм `BooleanMatrixMultiply`, выполняющий умножение двух булевых матриц за время $O(T(n))$.

10. Транзитивное замыкание (transitive closure) направленного графа G содержит ребро $u \rightarrow v$ тогда и только тогда, когда существует направленный путь из u в v в графе G . Для этой задачи предположим, что мы можем умножить две булевы матрицы $n \times n$ за время $O(n^\omega)$ для некоторой константы $2 \leq \omega \leq 3$. (Задача 9(b) указывает, что $\omega \leq 2.372864$.)
 - (a) Описать алгоритм построения транзитивного замыкания направленного графа с n вершинами за время $O(n^\omega \log n)$.
 - (b) Теперь предположим, что G – направленный ациклический граф. Описать алгоритм построения транзитивного замыкания графа G за время $O(n^\omega)$. [Подсказка: делайте то, что всегда делаете с НАГами, затем примените стратегию «разделяй и властвуй». Используйте тот факт, что $\omega \geq 2$.]
 - (c) Наконец, описать алгоритм построения транзитивного замыкания произвольного направленного графа за время $O(n^\omega)$. [Подсказка: делайте то, что всегда делаете, чтобы превратить произвольный направленный граф в НАГ.]
 - (d) Теперь рассмотрим задачу, обратную предыдущему сведению. Пусть задана подпрограмма `TransitiveClosure`, вычисляющая транзитивное замыкание направленного графа с n вершинами за время $O(n^\omega)$ для некоторой константы $2 \leq \omega \leq 3$. Описать и проанализировать алгоритм умножения булевых матриц, выполняемый за время $O(n^\omega)$.
11. Доказать, что приведенный ниже рекурсивный алгоритм корректно вычисляет длины кратчайших путей между всеми парами вершин за время $O(n^3)$. Для упрощения можно предположить, что n является степенью 2. Как обычно, массив D передается по ссылке во вспомогательную функцию `RecAPSP`. [Подсказка: это смешанная версия алгоритма Флойда–Уоршелла со значительным улучшением поведения кеша.¹⁰]

¹⁰ Joon-Sang Park, Michael Penner, and Viktor k. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Trans. Parallel and Distributed Systems* 15(9):769–782, 2004. Для существенного обобщения к более широкому классу задач динамического программирования см.: Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-oblivious dynamic programming. *Proc. 17th SODA* 591–600, 2006.

RecursiveAPSP(V,E,w):

```

n ← |V|
for i ← 1 to n
  for j ← 1 to n
    if i = j
      D[i,j] ← 0
    if i → j ∈ E
      D[i,j] ← w(i → j)
    else
      D[i,j] ← ∞
  RecAPSP(D, n, 1, 1, 1)
return D[1..n, 1..n]
```

RecAPSP(D, n, i, j, k):

```

if n = 1
  D[i,j] ← min{D[i,j], D[i,k] + D[j,k]}
else
  m ← n/2
  RecAPSP(D, n/2, i, j, k)
  RecAPSP(D, n/2, i, j, k + m)
  RecAPSP(D, n/2, i, j + m, k)
  RecAPSP(D, n/2, i, j + m, k + m)
  RecAPSP(D, n/2, i + m, j, k)
  RecAPSP(D, n/2, i + m, j, k + m)
  RecAPSP(D, n/2, i + m, j + m, k)
  RecAPSP(D, n/2, i + m, j + m, k + m)
```

- ♥12. Пусть $G = (V, E)$ – ненаправленный, невзвешенный, связный граф с n вершинами, представленный матрицей смежности $A[1..n, 1..n]$. В этой задаче мы выводим субкубический алгоритм Зайделя для вычисления матрицы $n \times n$ $D[1..n, 1..n]$ длин кратчайших путей в G , используя быстрое умножение матриц. Предположим, что у нас есть подпрограмма **MatrixMultiply**, вычисляющая стандартное произведение двух матриц $n \times n$ за время $O(n^\omega)$ для некоторой неизвестной константы $\omega \geq 2$.

- Пусть G^2 обозначает граф с теми же вершинами, что и G , где две вершины соединены ребром, если и только если они соединены путем с длиной не больше 2 в графе G . Описать алгоритм вычисления матрицы смежности графа G^2 с использованием одного вызова подпрограммы **MatrixMultiply** и дополнительного времени $O(n^2)$.
 - Предположим, мы обнаружили, что G^2 – полный граф. Описать алгоритм вычисления матрицы D длин кратчайших путей в G за дополнительное время $O(n^2)$.
 - Предположим, что мы рекурсивно вычислили матрицу D^2 длин кратчайших путей в G^2 . Доказать, что длина кратчайшего пути в графе G из узла i в узел j равно либо $2 \cdot D^2[i, j]$, либо $2 \cdot D^2[i, j] - 1$.
 - Теперь предположим, что G^2 не является полным графом. Пусть $X = D^2 \cdot A$ и пусть $\deg(i)$ обозначает степень вершины i в исходном графе G . Доказать, что длина кратчайшего пути из узла i в узел j в графе G равна $2 \cdot D^2[i, j]$, если и только если $X[i, j] \geq D^2[i, j] \cdot \deg(i)$.
 - Описать алгоритм вычисления матрицы D длин кратчайших путей в графе G за время $O(n^\omega \log n)$.
13. Гидеон Юваль (Gideon Yuval) предложил следующее сведение задачи умножения матриц минимум-плюс к стандартному умножению

матриц в 1976 г. Предположим, что заданы две целочисленные $n \times n$ матрицы A и B , элементы которых находятся между 0 и M , и нам необходимо вычислить их произведение минимум-плюс – матрицу C , определяемую выражением

$$C[i, k] = \min_j (A[i, j] + B[j, k])$$

для всех индексов i и k . Определим две новые матрицы $n \times n$ A' и B' , где:

$$A'[i, j] = n^{M-A[i, j]} \quad \text{и} \quad B'[i, j] = n^{M-B[i, j]}.$$

Наконец, пусть C' – (стандартное) произведение матриц A' и B' , определяемое выражением $C'[i, k] = \sum_j A'[i, j] \cdot B'[j, k]$.

- (a) Описать алгоритм создания A' из A с использованием только стандартных целочисленных арифметических операций $(+, -, \times)$.
- (b) Описать алгоритм извлечения минимум-плюс-произведения C из C' с использованием только стандартных целочисленных арифметических операций $(+, -, \times)$ ¹¹.
- (c) Предположим, что мы можем вычислить стандартное произведение двух целочисленных матриц $n \times n$, используя $O(n^\omega)$ арифметических операций, для некоторой константы $2 \leq \omega \leq 3$. Сколько арифметических операций требуется алгоритму Юваля для вычисления минимум-плюс-произведения C ?
- (d) Пусть задана одна целочисленная $n \times n$ матрица A . Сколько арифметических операций требуется для вычисления n -й «странной» степени A с использованием алгоритма Юваля? (Напомню, что если A – взвешенная матрица смежности графа, то n -я «странная» степень A – это матрица длин кратчайших путей.)
- (e) Почему алгоритм Юваля нельзя применить как алгоритм поиска кратчайших путей между всеми парами вершин, который быстрее алгоритма Флойда–Уоршелла при произвольных весах ребер? В чем подвох?

¹¹ В частности, нельзя использовать логарифмы, или деление, или функцию округления в меньшую сторону $\lfloor x \rfloor$. Поверьте мне – это банка с червями, которую вы не хотите открывать.

Глава 10

.....

Максимальные потоки и наименьшие разрезы

Процесс нельзя понять посредством его прекращения. Понимание должно двигаться вместе с процессом, слиться с его потоком и течь вместе с ним.

— Первый закон ментата, Френк Херберт (Frank Herbert), «Дюна» («Dune») (1965 г.)

Вопреки ожиданию поток обычно возникает не в моменты расслабленности во время досуга или развлечений, а наоборот — когда мы активно вовлечены в трудное дело, в задачу, которая напрягает наши умственные и физические способности... Трудно достичь потока без усилий. Поток — это не «пустая трата времени».

— Михай Чиксентмихайи (Mihaly Csikszentmihályi), «Поток: психология оптимального переживания» («Flow: The Psychology of Optimal Experience») (1990 г.)

Знать путь и пройти его — не одно и то же.

— Морфейс (Morpheus) [Лоренс Фишберн (Laurence Fishburne)],
«Матрица» («The Matrix») (1999 г.)

В середине 1950-х гг. исследователь из ВВС США Теодор Е. Харрис (Theodor E. Harris) и генерал армии США в отставке Фрэнк С. Росс (Frank S. Ross) написали секретный отчет об исследовании железнодорожной сети, соединяющей Советский Союз со странами-союзниками в Восточной Европе. Модель этой сети представляла собой граф с 44 вершинами, представляющими географические регионы, и 105 ребрами, представляющими связи между регионами в железнодорожной сети. Каждое ребро имело вес, обозначающий скорость, с которой материальные средства могли быть доставлены из одного региона в другой. По существу, методом проб и ошибок авторы отчета определили максимальный объем материальных средств, которые можно было бы переместить из России в Европу, а также самый дешевый способ разорвать эту сеть, удаляя связи (или, в менее абстрактных терминах, разрушая железнодорожные пути), которые авторы называли «узким

местом» (the bottleneck). Этот отчет, включающий схему железнодорожной сети, показанную на рис. 10.1, был рассекречен только в 1999 г.¹

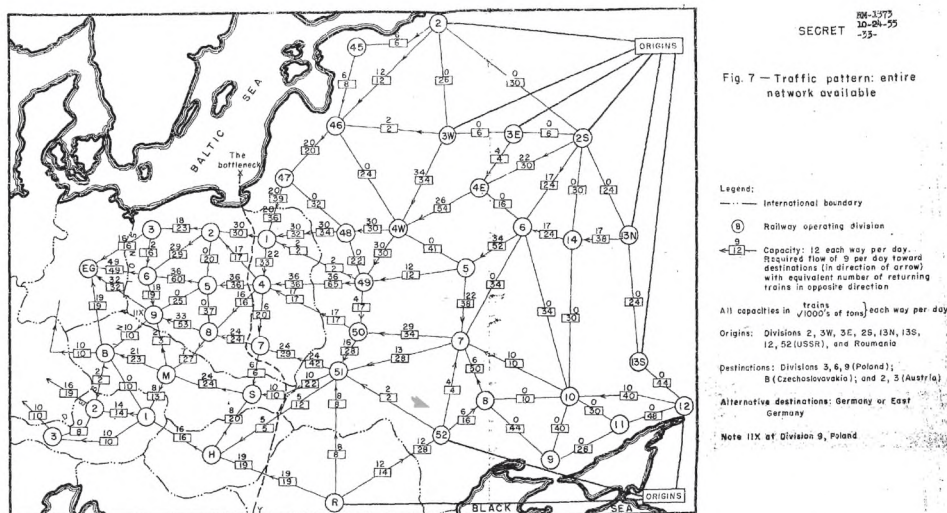


Рис. 10.1. Карта Харриса и Росса железнодорожной сети в странах Варшавского договора. (Источник: T[homas] E. Harris and F[rank] S. Ross. Fundamentals of a method for evaluating rail net capacities. The RAND Corporation, Research Memorandum RM-1517, October 24, 1955. United States Government work in the public domain. <http://www.dtic.mil/dtic/tr/fulltext/u2/093458.pdf>)

Это одно из первых документально зафиксированных приложений задач поиска *максимального потока* и *наименьшего (минимального) разреза* графа. Для обеих задач входными данными является направленный граф $G = (V, E)$ с двумя особыми вершинами s и t , называемыми соответственно источником и целью. Как и в предыдущих главах, я буду писать $u \rightarrow v$ для обозначения направленного ребра из вершины u в вершину v . Интуитивно задача поиска максимального потока предлагает найти максимальную скорость, с которой некоторый ресурс может быть перемещен из s в t . Задача поиска наименьшего разреза предлагает найти минимальное повреждение (разрыв), необходимое для отделения s от t .

10.1. Потoki

(s, t) -*поток* (или просто *поток (flow)*), если исходная (*source*) и целевая (*target*) вершины понятны из контекста) — это функция $f: E \rightarrow \mathbb{R}$, которая соответствует следующему *ограничению сохранения* (conservation constraint) в каждой вершине v , возможно, исключая s и t :

¹ Я изучал эту историю по великолепному обзору Александра Шрейвера (Alexander Schrijver) «On the history of combinatorial optimization (till 1960)». Отчет Харриса–Росса был рассекречен по запросу Шрейвера. Форд (Ford) и Фалкерсон (Fulkerson) (с которыми мы скоро встретимся) приписали Харрису авторство формулировки задачи максимального потока, хотя точная хронология несколько запутана, Харрис и Росс благодарили Джорджа Данцига (George Dantzig) «за помощь в формулировке задачи».

$$\sum_u f(u \rightarrow v) = \sum_w f(v \rightarrow w).$$

В английском языке общий поток в v равен общему потоку из v . Чтобы сохранить нотацию простой, мы определяем $f(u \rightarrow v) = 0$, если не существует ребра $u \rightarrow v$ в графе. Величина (value) потока f , обозначаемая $|f|$, – это общий сетевой поток из вершины-источника s :

$$|f| := \sum_w f(s \rightarrow w) - \sum_u f(u \rightarrow s).$$

Нетрудно доказать, что величина $|f|$ также равна общему сетевому потоку в целевую вершину t , как показано ниже. Для упрощения нотации пусть $\partial f(v)$ обозначает общий сетевой поток из любой вершины v :

$$\partial f(v) := \sum_u f(u \rightarrow v) - \sum_w f(v \rightarrow w).$$

Ограничение сохранения предполагает, что $\partial f(v) = 0$ для любой вершины v , исключая s и t , поэтому:

$$\sum_v \partial f(v) = \partial f(s) + \partial f(t).$$

С другой стороны, любой поток, который выходит из одной вершины, обязательно должен входить в другую вершину, поэтому мы непременно имеем $\sum_v \partial f(v) = 0$. Из этого сразу же следует, что $|f| = \partial f(s) = -\partial f(t)$.

Теперь предположим, что у нас есть другая функция $c : E \rightarrow \mathbb{R}_{\geq 0}$, которая присваивает неотрицательную пропускную способность (capacity) $c(e)$ каждому ребру e . Мы говорим, что поток f является *достижимым* (feasible) (с учетом c), если $0 \leq f(e) \leq c(e)$ для каждого ребра e . В основном мы рассматриваем только выполнимые потоки с учетом некоторой фиксированной функции пропускной способности c . Мы говорим, что поток f *насыщает* (saturates) ребро e , если $f(e) = c(e)$, и *избегает* (avoids) ребра e , если $f(e) = 0$. Задача *поиска максимального потока* (maximum flow problem) – это вычисление достижимого (s, t) -потока в заданном направленном графе с заданной функцией пропускной способности, величина которого максимально возможная.

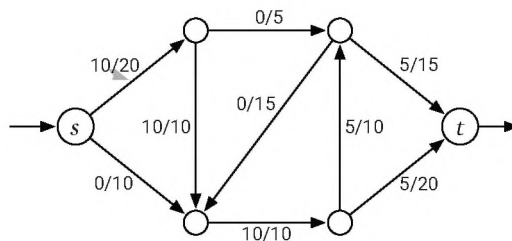


Рис. 10.2. Выполнимый (s, t) -поток с величиной 10.

Каждое ребро помечено величиной потока / пропускной способностью

10.2. Разрезы

(s,t) -разрез (или просто разрез (cut), если исходная и целевая вершина понятны из контекста) – это часть вершин в непересекающихся подмножествах S и T , означающих $S \cup T = V$ и $S \cap T = \emptyset$, – где $s \in S$ и $t \in T$.

Если у нас есть функция пропускной способности $c : E \rightarrow \mathbb{R}_{\geq 0}$, то пропускная способность разреза – это сумма пропускных способностей ребер, которые начинаются в S и заканчиваются в T :

$$\|S, T\| := \sum_{v \in S} \sum_{w \in T} c(v \rightarrow w).$$

(И в этом случае если $v \rightarrow w$ не является ребром в рассматриваемом графе, то мы предполагаем, что $c(v \rightarrow w) = 0$.) Следует отметить, что это определение асимметрично: ребра, начинающиеся в T и заканчивающиеся в S , не важны. Задача поиска наименьшего разреза (minimum cut problem) – это вычисление (s,t) -разреза, пропускная способность которого минимально возможная.

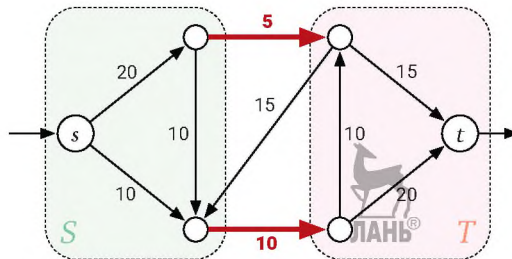


Рис. 10.3. (s,t) -разрез с пропускной способностью 15.

Каждое ребро помечено соответствующей пропускной способностью

Интуитивно наименьший разрез – это самый дешевый способ полностью прервать поток из s в t . Разумеется, нетрудно показать следующую взаимосвязь между потоками и разрезами.

Лемма 10.1. Пусть f – любой выполнимый (s,t) -поток и пусть (S, T) – любой (s,t) -разрез. Величина потока f не превышает пропускную способность (S, T) . Кроме того, $|f| = \|S, T\|$, если и только если f насыщает каждое ребро из S в T и избегает каждое ребро из T в S .

Доказательство. Выберите наиболее понравившийся вам поток f и разрез (S, T) , а затем следуйте по колеблющимся неравенствам:

$$\begin{aligned} |f| &= \partial f(s) && \text{[по определению]} \\ &= \sum_{v \in S} \partial f(v) && \text{[ограничение сохранения]} \\ &= \sum_{v \in S} \sum_w f(v \rightarrow w) - \sum_{v \in S} \sum_u f(u \rightarrow v) && \text{[математическое определение } \partial \text{]} \end{aligned}$$

$$\begin{aligned}
&= \sum_{v \in S} \sum_{w \notin S} f(v \rightarrow w) - \sum_{v \in S} \sum_{u \notin S} f(u \rightarrow v) \quad [\text{удаление ребер из } S \text{ в } S] \\
&= \sum_{v \in S} \sum_{w \in T} f(v \rightarrow w) - \sum_{v \in S} \sum_{u \in T} f(u \rightarrow v) \quad [\text{определение разреза}] \\
&\leq \sum_{v \in S} \sum_{w \in T} f(v \rightarrow w) \quad [\text{потому что } f(u \rightarrow v) \geq 0] \\
&\leq \sum_{v \in S} \sum_{w \in T} c(v \rightarrow w) \quad [\text{потому что } f(v \rightarrow w) \leq c(v \rightarrow w)] \\
&= \|S, T\| \quad [\text{по определению}].
\end{aligned}$$

На втором шаге мы просто добавляем нули, поскольку $df(v) = 0$ для каждой вершины $v \in S \setminus \{s\}$. На четвертом шаге мы удаляем величины потока $f(x \rightarrow y)$, где обе вершины x и y находятся в S , потому что они появляются в обеих суммах: с положительным знаком, когда $v = x$ и $w = y$, и с отрицательным знаком, когда $v = y$ и $u = x$.

Первое неравенство в этом выводе в действительности является равенством, если и только если f избегает каждое ребро из T в S . Аналогично второе неравенство в действительности является равенством, если и только если f насыщает каждое ребро из S в T . $\square$

Из этой леммы сразу же следует, что если $|f| = \|S, T\|$, то f непременно является максимальным потоком, а (S, T) – наименьшим разрезом.

10.3. Теорема о максимальном потоке и наименьшем разрезе (Maxflow-Mincut)

Удивительно, но в каждой транспортной сети существует выполнимый (s, t) -поток и (s, t) -разрез (S, T) такой, что $|f| = \|S, T\|$. Это знаменитая теорема о максимальном потоке и наименьшем разрезе (Maxflow-Mincut), впервые доказанная Лестером Фордом (Lester Ford) (известному по алгоритму поиска кратчайшего пути) и Делбертом Фалкерсоном (Delbert Fulkerson) в 1954 г. и независимо от них Питером Элиасом (Peter Elias), Эмиелом Файнштейном (Amiel Feinstein) и Клодом Шенноном (Claude Shannon) (известным по теории информации и роботу, решающему задачу поиска пути в лабиринте) в 1956 г.

Теорема о максимальном потоке и наименьшем разрезе (Maxflow-Mincut). В каждой транспортной сети с источником s и целью t величина максимального (s, t) -потока равна пропускной способности наименьшего (s, t) -разреза.

Форд и Фалкерсон доказали эту теорему следующим образом. Пусть определен граф G , вершины s и t и функция пропускной способности $c : E \rightarrow \mathbb{R}_{\geq 0}$. Доказательство будет проще, если мы предположим, что G является редуцированным (reduced) графом, – это означает, что существует

не более одного ребра между любыми двумя вершинами u и v . В частности, либо $c(u \rightarrow v) = 0$, либо $c(v \rightarrow u) = 0$. Это предположение легко обосновать: нужно дополнительно разделить каждое ребро $u \rightarrow v$ в G с помощью новой вершины x , заменив $u \rightarrow v$ на путь $u \rightarrow x \rightarrow v$, и определить $c(u \rightarrow v) = c(x \rightarrow v) = c(u \rightarrow v)$. Этот измененный граф имеет ту же величину максимального потока и пропускную способность наименьшего разреза, что и исходный граф.

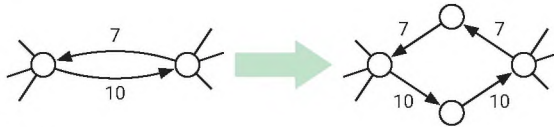


Рис. 10.4. Обоснование предположения об однонаправленности

Пусть f – произвольный выполнимый (s, t) -поток в графе G . Мы определяем новую функцию пропускной способности $c_f: V \times V \rightarrow \mathbb{R}$, называемую остаточной пропускной способностью (residual capacity), как показано ниже:

$$c_f(u \rightarrow v) = \begin{cases} c(u \rightarrow v) - f(u \rightarrow v) & , \text{ если } u \rightarrow v \in E, \\ f(v \rightarrow u) & , \text{ если } v \rightarrow u \in E, \\ 0 & \text{ иначе.} \end{cases}$$

Интуитивно остаточная пропускная способность ребра показывает, сколько еще потоков можно пропустить через это ребро. Поскольку $f \geq 0$ и $f \leq c$, эти остаточные пропускные способности всегда неотрицательны. Существует возможность $c_f(u \rightarrow v) > 0$, даже если $u \rightarrow v$ не является ребром в исходном графе G . Таким образом, мы определяем остаточный граф (residual graph) $G_f = (V, E_f)$, где E_f – множество ребер, остаточная пропускная способность которых положительна. В большинстве своем остаточные графы не являются редуцированными, в частности если $0 < f(u \rightarrow v) < c(u \rightarrow v)$, то остаточный граф G_f содержит ребро $u \rightarrow v$ и обратное ему ребро $v \rightarrow u$.

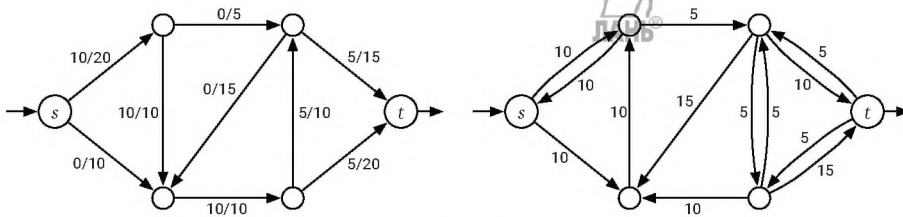


Рис. 10.5. Поток f во взвешенном графе G и соответствующий остаточный граф G_f

Теперь мы должны рассмотреть два случая: либо существует направленный путь из вершины-источника s в целевую вершину t в этом остаточном графе G_f , либо не существует.

Во-первых, предположим, что остаточный граф G_f содержит направленный путь P из s в t , и назовем P увеличивающим (дополняющим, остаточ-

ным – augmenting) путем. Пусть $F = \min_{u \rightarrow v \in P} c_f(u \rightarrow v)$ обозначает максимальную величину потока, которую мы можем провести через P . Мы определяем новый поток $f' : E \rightarrow \mathbb{R}$ (в исходном графе) следующим образом:

$$f'(u \rightarrow v) = \begin{cases} f(u \rightarrow v) + F & , \text{ если } u \rightarrow v \in P, \\ f(u \rightarrow v) - F & , \text{ если } v \rightarrow u \in P, \\ f(u \rightarrow v) & \text{ иначе.} \end{cases}$$

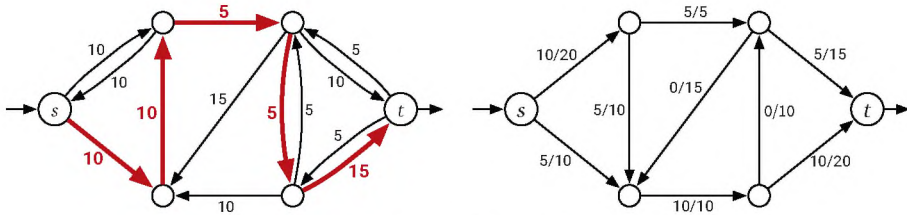


Рис. 10.6. Увеличивающий путь с величиной $F = 5$ и полученный в итоге увеличивающий поток f'

Я утверждаю, что этот новый поток f' является достижимым с учетом исходных пропускных способностей c , означающих, что $f' \geq 0$ и $f' \leq c$ везде. Рассмотрим ребро $u \rightarrow v$ в исходном графе G . Существуют следующие три случая:

- если увеличивающий путь P содержит ребро $u \rightarrow v$, то

$$f'(u \rightarrow v) = f(u \rightarrow v) + F > f(u \rightarrow v) \geq 0,$$

потому что f – достижимый, и

$$\begin{aligned} f'(u \rightarrow v) &= f(u \rightarrow v) + 1 && \text{по определению } f' \\ &\leq f(u \rightarrow v) + c_f(u \rightarrow v) && \text{по определению } F \\ &= f(u \rightarrow v) + c(u \rightarrow v) - f(u \rightarrow v) && \text{по определению } c_f \\ &= c(u \rightarrow v) && \text{Уф;} \end{aligned}$$

- если увеличивающий путь P содержит обратное ребро $v \rightarrow u$, то

$$f'(u \rightarrow v) = f(u \rightarrow v) - F < f(u \rightarrow v) \leq c(u \rightarrow v),$$

и снова потому, что f – достижимый, и

$$\begin{aligned} f'(u \rightarrow v) &= f(u \rightarrow v) - F && \text{по определению } f' \\ &\geq f(u \rightarrow v) - c_f(u \rightarrow v) && \text{по определению } F \\ &= f(u \rightarrow v) - f(u \rightarrow v) && \text{по определению } c_f \\ &= 0 && \text{Уф;} \end{aligned}$$

- наконец, если ни $u \rightarrow v$, ни $v \rightarrow u$ не находятся в увеличивающем пути, то $f'(u \rightarrow v) = f(u \rightarrow v)$, следовательно, $0 \leq f'(u \rightarrow v) \leq c(u \rightarrow v)$, потому что f – достижимый.

Итак, f , несомненно, является достижимым.

Наконец, только первое ребро в увеличивающем пути выходит из s , из чего следует $|f'| = |f| + F > |f|$. Следовательно, f' является достижимым потоком, величина которого больше f . Мы приходим к выводу, что если существует путь из s в t в остаточном графе G_f , то f не является максимальным потоком.

С другой стороны, предположим, что остаточный граф G_f не содержит направленного пути из s в t . Пусть S – множество вершин, достижимых из s в G_f , и пусть $T = V \setminus S$. Разбиение (S, T) , очевидно, является (s, t) -разрезом. Для каждой вершины $u \in S$ и $v \in T$ имеем:

$$c_f(u \rightarrow v) = (c(u \rightarrow v) - f(u \rightarrow v)) + f(v \rightarrow u) = 0.$$

Достижимость f предполагает, что $c(u \rightarrow v) - f(u \rightarrow v) \geq 0$ и $f(v \rightarrow u) \geq 0$, поэтому в действительности мы непременно получаем $c(u \rightarrow v) - f(u \rightarrow v) = 0$ и $f(v \rightarrow u) = 0$. Другими словами, наш поток f насыщает каждое ребро из S в T и избегает каждого ребра из T в S . Теперь лемма 10.1 предполагает, что $|f| = ||S, T||$, а это означает, что f – максимальный поток и (S, T) – наименьший разрез.

Доказательство завершено. □

10.4. Алгоритм увеличивающего пути Форда и Фалкерсона

Доказательство Форда и Фалкерсона теоремы о максимальном потоке и наименьшем разрезе (Maxflow-Mincut) сразу же предоставляет алгоритм вычисления максимальных потоков: начиная с нулевого потока, многократно увеличиваем этот поток вдоль любого пути из s в t в остаточном графе до тех пор, пока такого пути не останется.

Этот алгоритм имеет важное, но простое следствие.

Теорема целочисленности. Если все пропускные способности в транспортной сети являются целыми числами, то существует максимальный поток такой, что поток через каждое ребро является целым числом.

Доказательство. Докажем методом индукции, что после каждой итерации алгоритма увеличивающего пути величины всех потоков и все остаточные пропускные способности являются целыми числами.

- Перед первой итерацией величины всех потоков равны нулю (который является целым числом), а все остаточные пропускные способности равны исходным, которые являются целыми числами по определению.
- На каждой последующей итерации по индуктивному предположению пропускная способность F увеличивающего пути является целым числом, поэтому увеличение пути изменяет поток на каждом

ребре, следовательно, и остаточную пропускную способность каждого ребра на целое число.

В частности, на каждой итерации алгоритма увеличивающего пути величина потока возрастает на положительное число. Из этого следует, что алгоритм в конце концов остановится и вернет максимальный поток. $\square$

Если пропускная способность каждого ребра является целым числом, то по самым осторожным оценкам алгоритм Форда–Фалкерсона останавливается после не более чем $|f^*|$ итераций, где f^* – действительный максимальный поток. На каждой итерации мы можем сформировать остаточный граф G_f и выполнить поиск в любом направлении, чтобы найти увеличивающий путь за время $O(E)$. Таким образом, при этом условии алгоритм выполняется за время $O(E|f^*|)$ в наихудшем случае.

Джек Эдмондс (Jack Edmonds) и Ричард Карп (Richard Karp) заметили, что этот анализ времени выполнения является чрезвычайно строгим. Рассмотрим сеть с четырьмя узлами, показанную на рис. 10.7, где X – некоторое большое целое число. Очевидно, что максимальный поток в этой сети $2X$. Но алгоритм Форда–Фалкерсона может чередовать перемещение одного элемента потока по увеличивающему пути $s \rightarrow u \rightarrow v \rightarrow t$ и по увеличивающему пути $s \rightarrow v \rightarrow u \rightarrow t$, в результате время выполнения становится равным $\Theta(X) = \Omega(|f^*|)$.

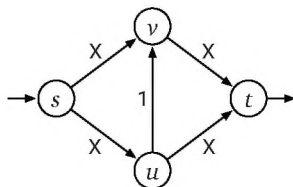


Рис. 10.7. Приведенный Эдмондсом и Карпом пример неэффективного выполнения алгоритма Форда–Фалкерсона

На практике алгоритм Форда–Фалкерсона обычно выполняется быстро, и всегда быстро, если величина максимального потока $|f^*|$ мала, но без дополнительных ограничений для увеличивающих путей это не самый эффективный алгоритм в наихудшем случае. Пример Эдмондса и Карпа неудобной сети можно описать с использованием всего лишь $O(\log X)$ бит, следовательно, время выполнения алгоритма Форда–Фалкерсона в действительности растет экспоненциально в зависимости от размера входных данных.

♥ Иррациональные пропускные способности

Но что, если пропускные способности не являются целыми числами? Если мы умножим все пропускные способности на одну и ту же (положительную) константу, то максимальный поток увеличится везде на тот же

постоянный множитель. Из этого следует, что если все пропускные способности ребер являются рациональными числами, то алгоритм Форда–Фалкерсона в итоге останавливается, но время его выполнения остается экспоненциальным (по числу битов, используемых для описания входных данных).

Но если мы разрешим иррациональные пропускные способности, то алгоритм может реально заикнуться бесконечно, постоянно находя все меньшие и меньшие увеличивающие пути. Что еще хуже, эта бесконечная последовательность увеличивающих путей может даже не сходить к максимальному потоку или хотя бы к какой-либо значимой части максимального потока. Самая маленькая сеть, демонстрирующая такое плохое поведение, была открыта Ури Цвиком (Uri Zwick) в 1993 г.²

Рассмотрим сеть с шестью узлами, показанную на рис. 10.8. Шесть из девяти ребер имеют одинаковую большую целочисленную пропускную способность X , для двух ребер пропускная способность равна 1, пропускная способность одного ребра равна числу $\phi = (\sqrt{5} - 1)/2 \approx 0.618034$, выбранному так, чтобы $1 - \phi = \phi^2$. Для доказательства того, что алгоритм Форда–Фалкерсона может застрять, мы можем отслеживать остаточные пропускные способности трех горизонтальных ребер по мере выполнения алгоритма. (Остаточные пропускные способности остальных шести ребер всегда будут не меньше $X - 3$.)

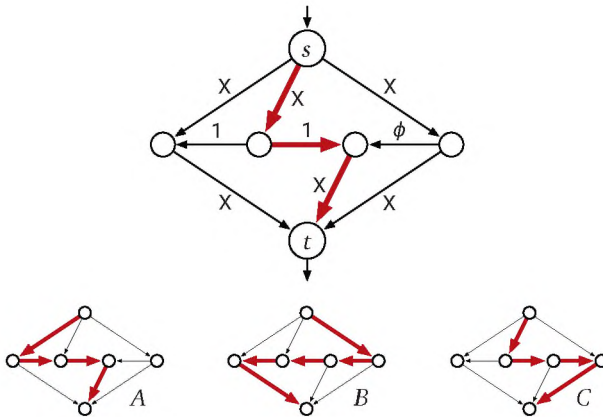


Рис. 10.8. Приведенный Ури Цвиком пример никогда не завершающегося потока и три увеличивающих пути

Предположим, что алгоритм Форда–Фалкерсона начинает с выбора центрального увеличивающего пути, показанного сверху на рис. 10.8. Три горизонтальных ребра в порядке слева направо теперь имеют остаточные пропускные способности 1, 0 и ϕ . Предположим индуктивно, что горизонтальные пропускные способности равны ϕ^{k-1} , 0 и ϕ^k при некотором неотрицательном целом числе k .

² В 1962 г. Форд и Фалкерсон описали более сложную сеть с 10 вершинами и 48 ребрами с таким же плохим поведением.

1. Увеличение вдоль пути B , прибавляющее ϕ^k к потоку. Теперь остаточные пропускные способности равны ϕ^{k+1} , ϕ^k и 0.
2. Увеличение вдоль пути C , прибавляющее ϕ^k к потоку. Теперь остаточные пропускные способности равны ϕ^{k+1} , 0 и ϕ^k .
3. Увеличение вдоль пути B , прибавляющее ϕ^{k+1} к потоку. Теперь остаточные пропускные способности равны 0, ϕ^{k+1} и ϕ^{1+2} .
4. Увеличение вдоль пути A , прибавляющее ϕ^{k+1} к потоку. Теперь остаточные пропускные способности равны ϕ^{k+1} , 0 и ϕ^{1+2} .

По индукции из этого следует, что после 4^{n+1} шагов увеличения горизонтальные ребра имеют остаточные пропускные способности $\phi^{2^{n-2}}$, 0 и $\phi^{2^{n-1}}$. Поскольку число увеличений возрастает до бесконечности, величина потока сходится к

$$1 + 2 \sum_{i=1}^{\infty} \phi^i = 1 + \frac{2}{1 - \phi} = 4 + \sqrt{5} < 7,$$

даже несмотря на то, что величина максимального потока, очевидно, $2X + 1 \gg 7$.

Практически мыслящие читатели, возможно, удивляются, почему никто не уделил внимание иррациональным пропускным способностям, ведь компьютеры не могут точно представить ничего, кроме (небольших) целых или рациональных чисел (как пары небольших чисел). Это хороший вопрос. Математики отвечают, что ограничение пропускных способностей целыми числами является в буквальном смысле искусственным, это артефакт (искусственный объект) цифровой вычислительной аппаратуры (или, возможно, других не относящихся к этому вопросу законов физики), а не внутренне присущее свойство абстрактной вычислительной задачи. Но еще более практическая причина состоит в том, что поведение алгоритма с иррациональными входными данными сообщает нам что-то о его поведении в наихудшем случае на практике с помощью пропускных способностей в виде чисел с плавающей точкой – это ужасно. Даже с чрезвычайно разумными пропускными способностями небрежная реализация алгоритма Форда–Фалкерсона может привести к бесконечному циклу (просто из-за ошибки округления) и никогда не позволит сколько-нибудь приблизиться к правильному ответу.

10.5. Объединения и разбиения потоков

Потоки обычно определяются как функции от ребер графа, удовлетворяющие определенным ограничениям в вершинах. Но потоки имеют вторую характеристику, более естественную и полезную в определенных контекстах.

Рассмотрим произвольный граф G с исходной вершиной s и целевой вершиной t . Определим любые два (s, t) -потока f и g и любые два действи-

тельных числа α и β и рассмотрим функцию $h : E \rightarrow \mathbb{R}$, определяемую условием

$$h(u \rightarrow v) := \alpha \cdot f(u \rightarrow v) + \beta \cdot g(u \rightarrow v)$$

для каждого ребра $u \rightarrow v$. Мы можем записать это определение проще: $h = \alpha f + \beta g$. По определению очевидно предполагается, что h также является (s, t) -поток с величиной $|h| = \alpha|f| + \beta|g|$. В более общем смысле любая линейная комбинация (s, t) -поток с величиной $|h|$ также является (s, t) -поток с величиной $|h|$.

Оказывается, что любой (s, t) -поток можно записать как взвешенную сумму потоков с помощью специальной структуры. Для любого направленного пути P из s в t мы определяем соответствующий путь потока (path flow) следующим образом:

$$P(u \rightarrow v) = \begin{cases} 1 & , \text{ если } u \rightarrow v \in P, \\ -1 & , \text{ если } v \rightarrow u \in P, \\ 0 & \text{ иначе.} \end{cases}$$

По определению очевидно предполагается, что функция $P : E \rightarrow \mathbb{R}$ действительно является (s, t) -поток с величиной 1. Я умышленно перегружаю смысл переменной P , чтобы она обозначала и путь (последовательность вершин и направленных ребер), и элемент потока вдоль этого пути.

Аналогично для любого направленного цикла C мы определяем соответствующий цикл потока (cycle flow) следующим условием:

$$C(u \rightarrow v) = \begin{cases} 1 & , \text{ если } u \rightarrow v \in C, \\ -1 & , \text{ если } v \rightarrow u \in C, \\ 0 & \text{ в иных случаях.} \end{cases}$$

И в этом случае легко проверить, что $C : E \rightarrow \mathbb{R}$ является (s, t) -поток с нулевой величиной.

Наше предыдущее обоснование предполагало, что любая линейная комбинация путей и циклов потоков является другим потоком; эта взвешенная сумма называется декомпозицией потока (flow decomposition). Кроме того, каждый неотрицательный поток имеет декомпозицию потока со следующей специальной структурой.

Теорема о декомпозиции потока. Каждый неотрицательный (s, t) -поток f можно записать как положительную линейную комбинацию направленных (s, t) -путей и направленных циклов. Кроме того, направленное ребро $u \rightarrow v$ появляется как минимум в одном из этих путей или циклов, если и только если $f(u \rightarrow v) > 0$ и общее число путей и циклов не превышает число ребер в рассматриваемой сети.

Доказательство. Докажем эту теорему методом индукции по числу ребер, несущих ненулевой поток, интуитивно выполняя алгоритм Форда–Фалкерсона в обратном направлении. Пока хотя бы одно ребро в графе

несет положительный поток, мы можем найти либо (s, t) -путь, либо направленный цикл, содержащий поток. Вычитание стольких потоков, сколько возможно, из этого пути или цикла опустошает как минимум одно ребро, поэтому Фей Рекурсия может дать нам остаток этого разбиения.

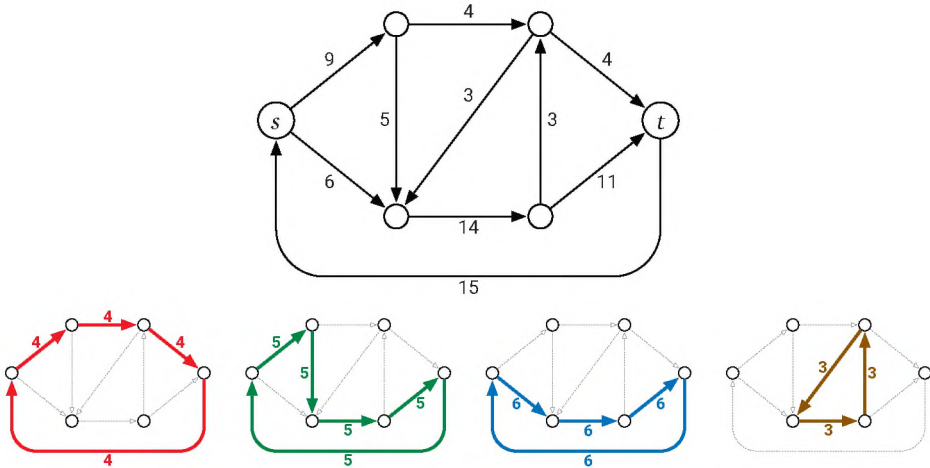


Рис. 10.9. Разбиение циркуляции на взвешенные направленные циклы

Для формализации этого обоснования сначала рассмотрим особый случай циркуляций (circulations) – это потоки с величиной 0, когда поток остается в одной вершине. Определим произвольную циркуляцию f в произвольной транспортной сети, и пусть $\#f$ обозначает число ребер $u \rightarrow v$ такое, что $f(u \rightarrow v) > 0$. Докажем, что f можно разбить на положительную линейную комбинацию из не более чем $\max\{0, \#f - 1\}$ циклов методом индукции по $\#f$. Необходимо рассмотреть три случая:

- если $\#f = 0$, то f является вакуантной линейной комбинацией из нулевых циклов;
- предположим, что $f(u \rightarrow v) > 0$ для одного направленного цикла из ребер $u \rightarrow v$. Тогда $\#f \geq 2$ и f является тривиальной линейной комбинацией из одного цикла;
- иначе выберем произвольное ребро $u \rightarrow v$ при $f(u \rightarrow v) > 0$. Рассмотрим произвольный маршрут $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots$ при $v_0 = u$ и $v_1 = v$ такой, что $f(v_{i-1} \rightarrow v_i) > 0$ при любом индексе i . Ограничение сохранения предполагает, что каждая вершина с входящим потоком также имеет и исходящий поток, поэтому мы можем сделать этот маршрут произвольно длинным. В частности, маршрут обязательно должен в конце концов посетить некоторую вершину более одного раза. Пусть k – наименьший индекс такой, что $v_j = v_k$ для некоторого индекса $j < k$. Отрезок маршрута $v_j \rightarrow v_{j+1} \rightarrow \dots \rightarrow v_k$ – это простой направленный цикл C .

Определим $F := \min_e f(e)$ и рассмотрим функцию $f' := f - F \cdot C$, или, более подробно:

$$f'(u \rightarrow v) = \begin{cases} f(u \rightarrow v) - F & , \text{ если } u \rightarrow v \in C, \\ f(u \rightarrow v) & \text{ в иных случаях.} \end{cases}$$

По определению очевидно предполагается, что f' – еще одна достижимая циркуляция в G . Существует как минимум одно ребро $e \in C$ такое, что $f(e) = F$, следовательно, $f'(e) = 0$, что предполагает $\#f' \leq \#f - 1$. Поскольку поток несет меньшее число ребер в f' , чем в f , Фейя Рекурсия может разбить f' на не более чем $\#f' - 1 \leq \#f - 2$ циклов. Добавление F элементов потока вокруг цикла C дает нам разбиение потока для f ; более лаконично: $f = f' + F \cdot C$.

Теперь пусть f – произвольный (s, t) -поток в произвольной транспортной сети, такой что $|f| > 0$. Добавим ребро $t \rightarrow s$ в эту сеть и определим циркуляцию f' по условию $f'(t \rightarrow s) = |f|$ и $f'(u \rightarrow v) = f(u \rightarrow v)$ для каждого исходного ребра $u \rightarrow v$, также заметим, что $\#f' = \#f + 1 \geq 2$. Предыдущее обоснование предполагало, что циркуляция f' является положительной линейной комбинацией не более чем $\#f' - 1$ направленных циклов. Удаление ребра $t \rightarrow s$ дает нам разбиение исходного потока f на не более чем $\#f' - 1 = \#f$ путей и циклов. В частности, циклы в f' , которые включают $t \rightarrow s$, становятся (s, t) -путями в f , а циклы в f' , не включающие $t \rightarrow s$, остаются циклами в f . $\square$

Доказательство теоремы о разбиении потока предполагает более строгие результаты в двух интересных частных случаях.

- Любую циркуляцию можно разбить на взвешенную сумму циклов; никакие пути не являются необходимыми.
- Любой ациклический (s, t) -поток можно разбить на взвешенную сумму (s, t) -потоков; никакие циклы не являются необходимыми.

Кроме того, удаляя циклы потока до тех пор, пока их не останется совсем, мы можем преобразовать любой поток в ациклический с той же величиной. В частности, каждая транспортная сеть содержит максимальный (s, t) -поток, являющийся ациклическим.

Это доказательство также сразу же переводится непосредственно в алгоритм, похожий на алгоритм Форда–Фалкерсона, для разбиения любого (s, t) -потока на пути и циклы. Алгоритм многократно ищет либо направленный (s, t) -путь, либо направленный цикл в оставшемся потоке, затем вычитает поток по возможному максимуму вдоль этого пути или цикла до тех пор, пока поток не станет пустым. Мы можем найти путь или цикл потока за время $O(V)$, как показано ниже:

- если любое ребро, исходящее из s , имеет положительный поток, то следовать по произвольному блужданию из s в графе потока до тех пор, пока не будет достигнута либо вершина t (что дает нам путь потока), либо та же вершина во второй раз (что дает нам цикл потока);

- если никакое ребро, исходящее из s , не имеет положительного потока, то найти другую вершину v с положительным исходящим потоком и следовать по произвольному маршруту из v в графе потока до тех пор, пока не будет достигнута та же вершина во второй раз (что дает нам цикл потока).

В обоих случаях ограничение сохранения предполагает, что этот алгоритм никогда не зайдет в тупик. Каждая итерация требует времени $O(V)$ и удаляет как минимум одно ребро из графа потока, следовательно, весь алгоритм декомпозиции выполняется за время $O(VE)$.

Декомпозиция потоков обеспечивают естественную более низкую границу времени выполнения любого алгоритма поиска максимального потока, который строит один путь или цикл потока за один шаг. Каждый поток можно разбить на не более чем E путей и циклов, каждый из которых использует не более чем V ребер, поэтому общая сложность алгоритма декомпозиции потока равна $O(VE)$. Кроме того, легко создать потоки, для которых каждая декомпозиция потока имеет сложность $\Omega(VE)$. Таким образом, любой алгоритм поиска максимального потока, который явно создает один путь или цикл потока за один шаг, – в частности, любая реализация алгоритма Форда–Фалкерсона поиска увеличивающего пути, – в наихудшем случае непременно должен выполняться за время $\Omega(VE)$.

10.6. Алгоритмы Эдмондса и Карпа

Алгоритм Форда–Фалкерсона не определяет, какой путь в остаточном графе выбирается для увеличения. Такое плохое поведение этого алгоритма в наихудшем случае можно списать на неудачные варианты выбора увеличивающего пути. В начале 1970-х гг. Джек Эдмондс (Jack Edmonds) и Ричард Карп (Richard Karp) опубликовали два естественных правила для выбора увеличивающих путей, которые привели к созданию более эффективных алгоритмов.

Самые насыщенные увеличивающие пути

Первое правило Эдмондса и Карпа, по существу, является жадным алгоритмом:

Выбрать увеличивающий путь с наибольшей величиной узкого места.

Нетрудно показать, что (s,t) -путь с максимальной величиной узкого места (maximum-bottleneck) в направленном графе можно вычислить за время $O(E \log V)$, используя метод обхода с поиском по первому наилучшему совпадению по аналогии с алгоритмом Ярника поиска минимального остовного дерева или с алгоритмом Дейкстры поиска кратчайшего пути. Алгоритм строит направленное дерево T с корнем в s по одной вершине за

шаг, многократно добавляя ребро с максимальной пропускной способностью, исходящее из T в T , до тех пор пока T не будет содержать путь из s в t . В другом варианте можно было бы эмулировать алгоритм Краскала – вставлять по одному ребру в порядке возрастания пропускной способности до тех пор, пока не появится путь из s в t , – хотя этот подход менее эффективен, по крайней мере, когда граф направленный.

Для завершения анализа времени выполнения алгоритма поиска потока нам необходима верхняя граница числа итераций, выполненных до остановки алгоритма. Фактически для произвольных действительных значений пропускной способности алгоритм может никогда не остановиться – см. упражнение 18. Но для целочисленных значений пропускной способности мы можем ограничить число итераций как функцию величины максимального потока $|f^*|$, как показано ниже.

Пусть f – любой поток в графе G , и пусть f' – максимальный поток в текущем остаточном графе G_f . (В начале алгоритма $G_f = G$ и $f' = f^*$.) Мы уже доказали, что f' можно разбить не более чем на E путей и циклов. Обоснование простого усреднения предполагает, что как минимум один из путей в этом разбиении обязательно должен содержать не менее $|f'|/E$ элементов потока a . Из этого сразу же следует, что самый насыщенный (s, t) -путь в G_f содержит не менее $|f'|/E$ элементов потока.

Таким образом, увеличение f вдоль пути с максимальной величиной узкого места в G_f умножает величину оставшегося максимального потока в G_f на коэффициент, не превышающий $1 - 1/E$. Другими словами, величина остаточного максимального потока убывает экспоненциально при увеличении числа итераций. После $E \cdot \ln |f^*|$ итераций величина максимального потока в G_f не превышает

$$|f^*| \cdot (1 - 1/E)^{E \cdot \ln |f^*|} < |f^*| e^{-\ln |f^*|} = 1.$$

(Здесь e – постоянная Эйлера, а не ребро e . Извините.) В частности, после $E \cdot \ln |f^*|$ итераций величина остаточного максимального потока меньше 1. Если все пропускные способности являются целыми числами, то величина остаточного максимального потока также является целым числом, поэтому обязательно должна быть равна 0. Другими словами, f – это максимальный поток!

Мы приходим к выводу, что для графов с целочисленными пропускными способностями алгоритм Эдмондса–Карпа «самых насыщенных путей» выполняется за время $O(E^2 \log E \log |f^*|)$. В отличие от времени выполнения исходного алгоритма Форда–Фалкерсона эта граница времени действительно является полиномиальной функцией от размера входных данных.

Как и исходный алгоритм Форда–Фалкерсона, алгоритм «самого насыщенного пути» может застрять в бесконечном цикле в сетях с произвольными действительными значениями пропускных способностей. Но наш анализ предполагает, что, даже если алгоритм никогда не остановится, он поддерживает поток f , который в пределе достигает максимального потока.

Кратчайшие увеличивающие пути

Второе правило Эдмондса–Карпа в действительности было предложено в качестве практической эвристики Фордом и Фалкерсоном в их первоначальной работе о поиске максимального потока. Вариант этого правила был независимо предложен в 1970 г. русским математиком Ефимом Диницем³.

Выбрать увеличивающий путь с наименьшим числом ребер.

Кратчайший увеличивающий путь можно найти за время $O(E)$, выполняя поиск в ширину в остаточном графе. Удивительно, что полученный в итоге алгоритм останавливается после полиномиального числа итераций независимо от реальных пропускных способностей ребер.

Доказательство этой полиномиальной верхней границы основано на двух замечаниях, касающихся развития остаточного графа. Пусть f_i – текущий поток после i увеличивающих шагов и G_i – соответствующий остаточный граф. В частности, f_0 везде равен нулю и $G_0 = G$. Для каждой вершины v пусть $level_i(v)$ обозначает расстояние невзвешенного кратчайшего пути из s в v в G_i или – равнозначно – уровень вершины v в дереве поиска в ширину графа G_i с корнем в s . В частности, если не существует пути из s в v в G_i , то $level_i(v) = \infty$ (потому что $\min \emptyset = \infty$).

Наше первое утверждение состоит в том, что уровень вершины может только увеличиваться с течением времени.

Лемма 10.2. $level_i(v) \geq level_{i-1}(v)$ для всех вершин v и для всех целых чисел $i > 0$.

Доказательство. Определим произвольное положительное целое число $i > 0$ и произвольную вершину v . Докажем это утверждение методом индукции по $level_i(v)$ (но не по целому числу i). По методу индукции предположим для каждой вершины u такой, что $level_i(u) < level_i(v)$, что $level_i(u) \geq level_{i-1}(u)$. Здесь необходимо рассмотреть три случая:

- если $v = s$, то сразу же имеем $level_i(s) = level_{i-1}(s) = 0$;
- если не существует пути из s в v в G_i , то $level_i(v) = \infty \geq level_{i-1}(v)$;
- иначе пусть $s \rightarrow \dots \rightarrow u \rightarrow v$ – любой невзвешенный кратчайший путь из s в v в графе G_i . Поскольку это кратчайший путь, мы имеем $level_i(v) = level_i(u) + 1$, поэтому по индуктивному предположению $level_i(u) \geq level_{i-1}(u)$. Для завершения доказательства мы должны пока-

³ В частности, Диниц открыл более сложный алгоритм поиска максимального потока, когда студентом изучал курс алгоритмов, который читал Георгий Адельсон-Вельский (один из авторов AVL-деревьев; «AB» в аббревиатуре AVL). Диниц разработал этот алгоритм, выполняя аудиторное задание. Алгоритм Диница также направляет потоки вдоль кратчайших путей, но с дополнительным хранением промежуточных результатов для сокращения времени выполнения с $O(VE^2)$ до $O(V^2E)$.

зять, что $level_{i-1}(u) \geq level_{i-1}(v) - 1$. Для этого необходимо рассмотреть два дополнительных случая:

- ♦ если $u \rightarrow v$ – ребро в G_{i-1} , то $level_{i-1}(v) \leq level_{i-1}(u) + 1$, потому что уровни определяются обходом поиска в ширину;
- ♦ с другой стороны, если $u \rightarrow v$ не является ребром в G_{i-1} , то обратное ему ребро $v \rightarrow u$ обязательно должно быть ребром в i -м увеличивающем пути, который по определению является кратчайшим путем из s в t в графе G_{i-1} . Из этого следует, что $level_{i-1}(v) = level_{i-1}(u) - 1 \leq level_{i-1}(u) + 1$.

В обоих дополнительных случаях мы заключаем, что $level_i(v) = level_i(u) + 1 \geq level_{i-1}(u) + 1 \geq level_{i-1}(v)$. $\square$

Когда мы увеличиваем поток, ребро, являющееся узким местом в увеличивающем пути, исчезает из остаточного графа, а некоторые ребра с направлением, противоположным увеличивающему пути, могут (повторно) появиться. Наше второе утверждение состоит в том, что ребро не может появляться или исчезать слишком часто.

Лемма 10.3. Во время выполнения алгоритма Эдмондса–Карпа поиска кратчайшего увеличивающего пути каждое ребро $u \rightarrow v$ исчезает из остаточного графа G_f не более $V/2$ раз.

Доказательство. Предположим, что ребро $u \rightarrow v$ находится в двух остаточных графах G_i и G_{j+1} , но не в каком-либо из промежуточных остаточных графов $G_{i+1}, \dots, G_j$ при некоторых значениях $i < j$. Тогда ребро $u \rightarrow v$ обязательно должно содержаться в i -м увеличивающем пути, поэтому $level_i(v) = level_i(u) + 1$, а $v \rightarrow u$ обязательно должно содержаться в j -м увеличивающем пути, поэтому $level_j(v) = level_j(u) - 1$. Предыдущая лемма предполагает, что

$$level_j(u) = level_j(v) + 1 \geq level_i(v) + 1 = level_i(u) + 2.$$

Другими словами, между исчезновением и повторным появлением ребра $u \rightarrow v$ расстояние из s в u увеличилось не более чем на 2. Поскольку каждый уровень либо меньше V , либо равен бесконечности, число исчезновений не превышает $V/2$. $\square$

Теперь мы можем вывести верхнюю границу по числу итераций. Поскольку каждое ребро исчезает не более $V/2$ раз, в общей сложности происходит не более $EV/2$ исчезновений ребер. Но как минимум одно ребро исчезает на каждой итерации, поэтому алгоритм обязательно должен остановиться после не более чем $EV/2$ итераций. Наконец, каждая итерация требует времени $O(E)$, поэтому общее время выполнения алгоритма равно $O(VE^2)$.

10.7. Дальнейшее развитие

История алгоритмов поиска максимального потока весьма далека от своего завершения. Десятилетия дальнейших исследований привели к появлению нескольких более быстрых алгоритмов, некоторые из них кратко описаны в табл. 10.1.⁴ Все перечисленные алгоритмы вычисляют максимальный поток за несколько итераций. У большинства алгоритмов имеется два варианта: простая версия, выполняющая каждую итерацию методом «грубой силы» (прямым перебором), и более быстрый вариант, использующий хитроумные структуры данных для сопровождения остоного дерева транспортной сети, так что каждая итерация может быть выполнена (с обновлением остоного дерева) за логарифмическое время. Нет никаких оснований верить, что самые лучшие алгоритмы, известные к настоящему моменту, являются оптимальными. Разумеется, задача поиска максимальных потоков остается весьма активной областью исследований.

Таблица 10.1. Несколько чисто комбинаторных алгоритмов поиска максимального потока с указанием их времени выполнения

Методика	Выполнение напрямую	Выполнение с использованием динамических деревьев	Источник(и)
Блокирующий поток	$O(V^2E)$	$O(VE \log V)$	[Диниц; Карзанов; Ивен (Even) и Итаи (Itai); Слейтор (Sleator) и Тарьян (Tarjan)]
Симплекс-метод для сети	$O(V^2E)$	$O(VE \log V)$	[Данциг (Dantzig); Гольдфарб (Goldfarb) и Хао (Hao); Гольдберг (Goldberg), Григориадис (Grigoriadis) и Тарьян (Tarjan)]
Алгоритм проталкивания предпотока (обобщенный)	$O(V^2E)$	—	[Гольдберг (Goldberg) и Тарьян (Tarjan)]
Алгоритм проталкивания предпотока (FIFO)	$O(V^3)$	$O(VE \log(V^2/E))$	[Гольдберг (Goldberg) и Тарьян (Tarjan)]
Алгоритм проталкивания предпотока (наивысшая метка)	$O(V^2\sqrt{E})$	—	[Чериян (Cheriyani) и Махешвари (Maheshwari); Тунсель (Tunçel)]
Игры с использованием алгоритма проталкивания предпотока с добавлением	—	$O(VE \log_{E/(V \log V)} V)$	[Чериян (Cheriyani) и Хегеруп (Hagerup); Кинг (King), Пао (Rao) и Тарьян (Tarjan)]

⁴ Чтобы табл. 10.1 не стала слишком длинной, я умышленно не включил в нее алгоритмы, время выполнения которых зависит от пропускных способностей ребер или от величины максимального потока. Даже с учетом этого ограничения список весьма неполный.

Методика	Выполнение напрямую	Выполнение с использованием динамических деревьев	Источник(и)
Псевдопоток	$O(V^2E)$	$O(VE \log V)$	[Хохбаум (Hochbaum)]
Псевдопоток (наивысшая метка)	$O(V^3)$	$O(VE \log(V^2/E))$	[Хохбаум (Hochbaum) и Орлин (Orlin)]
Инкрементальный поиск в ширину	$O(V^2E)$	$O(VE \log(V^2/E))$	[Голдберг (Goldberg), Хелд (Held), Каплан (Kaplan), Тарьян (Tarjan) и Вернек (Werneck)]
Компактные сети	—	$O(VE)$	[Орлин (Orlin)]

Самый быстрый известный (чисто комбинаторный) алгоритм поиска максимального потока, анонсированный Джеймсом Орлином (James Orlin) в 2012 г., выполняется за время $O(VE)$, что в точности соответствует сложности в наихудшем случае для алгоритма разбиения потока. Подробности алгоритма Орлина выходят за рамки тематики этой книги. В дополнение к собственным новым методикам Орлин использует несколько старых алгоритмов и структур данных как черные ящики, которые по большей части сами по себе достаточно сложны. В частности, алгоритм Орлина не формирует явное разбиение потока, фактически для графов, содержащих только $O(V)$ ребер, расширение алгоритма Орлина действительно выполняется всего лишь за время $O(V^2 / \log V)$. Тем не менее при анализе алгоритмов, использующих максимальные потоки, это временная граница, на которую вы должны ориентироваться. Поэтому запишите следующее утверждение в свои памятки-шпаргалки и цитируйте в своих домашних работах:

Максимальные потоки можно вычислить за время $O(VE)$.

Наконец, более быстрые алгоритмы поиска максимального потока известны для сетей единичной пропускной способности, где каждое ребро имеет пропускную способность 1. В 1973 г. Александр Карзанов доказал, что алгоритм Диница блокирующего потока – первый алгоритм в табл. 10.1 – при этом условии выполняется за время $O(\min\{V^{2/3}, E^{1/2}\}E)$. (Эта временная граница выглядит как ломающая барьер $\Omega(VE)$ декомпозиции потока, но в действительности анализ Карзанова предполагает, что любой поток в сети с единичной пропускной способностью можно разбить на пути с суммарной сложностью $O(\min\{V^{2/3}, E^{1/2}\}E)$.) Это был самый быстрый известный алгоритм при таком условии в течение четырех десятилетий. Достижение Карзанова наконец удалось превзойти в 2013 г., когда Александр Мондри (Aleksander Mądry) предъявил действительно замечательный алгоритм, который вычисляет максимальные потоки в сетях с единичной пропускной

способностью за время $O(E^{10/7} \text{polylog } E)$. Но подробности алгоритма Мондри выходят за рамки тематики этой книги, да и в целом за рамки компетенции ее автора.

Упражнения

0. Предположим, что задан направленный граф $G = (V, E)$, две вершины s и t , функция пропускной способности $c : E \rightarrow \mathbb{R}^+$ и вторая функция $f : E \rightarrow \mathbb{R}$. Описать алгоритм, определяющий, является ли f максимальным (s, t) -поток в G .
1. Пусть f и f' – два выполнимых (s, t) -потока в транспортной сети G , такой, что $|f'| > |f|$. Доказать, что существует выполнимый (s, t) -поток с величиной $|f'| - |f|$ в остаточной сети G_f .
2. Пусть $u \rightarrow v$ – произвольное ребро в произвольной транспортной сети G . Доказать, что если существует минимальный (s, t) -разрез (S, T) такой, что $u \in S$ и $v \in T$, то не существует минимального разреза (S', T') такого, что $u \in T'$ и $v \in S'$.
3. Пусть (S, T) и (S', T') – наименьшие (s, t) -разрезы в некоторой транспортной сети G . Доказать, что $(S \cap S', T \cup T')$ и $(S \cup S', T \cap T')$ также являются наименьшими (s, t) -разрезами в G .
4. Пусть G – транспортная сеть, которая содержит пару противоположно направленных ребер $u \rightarrow v$ и $v \rightarrow u$ с положительными пропускными способностями. Пусть G' – транспортная сеть, полученная из G снижением пропускных способностей обоих этих ребер на величину $\min\{c(u \rightarrow v), c(v \rightarrow u)\}$. Другими словами:
 - если $c(u \rightarrow v) > c(v \rightarrow u)$, то изменить пропускную способность ребра $u \rightarrow v$ на $c(u \rightarrow v) - c(v \rightarrow u)$ и удалить ребро $v \rightarrow u$;
 - если $c(u \rightarrow v) < c(v \rightarrow u)$, то изменить пропускную способность ребра $v \rightarrow u$ на $c(v \rightarrow u) - c(u \rightarrow v)$ и удалить ребро $u \rightarrow v$;
 - наконец, если $c(u \rightarrow v) = c(v \rightarrow u)$, то удалить оба ребра $u \rightarrow v$ и $v \rightarrow u$.

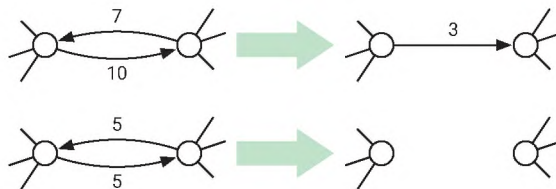


Рис. 10.10. Принудительное применение однонаправленного предположения

- (а) Доказать, что каждый максимальный (s, t) -поток в G' также является максимальным (s, t) -поток в G . (То есть, упрощая каждую

- пару противоположно направленных ребер в G , мы получаем новую редуцированную транспортную сеть с той же величиной максимального потока, что и в G .)
- (b) Доказать, что каждый наименьший (s, t) -разрез в G также является наименьшим (s, t) -разрезом в G' , и наоборот.
 - (c) Доказать, что существует как минимум один максимальный (s, t) -поток в G , который не является максимальным (s, t) -потокom в G' .
5. (a) Описать эффективный алгоритм, определяющий, содержит ли заданная транспортная сеть единственный неповторяющийся максимальный (s, t) -поток.
- (b) Описать эффективный алгоритм, определяющий, содержит ли заданная транспортная сеть единственный неповторяющийся наименьший (s, t) -разрез.
 - (c) Описать транспортную сеть, которая содержит единственный неповторяющийся максимальный (s, t) -поток, но не содержит единственный неповторяющийся наименьший (s, t) -разрез.
 - (d) Описать транспортную сеть, которая содержит единственный неповторяющийся наименьший (s, t) -разрез, но не содержит единственный неповторяющийся максимальный (s, t) -поток.
6. (s, t) -поток в сети G является ациклическим, если в этой сети не существует направленных циклов, в которых каждое ребро имеет положительную величину потока, т. е. подграф из ребер с положительной величиной потока является направленным ациклическим графом.
- (a) Описать и проанализировать алгоритм вычисления ациклического максимального (s, t) -потока в заданной транспортной сети. Ваш алгоритм должен иметь то же асимптотическое время выполнения, что и алгоритм Форда–Фалкерсона.
 - (b) Описать и проанализировать алгоритм, определяющий, является ли каждый максимальный (s, t) -поток в заданной транспортной сети ациклическим.
7. Пусть $G = (V, E)$ – транспортная сеть, в которой каждое ребро имеет пропускную способность 1 и расстояние кратчайшего пути из s в t не меньше d .
- (a) Доказать, что величина максимального (s, t) -потока не превышает E/d .
 - (b) Теперь предположим, что сеть G простая, т. е. для всех вершин u и v существует не более одного ребра из u в v . (Транспортные сети могут содержать параллельные ребра.) Доказать, что величина максимального (s, t) -потока не превышает $O((V^2/d^2))$. [Подсказка: сколько узлов имеется на среднем уровне дерева поиска в ширину с корнем в s ?]

8. Предположим, что задана транспортная сеть $G = (V, E)$, в которой каждое ребро имеет пропускную способность 1, а также целое число k . Описать и проанализировать алгоритм, идентифицирующий k ребер в G , таких, что после удаления этих k ребер величина максимального (s, t) -потока в оставшемся графе становится минимально возможной.
9. Анализ нашего доказательства теоремы о разбиении потока можно сделать более строгим. Пусть $G = (V, E)$ – произвольная транспортная сеть и f – произвольный (s, t) -поток в G .
 - (а) Доказать, что если $|f| = 0$, то f – взвешенная сумма не более чем $E - V + 1$ направленных циклов, где $f(e) > 0$ для каждого ребра e в каждом из этих циклов.
 - (б) Доказать, что если $|f| > 0$, то f – взвешенная сумма не более чем $E - V + 2$ направленных путей и циклов, где $f(e) > 0$ для каждого ребра e в каждом из этих циклов.
 - (с) Доказать, что обе описанные выше верхние границы являются строгими: существуют графы, в которых некоторые циркуляции невозможно разбить на менее чем $E - V + 1$ циклов, а некоторые потоки невозможно разбить на менее чем $E - V + 2$ путей и циклов. [Подсказка: это легко доказать.]
- ♣10. Наше утверждение о том, что любая линейная комбинация из (s, t) -потоков сама по себе является (s, t) -поток, предполагает, что множество всех (не обязательно выполнимых) (s, t) -потоков в любом графе в действительности определяет вещественное векторное пространство, которое мы можем назвать пространством потока (flow space) графа.
 - (а) Доказать, что пространство потока любого связного графа $G = (V, E)$ имеет размерность $E - V + 2$.
 - (б) Пусть T – любое остовное дерево графа G . Доказать, что следующий набор путей и циклов определяет базис для пространства потока этого графа:
 - единственный неповторяющийся путь в T из s в t ;
 - единственный неповторяющийся цикл в $T \cup \{e\}$ для каждого ребра $e \notin T$.
 - (с) Пусть T – любое остовное дерево графа G и F – лес, полученный удалением любого ребра в T . Доказать, что следующий набор путей и циклов определяет базис для пространства потока этого графа:
 - единственный неповторяющийся путь в $F \cup \{e\}$ из s в t для каждого ребра $e \notin F$, который имеет одну конечную точку в каждой компоненте F ;

- единственный неповторяющийся цикл в $F \cup \{e\}$ для каждого ребра $e \notin F$ с обеими конечными точками в одной и той же компоненте F .
- (d) Доказать или опровергнуть следующее утверждение: каждая связная транспортная сеть имеет базис потока, состоящий исключительно из простых путей из s в t .
11. Разрезы иногда определяются как подмножества ребер графа вместо частичных наборов его вершин. В этой задаче вы будете доказывать, что эти два определения почти равнозначны.
- Мы говорим, что подмножество X (направленных) ребер разделяет s и t , если каждый направленный путь из s в t содержит как минимум одно (направленное) ребро из подмножества X . Для любого подмножества вершин S пусть δS обозначает множество направленных ребер, исходящих из S , т. е. $\delta S := \{u \rightarrow v \mid u \in S, v \notin S\}$.
- (a) Доказать, что если $(S, T) - (s, t)$ -разрез, то δS разделяет s и t .
- (b) Пусть X – произвольное подмножество ребер, разделяющих s и t . Доказать, что существует (s, t) -разрез (S, T) такой, что $\delta S \subseteq X$.
- (c) Пусть X – минимальное подмножество ребер, разделяющих s и t . (Такое подмножество ребер иногда называют связью (bond).) Доказать, что существует (s, t) -разрез (S, T) такой, что $\delta S = X$.
12. Предположим, что вместо пропускных способностей мы рассматриваем сети, в которых каждое ребро $u \rightarrow v$ имеет неотрицательную потребность (demand) $d(u \rightarrow v)$. Теперь (s, t) -поток f является выполнимым, если и только если $f(u \rightarrow v) \geq d(u \rightarrow v)$ для каждого ребра $u \rightarrow v$. (Теперь величины выполнимых потоков могут быть произвольно большими.) Естественной задачей при таких условиях является поиск выполнимого (s, t) -потока с минимальной величиной.
- (a) Описать эффективный алгоритм вычисления выполнимого (s, t) -потока, если в качестве входных данных задан граф, функция потребности и вершины s и t . [Подсказка: найдите поток, который везде ненулевой, затем масштабируйте его в сторону увеличения, чтобы сделать его выполнимым.]
- (b) Предположим, что у вас есть доступ к подпрограмме MaxFlow, вычисляющей максимальные потоки в сетях с пропускными способностями ребер. Описать эффективный алгоритм вычисления минимального потока в заданной сети с потребностями ребер. Алгоритм должен вызывать MaxFlow ровно один раз.
- (c) Сформулировать и доказать теорему, аналогичную теореме о максимальном потоке и наименьшем разрезе (max-flow min-cut) для приведенных выше условий. (Соответствуют ли минимальные потоки максимальным разрезам?)

13. Для любой транспортной сети G и любых вершин u и v пусть $bottleneck_G(u, v)$ обозначает максимальный из всех путей p в G из u в v из ребер с минимальной пропускной способностью вдоль p .
- Описать и проанализировать алгоритм вычисления $bottleneck_G(s, t)$ за время $O(E \log V)$. Это величина потока, которую алгоритм Эдмондса–Карпа насыщенных увеличивающих путей продвигает на первой итерации.
 - Теперь предположим, что транспортная сеть G ненаправленная; равнозначно предположим, что $c(u \rightarrow v) = c(v \rightarrow u)$ для каждой пары вершин u и v . Описать и проанализировать алгоритм вычисления $bottleneck_G(s, t)$ за время $O(V+E)$. [Подсказка: найти среднюю пропускную способность ребер.] Почему это ускорение выполнения не работает для направленных графов?
 - ❖ Снова предположим, что транспортная сеть G ненаправленная. Описать и проанализировать алгоритм формирования остовного дерева T графа G , такого что $bottleneck_T(u, v) = bottleneck_G(u, v)$ для всех вершин u и v . (Ребра в T наследуют пропускные способности из G .) Для получения наивысшей оценки алгоритм должен выполняться за время $O(E)$.
14. Предположим, что задана транспортная сеть G с целочисленными пропускными способностями и целочисленным максимальным потоком f^* в G . Описать алгоритмы для выполнения следующих операций:
- Increment(e) – увеличения пропускной способности ребра e на 1 и обновления максимального потока;
 - Decrement(e) – уменьшения пропускной способности ребра e на 1 и обновления максимального потока.
- Оба алгоритма должны изменять f^* так, чтобы он оставался максимальным потоком, быстрее, чем повторное вычисление максимального потока с нуля.
15. Пусть G – сеть с целочисленными пропускными способностями ребер. Ребро в G является связанным сверху (upper-binding), если увеличение его пропускной способности на 1 также увеличивает величину максимального потока в G . Аналогично ребро является связанным снизу (lower-binding), если уменьшение его пропускной способности на 1 также уменьшает величину максимального потока в G .
- Каждая ли сеть G содержит хотя бы одно связанное сверху ребро? Докажите, что ваш ответ правильный.
 - Каждая ли сеть G содержит хотя бы одно связанное снизу ребро? Докажите, что ваш ответ правильный.
 - Описать алгоритм поиска всех связанных сверху ребер в G , если в качестве входных данных задана сеть G и максимальный поток в G , за время $O(E)$.

- (d) Описать алгоритм поиска всех связанных снизу ребер в G , если в качестве входных данных задана сеть G и максимальный поток в G , за время $O(EV)$.
16. Задана транспортная сеть G , возможно, содержащая более одного наименьшего (s,t) -разреза. Определим наилучший наименьший (s,t) -разрез как любой наименьший разрез (S, T) с наименьшим числом ребер, проходящих из S в T .
- (a) Описать эффективный алгоритм поиска наилучшего наименьшего (s,t) -разреза, если пропускные способности являются целыми числами.
- (b) Описать эффективный алгоритм поиска наилучшего наименьшего (s,t) -разреза для произвольных пропускных способностей ребер.
- (c) Описать эффективный алгоритм, определяющий, содержит ли заданная транспортная сеть единственный неповторяющийся наилучший наименьший (s,t) -разрез.
17. Новый старший преподаватель, читающий курс поиска максимальных потоков впервые, предлагает следующую жадную модификацию обобщенного алгоритма Форда–Фалкерсона увеличивающего пути. Вместо сопровождения остаточного графа нужно просто⁵ сократить пропускную способность ребер в увеличивающем пути. В частности, когда мы насыщаем ребро, то просто удаляем его из графа. Кому нужен этот бессмысленный остаточный граф?

```

GreedyFlow( $G, c, s, t$ ):
  for every edge  $e$  in  $G$ 
     $f(e) \leftarrow 0$ 
  while there is a path from  $s$  to  $t$ 
     $\pi \leftarrow$  an arbitrary path from  $s$  to  $t$ 
     $F \leftarrow$  minimum capacity of any edge in  $\pi$ 
    for every edge  $e$  in  $\pi$ 
       $f(e) \leftarrow f(e) + F$ 
      if  $c(e) = F$ 
        remove  $e$  from  $G$ 
      else
         $c(e) \leftarrow c(e) - F$ 
  return  $f$ 

```

- (a) Показать, что алгоритм GreedyFlow не всегда вычисляет максимальный поток.

⁵ Наречие «просто» (здесь: англ. – just) почти всегда является подсознательным сокращением фразы: «Я слишком ленив, чтобы объяснять все подробности, но вы все равно должны верить мне» или еще короче: «Вероятно, это неправильно». Смотрите также: merely, simply, clearly и obviously.

- (b) Показать, что алгоритм GreedyFlow даже не гарантирует вычисление правильной аппроксимации к максимальному потоку. То есть для любой константы $\alpha > 1$ существует транспортная сеть G такая, что величина максимального потока более чем в α раз превышает величину потока, вычисленного алгоритмом GreedyFlow. [Подсказка: предположите, что GreedyFlow выбирает наихудший возможный путь π на каждой итерации.]

18. В 1980 г. Морис Кейран (Maurice Queyranne) опубликовал пример транспортной сети, показанной на рис. 10.11, где эвристический метод Эдмондса и Карпа «самого насыщенного (увеличивающего) пути» никогда не останавливается. Как и в примере плохой сети Цвика для исходного алгоритма Форда–Фалкерсона, ϕ обозначает величину, обратную золотому сечению $(\sqrt{5} - 1)/2$. Три вертикальных ребра играют точно такую же роль, что и горизонтальные ребра в примере Цвика.

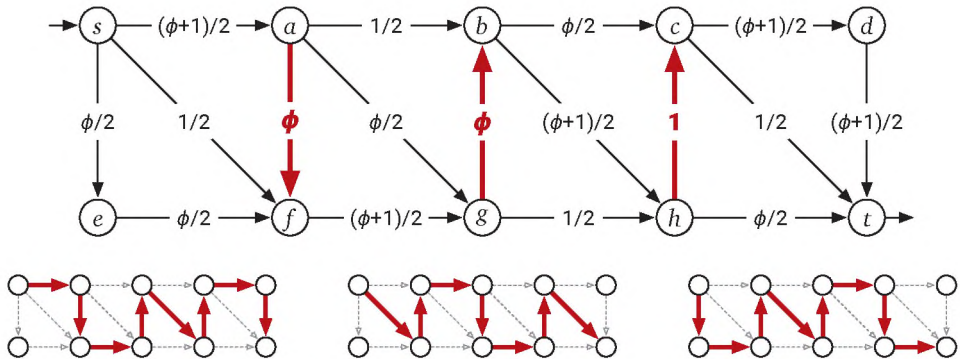


Рис. 10.11. Сеть Кейрана и последовательность «самых насыщенных увеличивающих путей»

- (a) Показать, что следующая бесконечная последовательность увеличивающих путей является вполне допустимым выполнением алгоритма Эдмондса–Карпа «самого насыщенного (увеличивающего) пути» (см. нижнюю часть рис. 10.11).

QueyranneFatPaths:

for $i \leftarrow 1$ to ∞

push ϕ^{3i-2} units of flow along $s \rightarrow a \rightarrow f \rightarrow g \rightarrow b \rightarrow h \rightarrow c \rightarrow d \rightarrow t$

push ϕ^{3i-1} units of flow along $s \rightarrow f \rightarrow a \rightarrow b \rightarrow g \rightarrow h \rightarrow c \rightarrow t$

push ϕ^{3i} units of flow along $s \rightarrow e \rightarrow f \rightarrow a \rightarrow g \rightarrow b \rightarrow c \rightarrow h \rightarrow t$

- (b) Описать последовательность $O(1)$ увеличивающих путей, которая создает максимальный поток в сети Кейрана.

- ♥19. (s, t) -последовательно-параллельный граф – это направленный ациклический граф с двумя различными вершинами s и t и с одной из следующих структур. Это:

- *базовый случай*: единственное направленное ребро из s в t ;
- *последовательность*: объединение (s,u) -последовательно-параллельного графа и (u,t) -последовательно-параллельного графа, которые совместно используют общую вершину u , но не какие-либо другие вершины и ребра;
- *параллельность*: объединение двух меньших (s,t) -последовательно-параллельных графов с одним и тем же источником s и целью t , но без каких-либо других общих вершин и ребер.

Каждый (s,t) -последовательно-параллельный граф G может быть представлен деревом разбиения, которое является двоичным деревом с тремя типами узлов: листьями (соответствующими ребрам в G), последовательными узлами (соответствующими вершинам, отличающимся от s и t) и параллельными узлами. Один последовательно-параллельный граф можно представить многими различными деревьями разбиения.

- (а) Предположим, что задан направленный граф G с двумя особыми вершинами s и t . Описать и проанализировать алгоритм, который либо создает дерево разбиения для G , либо корректно сообщает, что G не является (s,t) -последовательно-параллельным графом. [Подсказка: создавайте это дерево снизу вверх.]
- (б) Описать и проанализировать алгоритм вычисления максимального (s,t) -потока в заданной (s,t) -последовательно-параллельной транспортной сети с произвольными пропускными способностями ребер. [Подсказка: с учетом части (а) вы можете предположить, что в действительности вам задано дерево разбиения. Сначала вычислите величину максимального потока, затем – действительный максимальный поток.]

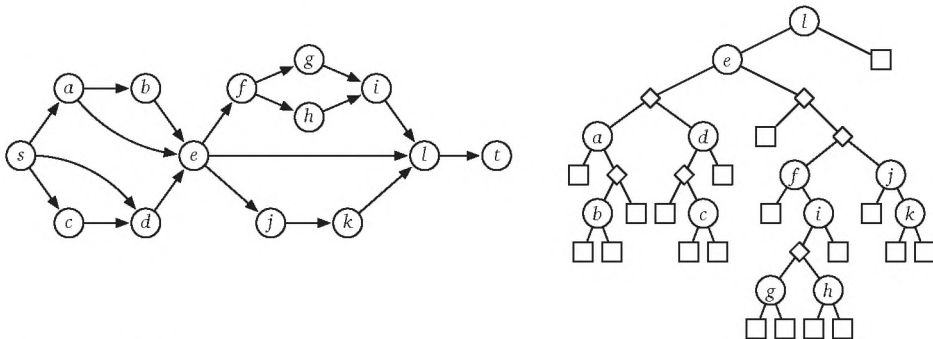


Рис. 10.12. Последовательно-параллельный граф и соответствующее дерево разбиения. Квадраты в дереве разбиения – листья, ромбы – параллельные узлы

20. Мы можем ускорить алгоритм Эдмондса–Карпа «самого насыщенного увеличивающего пути», по крайней мере, для сетей с малыми целочисленными пропускными способностями, ослабив наши

требования для следующего увеличивающего пути. Вместо поиска увеличивающего пути с максимальной пропускной способностью узкого места мы ищем путь, пропускная способность узкого места которого равна как минимум половине максимума, используя для этого описанный ниже алгоритм масштабирования пропускных способностей (capacity scaling). (Этот алгоритм действительно был предложен Эдмондсом и Карпом.)

Предположим, что пропускные способности всех ребер являются положительными целыми числами, меньшими $U = 2^k$ при некотором целом k . Алгоритм масштабирования поддерживает пороговое значение узкого места Δ . Изначально мы устанавливаем $\Delta \leftarrow U$. На каждом этапе алгоритм продвигается вдоль путей из s в t , в которых каждое ребро имеет остаточную пропускную способность не менее Δ . Если такого пути нет, то текущий этап завершается, мы устанавливаем $\Delta \leftarrow \lfloor \Delta/2 \rfloor$ и начинается следующий этап. Алгоритм завершается, когда $\Delta = 0$.

- (a) Сколько этапов выполнит этот алгоритм в наихудшем случае?
- (b) Пусть f – поток в конце этапа при конкретном значении Δ . Доказать, что пропускная способность наименьшего разреза в остаточном графе G_f не превышает $E \cdot \Delta$.
- (c) Доказать, что на каждом этапе алгоритма масштабирования существует не более $2E$ увеличивающих путей.
- (d) Каково общее время выполнения алгоритма масштабирования пропускных способностей?

Глава 11

Приложения потоков и разрезов

Долгое время меня удивляло, как такая дорогая, такая революционная вещь может быть столь недееспособной, но потом я понял, что компьютер — глупая машина со способностью делать что-то невероятно умное, а программисты — умные люди со способностью делать невероятно глупые вещи. В общем, они идеально дополняют друг друга.

— Билл Брайсон (Bill Bryson), «Notes from a Big Country» (1999)
(Брайсон Б. Страна Дяди Сэма. Привет, Америка! = Notes from a Big Country / пер. с англ. Е. Габитбаевой. М. СПб.: Эксмо. Мидгард, 2009. 415 с.
(Биографии великих стран: BBC.) ISBN 978-5-699-34020-0

Вскоре после падения «железного занавеса» в 1990 году встретились американец и русский, которые оба работали над созданием оружия. Американец спросил: «Когда вы разрабатывали бомбу, как вы смогли выполнить такое огромное количество вычислений на своих слабых компьютерах?» Русский ответил: «Мы использовали более совершенные алгоритмы».

— Ефим Диниц (Yefim Dinitz), в статье «Dinitz' Algorithm: The Original Version and Even's Version» (2006)

*Так что теперь я плохой парень, потому что я не плыву в потоке.
Никогда не плыви в потоке по течению, сам будь потоком.*

— Джей-Зи (Jay-Z), «Поток сознания»
(«Stream of Consciousness»), 16 мая 2015 г.

11.1. Реберно-непересекающиеся пути

Одним из самых ранних приложений алгоритмов поиска максимальных потоков было вычисление максимального числа реберно-непересекающихся путей между двумя конкретно заданными вершинами s и t в направленном графе G с использованием максимальных потоков. Множество путей в G является реберно-непересекающимся, если каждое ребро в G

встречается не более одного раза в этих путях, но несколько реберно-непересекающихся путей могут проходить через одну и ту же вершину.

Если мы присвоим каждому ребру пропускную способность 1, то максимальный поток из s в t продвигает либо 0, либо 1 элемент потока вдоль каждого ребра. Теорема о разбиении потока предполагает, что подграф S из насыщенных ребер является объединением нескольких реберно-непересекающихся путей и циклов. Кроме того, число путей в этом разбиении в точности равно величине потока. Извлечение реальных путей из S просто и очевидно – нужно следовать по любому направленному пути в S из s в t , удалить этот путь из S и применить рекурсию.

Обратная операция: мы можем преобразовать любой набор из k реберно-непересекающихся путей в поток, продвигая один элемент потока вдоль каждого пути из s в t . Величина полученного потока равна в точности k . Из этого следует, что любой алгоритм поиска максимального потока в действительности вычисляет наибольшее возможное множество реберно-непересекающихся путей.

Если мы воспользуемся алгоритмом Орлина для вычисления максимальных потоков, то сможем вычислить реберно-непересекающиеся пути за время $O(VE)$, но алгоритм Орлина избыточен для такого простого приложения. Пропускная способность разреза $(\{s\}, V \setminus \{s\})$ не превышает $V - 1$, поэтому величина максимального потока не превышает $V - 1$. Таким образом, исходный алгоритм Форда–Фалкерсона поиска увеличивающего пути уже выполняется за время $O(|f^*| E) = O(VE)$.

Тот же алгоритм можно использовать для поиска реберно-непересекающихся путей в ненаправленных графах. Во-первых, нужно заменить каждое ненаправленное ребро uv в G на пару направленных ребер $u \rightarrow v$ и $v \rightarrow u$ с единичной пропускной способностью каждое, затем назвать полученный направленный граф G' . Далее вычислить максимальный (s, t) -поток f^* в G' , используя алгоритм Форда–Фалкерсона. Для любого ребра uv в G если f^* насыщает оба направленных ребра $u \rightarrow v$ и $v \rightarrow u$ в G' , то мы можем удалить оба эти ребра из потока без изменения его величины. (Более обобщенно: мы можем найти ациклический максимальный поток в G' , удаляя все циклы в f^* , а не только циклы длиной 2.) Таким образом, без потери для обобщения f^* назначает неповторяющееся направление для каждого насыщенного ребра. Наконец, мы можем извлечь реберно-непересекающиеся пути, выполнив поиск в подграфе из направленных ребер, насыщенных f^* .

11.2. Пропускные способности вершин и вершинно-непересекающиеся пути

Теперь предположим, что не только ребра, но и вершины в исходном графе G имеют пропускные способности. В дополнение к нашим прочим ограничениям для каждой вершины v , отличающейся от вершин s и t , мы тре-

буем, чтобы суммарный поток в v (следовательно, и суммарный поток из v) не превышал некоторой неотрицательной величины $c(v)$:

$$\sum_{u \rightarrow v} f(u \rightarrow v) \leq c(v).$$

Сможем ли мы вычислить максимальные потоки при этих новых ограничениях для вершин?

В 1962 г. Форд и Фалкерсон предложили следующую редукцию для транспортной сети $\bar{G}$ с пропускными способностями только для ребер. Нужно заменить каждую вершину v на две вершины v_{in} и v_{out} , соединенные ребром $v_{in} \rightarrow v_{out}$ с пропускной способностью $c(v)$, затем заменить каждое направленное ребро $u \rightarrow v$ на ребро $u_{out} \rightarrow v_{in}$ (сохраняя ту же пропускную способность). Подпрограмма определения-поиска предполагает, что каждый выполнимый (s_{out}, t_{in}) -поток в $\bar{G}$ равнозначен выполнимому (s, t) -потoku с той же величиной в исходном графе G , и наоборот. В частности, каждый максимальный поток в $\bar{G}$ равнозначен максимальному потоку в G . Редукция из G в $\bar{G}$ занимает время $O(E)$, после чего мы можем вычислить максимальный поток в $\bar{G}$, используя алгоритм Орлина. В общей сложности вычисление максимального потока в G требует времени $O(VE)$.

Теперь легко вычислить максимальное число вершинно-непересекающихся путей из s в t за время $O(VE)$: нужно присвоить пропускную способность 1 каждой вершине и вычислить максимальный поток.

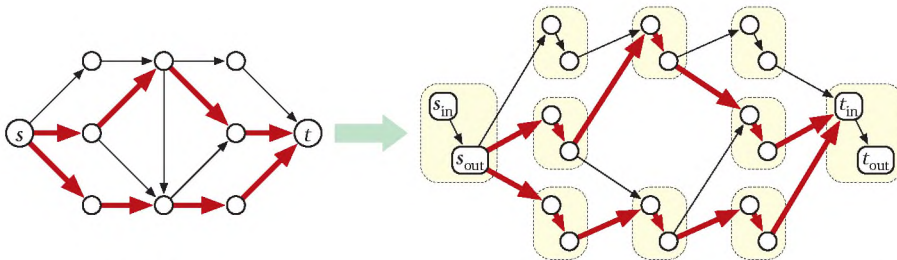


Рис. 11.1. Редукция задачи поиска вершинно-непересекающихся путей в графе G к задаче поиска реберно-непересекающихся путей в графе $\bar{G}$

11.3. Задача о паросочетании в двудольном графе

Еще одним естественным применением алгоритмов поиска максимальных потоков является поиск максимальных паросочетаний в двудольных графах. Паросочетание – это подграф, в котором степень каждой вершины не превышает единицы, или равнозначное определение: набор ребер таких, что никакие два ребра не разделяют одну вершину. Задача состоит в поиске паросочетаний с максимальным числом ребер.

Например, предположим, что имеется множество врачей, ищущих работу, и множество больниц, ищущих врачей. Каждый врач составил список всех больниц, в которых он хотел бы работать, а каждая больница составила список врачей, которых желательно принять на работу. Наша задача – найти наибольшее подмножество пар врач–больница, приемлемых для обеих сторон.¹ Эта задача равнозначна поиску максимального паросочетания в двудольном графе, вершинами которого являются врачи и больницы, а ребро между врачом и больницей существует, если и только если каждая сторона считает другую приемлемым вариантом выбора.

Мы можем решить эту задачу, редуцируя ее к задаче поиска максимального потока, как показано ниже. Пусть G – заданный двудольный граф с множеством вершин $L \cup R$ таким, что каждое ребро соединяет вершину в L с вершиной в R . Мы создаем новый направленный граф G' , (1) ориентируя каждое ребро из L в R , (2) добавляя новую вершину-источник s с ребрами в каждую вершину в L , (3) добавляя новую целевую вершину t с ребрами из каждой вершины в R . Наконец, мы присваиваем каждому ребру в G' пропускную способность 1.

Любое паросочетание M в G можно преобразовать в поток f_M в G' , как показано ниже. Для каждого ребра uv в M один элемент потока продвигается вдоль пути $s \rightarrow u \rightarrow v \rightarrow t$. Эти пути являются непересекающимися за исключением вершин s и t , поэтому полученный поток соблюдает ограничения пропускной способности. Кроме того, величина полученного потока равна числу ребер в M .

Наоборот, рассмотрим любой (s, t) -поток f в G' , вычисленный с использованием алгоритма Форда–Фалкерсона поиска увеличивающего пути. Поскольку пропускные способности ребер являются целыми числами, алгоритм Форда–Фалкерсона присваивает целочисленный поток каждому ребру. (Подсказка: это легко проверить методом индукции.) Кроме того, поскольку каждое ребро имеет единичную пропускную способность, вычисленный поток либо насыщает ($f(e) = 1$) каждое ребро, либо избегает ($f(e) = 0$) его в G' . Наконец, поскольку не более одного элемента потока может входить в каждую вершину в U или исходить из каждой вершины в W , насыщенные ребра из U в W формируют паросочетание в G . Размер этого паросочетания в точности равен $|f|$.

Таким образом, размер максимального паросочетания в G равен величине максимального потока в G' , и, предполагая, что максимальный поток вычисляется с использованием увеличивающих путей, мы можем преобразовать действительный максимальный поток в максимальное паросочетание за время $O(E)$. Мы можем вычислить максимальный поток за время $O(VE)$, используя либо алгоритм Орлина, либо алгоритм Форда–Фалкерсона «из коробки».

¹ Эта задача полностью отличается от задачи поиска устойчивого соответствия, которую мы рассматривали в главе 4, потому что здесь мы пытаемся сделать каждого врача и каждую больницу максимально удовлетворенными выбором.

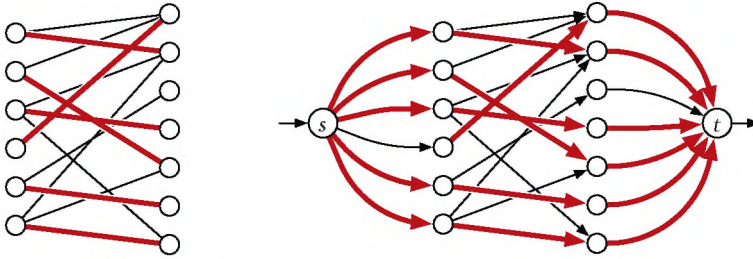


Рис. 11.2. Максимальное паросочетание в двудольном графе G и соответствующий максимальный поток в графе G'

Ниже приведено подробное объяснение поведения алгоритма Форда–Фалкерсона в графе G' с точки зрения исходного двудольного графа G . Алгоритм обрабатывает паросочетание M в G , которое изначально пустое. Ребра M соответствуют ребрам в G' , несущим поток. Назовем вершину в G паросочетающейся, если она является конечной точкой некоторого ребра в M , и непаросочетающейся в противном случае. На каждой итерации алгоритма мы ищем чередующуюся цепь (alternating path) в G – путь из непаросочетающейся вершины в L в непаросочетающуюся вершину в R , где чередуются ребра в M и ребра, которых нет в M . (Чередующиеся цепи в G в точности соответствуют увеличивающим путям для текущего потока в G' .) Если мы находим увеличивающий путь P , то обновляем M до симметричной разности $M \oplus P$, которая увеличивает число ребер в M на 1, и переходим к следующей итерации. Если увеличивающий путь не существует, то по теореме максимального потока и наименьшего разреза предполагается, что M является максимальным паросочетанием, и алгоритм завершает работу. Поиск одного увеличивающего пути требует времени $O(E)$, и алгоритм останавливается после не более чем V итераций, поэтому весь алгоритм в целом выполняется за время $O(VE)$. На рис. 11.3 показана схема работы этого алгоритма.

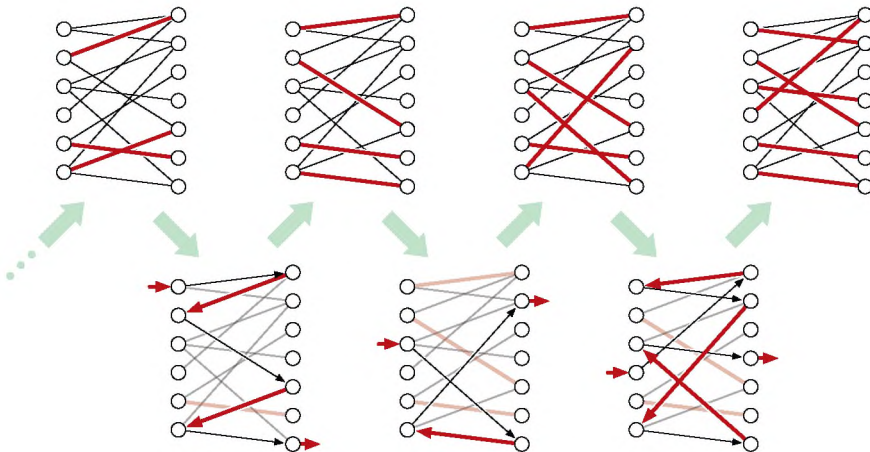


Рис. 11.3. Возрастающая последовательность паросочетаний, соединенных чередующимися цепями

Эта характеристика задачи поиска максимальных паросочетаний в двудольных графах в терминах чередующихся цепей была предложена Клодом Бержем (Claude Berge) в 1957 г. (независимо от теоремы максимального потока и наименьшего разреза), хотя она неявно использовалась уже в алгоритмах, описанных Харальдом Куном (Harald Kuhn) в 1955 г., Денешем Кёнигом (Dénes Kőnig) в 1916 г. и Карлом Якоби (Carl Jacobi) около 1836 г.

Более хитроумный алгоритм, предложенный Джоном Хопкрофтом (John Hopcroft) и Ричардом Карпом (Richard Karp) в 1973 г., вычисляет максимальные паросочетания в двудольных графах всего лишь за время $O(\sqrt{V}E)$, выполняя на каждой итерации поиск нескольких непересекающихся чередующихся цепей.

11.4. Выбор кортежа

Задача поиска максимальных паросочетаний в двудольных графах – это простейший пример из более широкого класса задач, которые я называю *выбором кортежа* (tuple selection)². Входные данные для задачи выбора кортежа состоят из нескольких конечных множеств $X_1, X_2, \dots, X_d$, представляющих различные дискретные ресурсы. Наша задача – выбрать максимальное возможное множество d -кортежей, каждый из которых содержит ровно один элемент из каждого множества X_i , с учетом нескольких ограничений пропускной способности, изложенных в следующей форме:

- для каждого индекса i каждый элемент $x \in X_i$ может присутствовать не более чем в $c(x)$ выбранных кортежах;
- для каждого индекса i любые два элемента $x \in X_i$ и $y \in X_{i+1}$ могут присутствовать не более чем в $c(x, y)$ выбранных кортежах.

Каждая из приведенных здесь верхних границ $c(x)$ и $c(x, y)$ является либо неотрицательным (обычно малым) целым числом, либо ∞ .

В задаче поиска максимального паросочетания мы имеем $d = 2$ ресурса, каждый элемент x имеет пропускную способность $c(x) = 1$, а каждая пара (x, y) имеет пропускную способность $c(x, y) = 1$ или $c(x, y) = 0$ в зависимости от того, является или не является x ребром во внутреннем двудольном графе.

Поскольку ресурсы линейно упорядочены и ограничиваются только пары объектов в смежных подмножествах X_i и X_{i+1} ³, задачу выбора кортежа можно привести к задаче поиска максимального потока в направленном графе G , определенной следующим образом:

² Я не смог найти стандартного названия для этих задач, поэтому придумал свое. Их иногда называют «задачами присваивания» (assignment problems), но в более общем смысле термин «the assignment problem» обозначает задачу поиска паросочетания с максимальным весом в двудольном графе с взвешенными ребрами.

³ Если пары объектов хотя бы из одной несмежной пары подмножеств $(X_i \text{ и } X_j, \text{ где } j > i + 1)$ также ограничены, то задача становится NP-трудной по очевидной редукции от Exact3Dimensional-Matching. NP-трудность мы будем рассматривать в следующей главе.

- граф G содержит вершину для каждого элемента из каждого множества X_i , а также вершину-источник s и целевую вершину t . Каждая вершина (исключая s и t) имеет пропускную способность $c(x)$;
- граф G содержит ребро $s \rightarrow w$ для каждого элемента $w \in X_1$, ребро $z \rightarrow t$ для каждого элемента $z \in X_d$ и ребро $x \rightarrow y$ с пропускной способностью $c(x, y)$ для каждой пары элементов $x \in X_i$ и $y \in X_{i+1}$ для всех i . (Дополнительно (но не обязательно) мы можем исключить ребра $x \rightarrow y$ с пропускной способностью $c(x, y) = 0$.)

Каждый путь из s в t в графе G соответствует (или «является» (is)) d -кортежем, который мы могли бы выбрать. Обратно: каждый потенциально выбираемый d -кортеж, соблюдающий установленные ограничения, соответствует (или «является» (is)) пути из s в t в графе G .

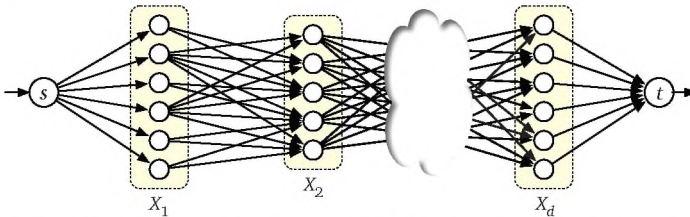


Рис. 11.4. Транспортная сеть для задачи выбора кортежа

Более обобщенно: пусть f – произвольный выполнимый целочисленный (s, t) -поток в графе G . Поскольку все пропускные способности являются целыми числами или ∞ , по теореме о разбиении потока предполагается, что f равен сумме $|f|$ путей из s в t , при этом каждый путь несет ровно один элемент потока. По очевидному определению-поиску предполагается, что полученное в результате множество кортежей соблюдает все ограничения пропускной способности. Обратно: для любого множества из k кортежей, соблюдающего ограничения пропускной способности, сумма k соответствующих путей является выполнимым целочисленным (s, t) -поток с величиной k .

Таким образом, мы можем выбрать максимальное число кортежей, соблюдающих заданные ограничения пропускной способности, вычислив максимальный (s, t) -поток f^* в графе G , а затем вычислив разбиение потока f^* . Поскольку все конечные пропускные способности в G являются целыми числами, мы можем предположить без потери для обобщения, что f^* – целочисленный поток, следовательно (из предыдущего абзаца), он соответствует допустимому корректному множеству $|f^*|$ кортежей.

Расписание экзаменов

Следующая задача составления расписания «в реальной жизни» может помочь прояснить нашу обобщенную редукцию.

Университет Шам-Пубанана (Sham-Poobanana University) нанял вас для написания алгоритма составления расписания выпускных экзаменов. Су-

существует n различных учебных курсов, для каждого из которых требуется составить расписание выпускных экзаменов в одной из r аудиторий в течение одного из t различных интервалов (слотов) времени. В каждом интервале времени в каждой аудитории можно запланировать не более одного выпускного экзамена по учебному курсу и обратно: учебные курсы нельзя разделять по разным аудиториям и по нескольким интервалам времени. Кроме того, за каждым экзаменом обязательно должен наблюдать один из p инспекторов⁴. Каждый инспектор может наблюдать одновременно не более чем за одним экзаменом. Каждый инспектор доступен только в определенные интервалы времени. Ни одному из инспекторов не разрешается наблюдать в общей сложности за более чем пятью экзаменами. Входные данные для этой задачи составления расписания состоят из трех массивов:

- целочисленного массива $E[1..n]$, где $E[i]$ – число студентов, обучающихся по i -му учебному курсу;
- целочисленного массива $S[1..r]$, где $S[j]$ – число мест в j -й аудитории. Выпускной экзамен i -го учебного курса можно провести в j -й аудитории, если и только если $E[i] \leq S[j]$;
- массива логических значений $A[1..t, 1..p]$, где $a[k, \ell] = \text{True}$, если и только если ℓ -й инспектор доступен в течение k -го интервала времени.⁵

Пусть $N = n + r + tp$ обозначает общий размер входных данных. Ваша задача – спроектировать алгоритм, который либо планирует (определяет пункт расписания) аудиторию, интервал времени и инспектора для выпускного экзамена по каждому учебному курсу, либо корректно сообщает, что такое расписание невозможно.

Это стандартная задача выбора кортежа с четырьмя ресурсами: учебными курсами, аудиториями, интервалами времени и инспекторами. Для решения этой задачи мы формируем транспортную сеть G с шестью типами вершин – вершина-источник s' , вершина c_i для каждого учебного курса, вершина r_j для каждой аудитории, вершина t_k для каждого интервала времени, вершина p_ℓ для каждого инспектора и целевая вершина t' – и пять типов ребер, которые показаны на рис. 11.5:

- ребро $s' \rightarrow c_i$ с пропускной способностью 1 для каждого учебного курса i («Каждый учебный курс может завершаться не более чем одним выпускным экзаменом»);
- ребро $c_i \rightarrow r_j$ с пропускной способностью ∞ для каждого учебного курса i и аудитории j таких, что $E[i] \leq S[j]$. («По учебному курсу i можно провести экзамен в аудитории j , если и только если в этой аудитории достаточно посадочных мест».) Это единственный случай, когда используется количество студентов $E[i]$ и число посадочных мест в аудитории $S[j]$;

⁴ В США – proctor, за пределами США – invigilator.

⁵ Вероятно, эту информацию лучше представить в форме графа, но я думаю, что это в большей степени запутает процесс редукции.

- ребро $r_j \rightarrow t_k$ с пропускной способностью 1 для каждой аудитории j и интервала времени k . («В аудитории j в течение интервала времени k можно провести не более одного экзамена»);
- ребро $t_k \rightarrow p_\ell$ с пропускной способностью 1 для интервала времени k и инспектора ℓ таких, что $A[\ell, k] = \text{True}$. («Инспектор может наблюдать не более одного экзамена в любой интервал времени и только в те интервалы времени, когда он доступен»);
- ребро $p_\ell \rightarrow t'$ с пропускной способностью 5 для каждого инспектора ℓ . («Каждый инспектор может наблюдать не более пяти экзаменов (в сумме)»).

(Я называю вершину-источник и целевую вершину s' и t' , а не s и t лишь потому, что в формулировке задачи уже используется переменная t для обозначения числа интервалов времени.) В целом G содержит $n + r + t + p + 2 = O(N)$ вершин и $O(nr + rt + tp) = O(N^2)$ ребер.

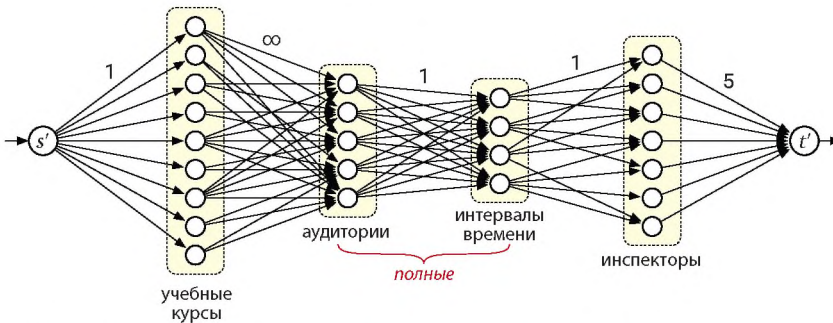


Рис. 11.5. Транспортная сеть для задачи составления расписания экзаменов

Каждый путь из s' в t' в G представляет неповторяющийся вариант выбора курса, аудитории, интервала времени и инспектора для одного выпускного экзамена, в частности, для курса подходит аудитория, а инспектор доступен в это время. Обратно для каждого допустимого варианта выбора (курс, аудитория, интервал времени, инспектор) существует соответствующий путь из s' в t' в G . Таким образом, мы можем сформировать корректное расписание для максимально возможного числа экзаменов, вычислив максимальный (s, t) -поток f^* в G , разбив поток f^* на пути из s' в t' , а затем преобразовав каждый путь в представление курс–аудитория–время–инспектор. Если $|f^*| = 0$, то мы можем вернуть полученное расписание, иначе можно корректно сообщить, что такое расписание для всех n выпускных экзаменов составить невозможно.

Создание графа G из заданных входных данных методом грубой силы (прямым перебором) занимает время $O(E)$. Мы можем вычислить максимальный поток за время $O(VE)$, используя либо алгоритм Форда–Фалкерсона (потому что $|f^*| \leq n < V$), либо алгоритм Орлина, а разбиение потока можно вычислить за время $O(VE)$. Таким образом, общее время выполнения алгоритма $O(VE) = O(N^5)$.

11.5. Покрывтия непересекающихся путей

Покрывтие путей (path cover) направленного графа G – это набор направленных путей в G таких, что каждая вершина G расположена как минимум в одном пути. Покрывтие непересекающихся путей (disjoint-path cover) графа G – это покрытие путей такое, что каждая вершина G расположена ровно в одном пути. Каждый направленный граф имеет тривиальное покрытие непересекающихся путей, состоящее из нескольких путей нулевой длины, но это скучный вариант. Вместо него рассмотрим покрытия непересекающихся путей, которые содержат минимально возможное число путей. Эта задача является NP-трудной в общем случае – граф имеет покрытие непересекающихся путей размером 1, если и только если он содержит гамильтонов путь, – но существует эффективный алгоритм на основе потоков для направленных ациклических графов.

Чтобы решить эту задачу для заданного направленного ациклического графа $G = (V, E)$, создадим новый двудольный граф $G' = (V', E')$, как показано ниже.

- H содержит две вершины v^i и v^f для каждой вершины v графа G .
- H содержит ненаправленное ребро $u^i v^f$ для каждого направленного ребра $u \rightarrow v$ в G .

(Если граф G представлен как матрица смежности, то G' – двудольный граф, представленный той же матрицей смежности.)

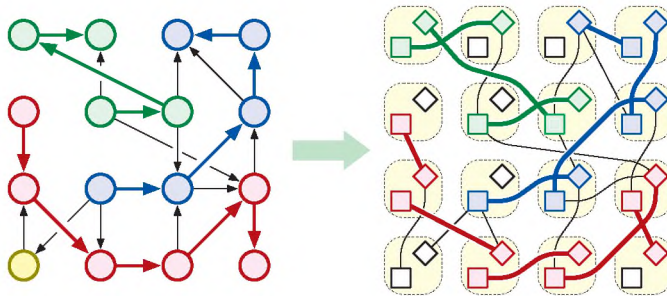


Рис. 11.6. Редукция задачи минимального покрытия непересекающихся путей в НАГ к задаче о паросочетании в двудольном графе.

Квадраты обозначают бемоли ♭, а ромбы – дизезы #

Теперь я утверждаю, что граф G можно покрыть k непересекающимися путями, если и только если новый граф G' содержит паросочетание размером $V - k$. Как обычно, мы доказываем равнозначность в два этапа.

⇐ Предположим, что G содержит покрытие непересекающихся путей P с k путями, будем считать P подграфом G . Каждая вершина в P имеет степень входа 0 или 1. Кроме того, существует ровно одна вершина со степенью входа 0 в каждом пути в P . Из этого следует, что P содержит

ровно $V - k$ ребер. Теперь определим подмножество M ребер G' , как показано ниже:

$$M := \{ u \rightarrow v \in E' \mid u \rightarrow v \in P \}.$$

По определению покрытия непересекающихся путей каждая вершина G имеет не более одного входящего ребра в P и не более одного исходящего ребра в P . Отсюда выводим, что каждая вершина G' инцидентна не более чем одному ребру в M , т. е. M является паросочетанием размером $V - k$.

⇒ Предположим, что G' содержит паросочетание M размером $V - k$. Отобразим M' обратно в G , определив подграф $P = (V, M')$, где

$$M' := \{ u \rightarrow v \in E \mid u \rightarrow v \in M \}.$$

По определению паросочетания каждая вершина G содержит не более одного входящего ребра в P и не более одного исходящего ребра в P . Из этого следует, что P является набором непересекающихся направленных путей в G . Поскольку P включает каждую вершину, P определяет покрытие непересекающихся путей с $V - k$ ребрами. Число путей в P равно числу вершин в G , которые не имеют входящего ребра в M' . Отсюда выводим, что P содержит ровно k путей.

Из этого сразу же следует, что мы можем найти минимальное покрытие непересекающихся путей в G , вычислив максимальное паросочетание в G' , используя алгоритм Форда–Фалкерсона вычисления максимального потока, за время $O(VE)$.

Несмотря на то что задача сформулирована в терминах НАГов и путей, в действительности это задача поиска паросочетания: нам необходимо найти паросочетания как можно большего числа вершин для различения следующих друг за другом элементов в графе. Число путей, требуемое для покрытия НАГ, равно числу вершин без последующих элементов. (И разумеется, каждая задача поиска паросочетания в двудольном графе в действительности является задачей поиска потока.)

Набор минимального преподавательского состава

Вернемся в университет Шам-Пубанана для другой задачи планирования «из реальной жизни»⁶. Ежедневно университет предлагает несколько тысяч учебных курсов. Из-за экстремального сокращения бюджета университет вынужден существенно сократить численность преподавательского состава. Но, поскольку студенты оплачивают обучение (а университет не может позволить себе держать в штате юристов), необходимо оставить достаточное число преподавателей, чтобы обеспечить реальное обучение по каждо-

⁶ Чтобы придать чуть больше реалистичности (и уменьшить негативные эмоции) формулировке этой задачи, рассматривайте самолеты и их рейсы или автобусы и их маршруты вместо преподавателей и учебных курсов.

му курсу, заявленному в каталоге. Сколько преподавателей может уволить университет? Каждому оставшемуся члену преподавательского состава будет назначена последовательность учебных курсов для преподавания в каждый заданный день. Курсы, распределяемые для каждого преподавателя, не должны пересекаться, кроме того, в расписании каждого преподавателя обязательно должен существовать резерв времени, необходимый для перехода из одной аудитории в другую. Для этой задачи предположим, что каждый преподаватель способен вести обучение по каждому курсу и что у преподавателей нет часов консультаций и перерывов на ланчи и удовлетворение прочих естественных потребностей.⁷

Конкретно предположим, что существует n учебных курсов, предлагаемых в m различных локациях. Входные данные для нашей задачи состоят из следующих элементов:

- массива $C[1..n]$ учебных курсов, где каждый курс $C[i]$ содержит три поля: время начала $C[i].start$, время окончания $C[i].end$ и локацию $C[i].loc$;
- двумерного массива $T[1..m, 1..m]$, где $T[u, v]$ – время, требуемое для перехода из локации u в локацию v .

Необходимо найти минимальное число преподавателей, которые в совокупности могут вести обучение по каждому курсу, такое, что если преподавателю назначены два курса i и j , где $C[j].start \geq C[i].start$, то мы действительно имеем:

$$C[j].start \geq C[i].end + T[C[i].loc, C[j].loc].$$

Мы можем решить эту задачу, приведя ее к задаче поиска покрытия непересекающихся путей, как показано ниже. Мы создаем НАГ $G = (V, E)$, вершинами которого являются учебные курсы, а ребра представляют пары курсов, которые разнесены по времени достаточно далеко друг от друга, чтобы их мог читать один преподаватель. В частности, направленное ребро $i \rightarrow j$ означает, что один преподаватель может читать курс i , а затем курс j . Такой НАГ легко сформировать прямым перебором за время $O(n^2)$. Затем мы выполняем поиск покрытия непересекающихся путей в G , используя алгоритм поиска паросочетаний, описанный выше, – каждый направленный путь в G представляет допустимое расписание учебных курсов для одного преподавателя. В общей сложности алгоритм выполняется за время $O(n^2 + VE) = O(n^3)$ ⁸.

⁷ Но при этом предполагается, что преподаватели могут отвечать на сообщения электронной почты от студентов при переходах между аудиториями.

⁸ Если предположить, что все интервалы времени $T[u, v]$ равны⁹, эту задачу составления расписания в действительности можно решить за время $O(n \log n)$, используя простой жадный алгоритм.

⁹ Во многих американских университетах запланированы десятиминутные перерывы между учебными курсами, исходя из полной уверенности в том, что человек способен перейти из одной аудитории в другую в пределах одного кампуса за десять минут. Я предлагаю любому, кто действительно в этом уверен, посетить мой кампус и пройти из одного здания Зибель-центра (Siebel Center) в другое.

Несмотря на то что в начальном описании использовались понятия интервалов времени и расстояния, в действительности это задача поиска максимальных паросочетаний (а на самом деле это реальная задача поиска максимального потока). В частности, мы должны найти паросочетания возможно большего числа учебных курсов со следующим курсом, которые может читать один преподаватель. Необходимое число преподавателей равно числу учебных курсов, у которых нет последующего элемента, т. е. каждый курс без последующего элемента – это последний курс, который читает тот же преподаватель.

11.6. Алгоритм исключения для бейсбола

Каждый год американские фанаты бейсбола внимательно следят за своей любимой командой в надежде, что она завоюет место в плей-офф и в итоге выиграет чемпионат США (World Series). К несчастью, большинство команд «математически исключается» за несколько дней или даже недель до окончания регулярного сезона. Зачастую легко определить, когда команда исключается, – она не может выиграть достаточное число игр, чтобы догнать текущего лидера в своем дивизионе. Но иногда ситуация более запутанная. Например, в табл. 11.1 показано реальное положение команд Восточного дивизиона Американской лиги (American League East) на 30 августа 1996 г.

Таблица 11.1. Положение команд Восточного дивизиона Американской лиги (American League East) на 30 августа 1996 г.

Команда	Выигранные – проигранные игры	Осталось игр	NYN	BAL	BOS	TOR	DET
Янкиз (New York Yankees)	75 – 59	28		3	8	7	3
Иволги (Baltimore Orioles)	71 – 63	28	3		2	7	4
Красные носки (Boston Red Sox)	69 – 66	27	8	2		0	0
Синие сойки (Toronto Blue Jays)	63 – 72	27	7	7	0		0
Тигры (Detroit Tigers)	49 – 86	27	3	4	0	0	

Детройт явно отстает, но некоторые самые преданные фанаты «Тигров», возможно, не теряют надежды на то, что их команда все еще может победить. В конце концов, если Детройт выиграет все 27 оставшихся игр, то закончит регулярный сезон с 76 победами – это больше, чем сейчас имеет любая другая команда. Итак, пока все остальные команды проигрывают каждую игру... но это же невозможно, потому что остальные команды должны будут сыграть и между собой. Ниже приведено полное объяснение¹⁰:

¹⁰ Пример и это объяснение взяты с веб-сайта Эли Олиника (Eli Olinick) <https://s2.smu.edu/~olinick/riot/detroit.html>. Объяснение основано на совместном исследовании Олиника с Иланом Адлером (Ilan Adler), Аланом Эрерой (Alan Ereira) и Дорит Хохбаум (Dorit Hochbaum).

Выиграв все оставшиеся игры, Детройт может закончить регулярный сезон с результатом 76 – 86. Если «Янкиз» выигрывают всего лишь еще две игры, то закончат сезон с результатом 77 – 85, что позволит им опередить Детройт. Тогда предположим, что «Тигры» пройдут непообежденными оставшуюся часть сезона, а «Янкиз» не смогут выиграть любую другую игру.

В этом сценарии проблема заключается в том, что у Нью-Йорка все еще остается восемь игр с Бостоном. Если «Красные носки» выигрывают все эти игры, то закончат сезон как минимум с 77 победами, опережая «Тигров». Таким образом, единственный вариант для Детройта хотя бы получить шанс финишировать на первом месте – если Нью-Йорк выигрывает ровно одну из восьми игр с Бостоном и проигрывает все остальные игры. Но при этом «Носки» обязательно должны проиграть все игры против всех команд, за исключением Нью-Йорка. При этом первое место разделяют три команды...

Теперь посмотрим, что происходит с «Иволгами» и «Синими сойками» в нашем сценарии. У Балтимора осталось две игры с Бостоном и три с Нью-Йорком. И если все происходит так, как описано выше, то «Иволги» финишируют как минимум с 76 победами. Поэтому Детройт может догнать Балтимор, только если «Иволги» проигрывают все свои игры командам за исключением Нью-Йорка и Бостона. В частности, это означает, что Балтимор обязательно должен проиграть все семь оставшихся игр против Торонто. У «Синих соек» также осталось семь игр с «Янкиз», а мы уже видели, что Детройту для финиша на первом месте необходимо, чтобы Торонто непременно выиграл все эти игры. Но если это происходит, то «Синие сойки» выигрывают как минимум еще 14 игр, что приводит их к итоговому результату 77 – 85 или даже лучше, а это означает, что они опережают «Тигров». Итак, независимо от того, что происходит в этот момент продолжающегося сезона, Детройт не сможет выйти на первое место в Восточном дивизионе Американской лиги.

Должен существовать более эффективный способ описания этой ситуации.

Ниже приведена более абстрактная формулировка этой задачи. Наши входные данные состоят из двух массивов $W[1..n]$ и $G[1..n, 1..n]$, где $W[i]$ – число игр, уже выигранных командой i , а $G[i, j]$ – число будущих игр между командами i и j . Мы должны определить, сможет ли команда n завершить сезон с максимальным числом побед (возможно, равным числу побед других команд)¹¹.

¹¹ Мы неявно предполагаем, что ни одна из игр не завершается ничьей и что каждая игра действительно будет сыграна. Оба предположения соответствуют правилам Главной лиги бейсбола (Major League Baseball), по крайней мере, для игр, которые влияют на положение команд по итогам сезона при условии, что не произойдет войн, природных катастроф или нашествия пчелиных роев.

В середине 1960-х гг. Бенджамин Шварц (Benjamin Schwartz) обнаружил, что этот вопрос можно смоделировать как задачу поиска максимального потока, около 20 лет спустя Дэн Гасфилд (Dan Gusfield), Чарльз Мартел (Charles Martel) и Давид Фернандес-Бака (David Fernández-Baca) упростили формулировку Шварца задачи о потоке до задачи выбора пар. В частности, нам необходимо узнать, возможно ли выбрать победителя каждой игры так, чтобы команда n вышла на первое место. Пусть $R[i] = \sum_j G[i, j]$ обозначает число оставшихся игр для команды i . Предположим, что команда n выиграет все $R[n]$ оставшихся игр. Тогда команда n может выйти на первое место, если и только если каждая другая команда i выиграет не более $W[n] + R[n] - W[i]$ своих $R[i]$ оставшихся игр.

Поскольку мы должны выбрать команду, победившую в каждой игре, начнем с создания двудольного графа, узлы которого представляют игры и команды. Мы получаем $\binom{n}{2}$ узлов игр $g_{i,j}$, по одному для каждой пары $1 \leq i < j < n$, и $n - 1$ узлов команд t_i , по одному для каждого $1 \leq i < n$. Для каждой пары i, j мы добавляем ребра $g_{i,j} \rightarrow t_i$ и $g_{i,j} \rightarrow t_j$ с бесконечной пропускной способностью. Мы также добавляем вершину-источник s и ребра $s \rightarrow g_{i,j}$ с пропускной способностью $G[i, j]$ для каждой пары i, j . Наконец, добавляем целевой узел t и ребра $t_i \rightarrow t$ с пропускной способностью $W[n] - W[i] + R[n]$ для каждой команды i .

На рис. 11.7 показан граф, выведенный из описания положения команд Восточного дивизиона Американской лиги, где «команда n » – это Detroit Tigers. Все ребра без меток имеют бесконечную пропускную способность.

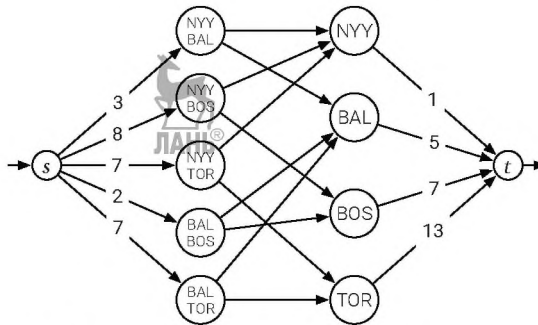


Рис. 11.7. Новички победили! Новички победили!

Теорема. Команда n может завершить сезон на первом месте, если и только если существует выполнимый путь в этом графе, который насыщает каждое ребро, исходящее из s .

Доказательство. Предположим, что команда n может завершить сезон на первом месте. Тогда каждая команда $i < n$ побеждает не более чем в $W[n] + R[n] - W[i]$ оставшихся играх. Для каждой игры между командами i и j , которую выигрывает команда i , добавляем один элемент потока вдоль пути $s \rightarrow g_{i,j} \rightarrow t_i \rightarrow t$. Поскольку существует ровно $G[i, j]$ игр между командами i и j , каждое ребро, исходящее из s , является насыщенным. Так как каждая

команда i выигрывает не более $W[n] + R[n] - W[i]$ игр, итоговый поток является выполнимым.

Обратно пусть f – выполнимый поток, который насыщает каждое ребро, исходящее из s . Предположим, что команда i выигрывает ровно $f(g_{i,j} \rightarrow t_j)$ игр против команды j для всех i и j . Тогда команды i и j играют $f(g_{i,j} \rightarrow t_j) + f(g_{j,i} \rightarrow t_i) = f(s \rightarrow g_{i,j}) = G[i, j]$ игр, поэтому каждая последующая игра будет сыграна. Кроме того, каждая команда i выигрывает в сумме $\sum_j f(g_{i,j} \rightarrow t_j) = f(t_i \rightarrow t) \leq W[1] + R[n] - W[i]$ будущих игр, следовательно, всего не более $W[n] + R[n]$ игр. Таким образом, если команда n выигрывает все свои оставшиеся игры, то она заканчивает сезон на первом месте. $\square$

Подводя итоги, отметим, что, принимая решение о том, сможет ли наша любимая команда победить, мы создаем транспортную сеть, вычисляем максимальный поток и сообщаем, насыщает ли этот максимальные поток каждое ребро, исходящее из s . Например, на графе на рис. 11.7 общая пропускная способность ребер, исходящих из s , равна 27 (потому что осталось 27 игр). С другой стороны, общая пропускная способность ребер, входящих в t , равна всего лишь 26, из чего следует, что величина максимального потока не превышает 26. Отсюда делаем вывод, что Детройт математически исключается.¹²

Транспортная сеть содержит $O(n^2)$ вершин и $O(n^2)$ ребер и может быть создана за время $O(n^2)$. Используя алгоритм Орлина, мы можем вычислить максимальный поток за время $O(VE) = O(n^4)$.

Это не самый быстрый алгоритм для задачи исключения бейсбольной команды. В 2001 г. Кевин Уэйн (Kevin Wayne) доказал, что можно определить все команды, которые математически исключаются из плей-офф, за время $O(n^3)$, по существу, используя одно вычисление максимального потока.

11.7. Выбор проекта

В заключительном примере предположим, что нам задано множество n проектов, которые мы, возможно, могли бы выполнить. Некоторые проекты не могут начаться до тех пор, пока не будут завершены другие конкретные проекты. Проект и зависимости между ними описываются направленным ациклическим графом G , вершинами которого являются проекты, а каждое ребро $u \rightarrow v$ указывает, что проект u не может быть выполнен раньше проекта v . (Это точно такая же форма графа зависимостей, которую мы рассматривали в разделе 6.4.) Наконец, каждый проект v связан с выгодой (или прибылью) $\$(v)$, которую мы получим, если проект будет завершен. Некоторые проекты имеют отрицательные выгоды, которые мы интерпретируем как положительные издержки (или накладные расходы). Мы можем выбрать для окончательного выполнения любое подмножество проектов X , в которое включены все их зависимости, т. е. для каждого про-

¹² К счастью (или к несчастью), мы, наконец, добрались до результата. Возможно, команда будет исключена, даже если общая пропускная способность всех ребер, входящих в t , не меньше общей пропускной способности всех ребер, исходящих из s .

екта $x \in X$ каждый проект, от которого зависит x , также включен в X . Наша цель – найти допустимое подмножество проектов, суммарная выгода от которых будет максимально возможной. В частности, если все задания имеют отрицательную выгоду, то правильный ответ – ничего не делать.

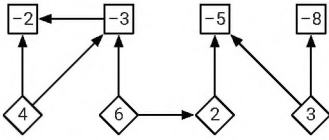


Рис. 11.8. Граф зависимостей для множества из восьми проектов. Ромбы обозначают выгодные проекты, квадраты – проекты с издержками. Каждое ребро $u \rightarrow v$ означает, что u зависит от v

На верхнем уровне наша задача – разделить проекты на два подмножества S и T – задания, которые мы *выбираем* (Select) и которые *отклоняем* (Turn down). Поэтому интуитивно мы должны были бы смоделировать эту задачу как поиск наименьшего разреза в конкретном графе. Но в каком графе? Как мы применим предварительные условия? Нам необходимо максимизировать выгоду, но мы знаем только лишь как найти наименьшие разрезы. И как преобразовать отрицательные выгоды в положительные пропускные способности?

Для преобразования заданного графа ограничений G в транспортную сеть G' мы добавляем вершину-источник s и целевую вершину t в граф зависимостей с ребром $s \rightarrow v$ для каждого выгодного задания v ($c(v) > 0$) и ребром $u \rightarrow t$ для каждого задания с издержками u ($c(u) < 0$). Интуитивно мы можем представить s как новое задание («Спать!») с выгодой/издержками 0, которое обязательно должны выполнить самым последним. Мы назначаем пропускные способности ребрам G' следующим образом:

- $c(s \rightarrow v) = c(v)$ для каждого выгодного задания v ;
- $c(u \rightarrow t) = -c(u)$ для каждого задания с издержками u ;
- $c(u \rightarrow v) = \infty$ для каждого ребра зависимости $u \rightarrow v$.

Все пропускные способности ребер положительны, так что это корректные входные данные для задачи поиска максимального разреза.

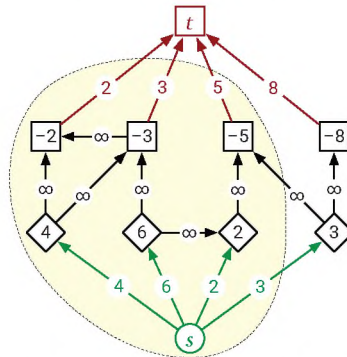


Рис. 11.9. Транспортная сеть для нашего примера графа зависимостей вместе с его наименьшим разрезом. Разрез имеет пропускную способность 13 и $P = 15$, поэтому общая выгода для выбранных заданий равна 2

Теперь рассмотрим произвольный (s, t) -разрез (S, T) в графе G' . Для любого ребра $u \rightarrow v$ в исходном графе зависимостей если $u \in S$ и $v \in T$, то $\|S, T\| = \infty$. Таким образом, мы можем легально выбирать задания в S , если и только если пропускная способность разреза (S, T) конечна.

В действительности оказывается, что разрезы с меньшей пропускной способностью соответствуют вариантам выбора заданий с более высокой выгодой. В частности, я утверждаю, что выбор заданий в S приносит общую выгоду $P - \|S, T\|$, где P – сумма всех положительных выгод:

$$P = \sum_v \max\{0, \$ (v)\} = \sum_{\$ (v) > 0} \$ (v).$$

Мы можем доказать это утверждение очевидным определением с выбором, как показано ниже. Для любого подмножества проектов X определяем три значения. (Как обычно, здесь мы определяем $c(u \rightarrow v) = 0$, если $u \rightarrow v$ не является ребром.)

$$\begin{aligned} \text{cost}(X) &:= \sum_{\substack{u \in X \\ \$ (u) < 0}} -\$ (u) = \sum_{u \in X} c(u \rightarrow t), \\ \text{yield}(X) &:= \sum_{\substack{v \in X \\ \$ (v) > 0}} \$ (v) = \sum_{v \in X} c(s \rightarrow v), \\ \text{profit}(X) &:= \sum_{v \in X} \$ (v) = \text{yield}(X) - \text{cost}(X). \end{aligned}$$

По определению $P = \text{yield}(V) = \text{yield}(S) + \text{yield}(T)$. Поскольку разрез (S, T) имеет конечную пропускную способность, только ребра формы $s \rightarrow v$ и $u \rightarrow t$ могут пересекать этот разрез. По построению каждое ребро $s \rightarrow v$ указывает на выгодное задание, а каждое ребро $u \rightarrow t$ указывает на задание с издержками. Следовательно, $\|S, T\| = \text{cost}(S) + \text{yield}(T)$. Из этого сразу же выводим, что $P - \|S, T\| = \text{yield}(S) - \text{cost}(S) = \text{profit}(S)$, как утверждалось выше.

Из этого сразу же следует, что мы можем максимизировать свою общую выгоду, вычислив наименьший разрез в графе G' . Можно легко создать G' из графа G за время $O(V+E)$ и вычислить наименьший (s, t) -разрез в G' за время $O(VE)$, используя алгоритм Орлина. Мы приходим к заключению о том, что весь алгоритм выбора проекта в целом выполняется за время $O(VE)$.

Упражнения

1. Пусть $G = (V, E)$ – направленный граф, в котором для каждой вершины v степени входа и выхода v равны. Предположим, что G содержит k реберно-непересекающихся путей из некоторой вершины u в другую вершину v . При таких условиях должен ли граф G также обязательно содержать k реберно-непересекающихся путей из v в u ? Привести доказательство или контрпример с объяснением.

2. Задан ненаправленный граф $G = (V, E)$ с тремя вершинами u, v и w . Описать и проанализировать алгоритм, определяющий, существует ли путь из u в w , проходящий через v . [Подсказка: если бы G был направленным графом, то эта задача стала бы NP-трудной.]
3. Рассмотрим направленный граф $G = (V, E)$ с несколькими вершинами-источниками $s_1, s_2, \dots, s_\sigma$ и целевыми вершинами $t_1, t_2, \dots, t_\tau$, где нет вершин, являющихся одновременно и источником, и целью. Многополюсный поток (multi-terminal flow) – это функция $f: E \rightarrow \mathbb{R}_{\geq 0}$, соблюдающая ограничение сохранения потока в каждой вершине, не являющейся ни источником, ни целью. Величина $|f|$ многополюсного потока равна общему избыточному потоку, исходящему из всех вершин-источников:

$$|f| := \sum_{i=1}^{\sigma} \left(\sum_w f(s_i \rightarrow w) - \sum_u f(u \rightarrow s_i) \right).$$

Как обычно, нас интересует поиск потоков с максимальной величиной с учетом ограничений пропускных способностей ребер. (В частности, нас не волнует, сколько потоков проходит из любого конкретного источника в любую конкретную цель.)

- (а) Рассмотреть приведенный ниже алгоритм вычисления многополюсных потоков. Переменные f и f' представляют функции потока. Подпрограмма $\text{MaxFlow}(G, s, t)$ решает стандартную задачу поиска максимального потока с источником s и целью t .

```

MaxMultiFlow( $G, s[1.. \sigma], t[1.. \tau]$ ):
   $f \leftarrow 0$                                      «Инициализация потока.»
  for  $i \leftarrow 1$  to  $\sigma$ 
    for  $j \leftarrow 1$  to  $\tau$ 
       $f' \leftarrow \text{MaxFlow}(G, s[i], t[j])$ 
       $f \leftarrow f + f'$                          «Обновление потока.»
  return  $f$ 

```

Доказать, что этот алгоритм корректно вычисляет многополюсный поток в графе G .

- (б) Описать более эффективный алгоритм вычисления максимального многополюсного потока в графе G .
4. На острове Содор (Sodor) находится большое число городов и деревень, соединенных крупной сетью железных дорог. Недавно поступило сообщение о нескольких случаях смертельно опасной инфекционной болезни (свиной грипп или превращение в зомби – из сообщения не ясно) в деревне Скарлой (Skarloey). Управляющий железными дорогами Содора планирует закрыть конкретные железнодорожные станции, чтобы предотвратить распространение болезни

в Тидмауте (Tidmouth), городе, в котором он живет. Никакие поезда не могут проходить через закрытую станцию. Для минимизации затрат (и внимания общественности) управляющий хочет закрыть как можно меньше станций. Но он не может закрыть станцию Скарлой, потому что это может создать опасность его заражения, и он не может закрыть станцию Тидмаут, потому что потеряет возможность посещать свой любимый паб.

Описать и проанализировать алгоритм поиска минимального числа станций, которые обязательно должны быть закрыты, чтобы заблокировать все железнодорожные маршруты из Скарлой в Тидмаут. Железнодорожная сеть Содора представлена ненаправленным графом с вершиной для каждой станции и ребром для каждого железнодорожного пути между двумя станциями. Две особые вершины s и t представляют станции в Скарлой и Тидмаут.

Например, если задан исходный граф, показанный на рис. 11.10, то алгоритм должен вернуть 2.

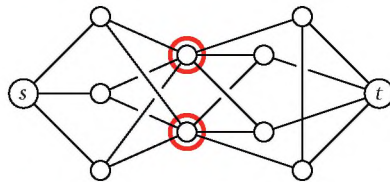


Рис. 11.10. Пример блокировки двух станций в железнодорожной сети Содора

5. Сетка $n \times n$ — это ненаправленный граф с n^2 вершинами, организованными в n строк и n столбцов. Мы обозначаем вершину в i -й строке и j -м столбце как (i, j) . Каждая вершина (i, j) имеет ровно четырех соседей $(i - 1, j)$, $(i + 1, j)$, $(i, j - 1)$ и $(i, j + 1)$, кроме граничных вершин, для которых $i = 1$, $i = n$, $j = 1$ или $j = n$.

Пусть (x_1, y_1) , (x_2, y_2) , ..., (x_m, y_m) — различные вершины, называемые *терминалами* (terminals) в сетке $n \times n$. *Задача спасения* (escape problem) состоит в том, чтобы определить, существует ли m вершинно-непересекающихся путей в сетке, соединяющих терминалы с любыми m различными граничными вершинами.

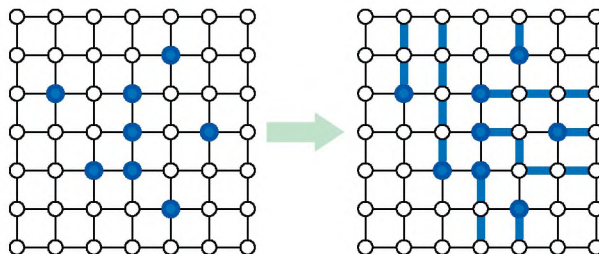


Рис. 11.11. Положительный вариант задачи спасения и ее решение

- (а) Описать и проанализировать эффективный алгоритм решения задачи спасения. Время выполнения алгоритма должно быть представлено небольшой полиномиальной функцией от n .
 - (б) Теперь предположим, что входные данные для задачи спасения состоят из одного целого числа n и списка из m терминальных вершин. Если m очень малое число, то предыдущее время выполнения в действительности является экспоненциальным относительно размера входных данных. Описать и проанализировать алгоритм решения задачи спасения за полиномиальное по m время.
 - ♥(с) Изменить предыдущий алгоритм так, чтобы он выводил явное описание путей спасения (если они существуют) с сохранением полиномиального по m времени выполнения.
6. Кафедра обеспечения тишины в пригородных районах (Commuter Silence Department) университета Шам-Пубанана устанавливает поле для мини-гольфа в цокольном этаже центра Си-Булл (See-Bull). Игровое поле представляет собой замкнутый многоугольник, ограниченный m горизонтальными и вертикальными отрезками прямой, соединенными под прямым углом. На поле имеется n начальных позиций и n лунок с установленным между ними соответствием один-к-одному. Всегда существует возможность пробить мяч вдоль прямой линии непосредственно из начальной позиции в соответствующую лунку без касания границ игрового поля. (Игрокам не разрешается сильно направлять мячи для гольфа за ограждения – слишком много стеклянных объектов вокруг.) Все n начальных позиций и n лунок находятся в различных местах.

К несчастью, компьютер архитектора сломался прямо перед началом процесса установки поля. Благодаря геркулесовым усилиям местных сисадминов удалось восстановить места начальных позиций и лунок, но вся информация о соответствии конкретной начальной позиции и конкретной лунки была потеряна безвозвратно.

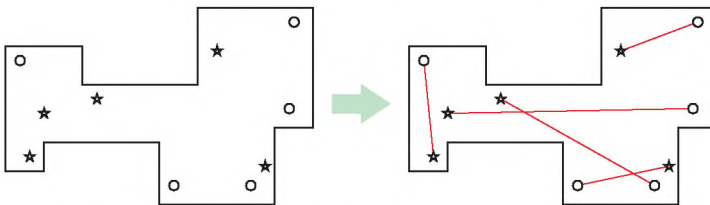


Рис. 11.12. Поле для мини-гольфа с пятью начальными позициями (★) и лунками (○) и допустимое соответствие между ними

Описать и проанализировать алгоритм вычисления соответствия один-к-одному между начальными позициями и лунками с соблюдением ограничения соединения по прямой линии или сообщения о том, что такого соответствия не существует. Входные данные состоят

из координат x и y для m углов игрового поля, n начальных позиций и n лунок. Предположим, что вы можете определить за постоянное время, пересекаются ли два отрезка, если заданы координаты x и y их конечных точек.

7. Покрытие циклами (cycle cover) заданного направленного графа $G = (V, E)$ – это множество вершинно-непересекающихся циклов, которое покрывает каждую вершину в G . Описать и проанализировать эффективный алгоритм поиска покрытия циклами для заданного графа или корректного сообщения о том, что такого покрытия циклами не существует. [Подсказка: используйте алгоритм поиска паросочетаний в двудольном графе.]
8. Предположим, что задана шашечная доска $n \times n$ с некоторыми удаленными клетками. У вас есть большой набор костяшек домино как раз такого размера, чтобы накрыть две клетки на доске. Описать и проанализировать алгоритм, определяющий, можно ли покрыть всю доску костяшками домино, – каждая костяшка непременно должна покрывать ровно две неудаленные клетки, и каждая неудаленная клетка обязательно должна быть покрыта ровно одной костяшкой домино.

Вашими входными данными является массив логических значений $Deleted[1..n, 1..n]$, где $Deleted[i, j] = \text{True}$, если и только если клетка по горизонтали i и вертикали j удалена. Вашим выводом является одно логическое значение, не требуется вычисление реального размещения костяшек домино. Например, для доски, показанной на рис. 11.13, алгоритм должен вернуть True .

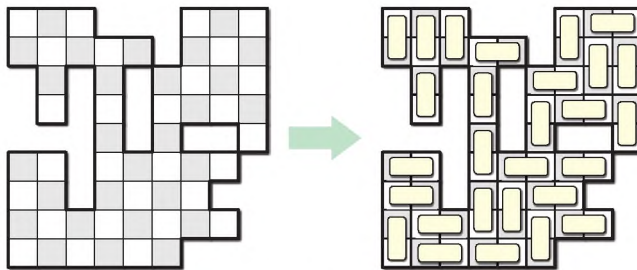


Рис. 11.13. Покрытие шашечной доски с удаленными клетками костяшками домино

9. Предположим, что задана квадратная сетка $n \times n$, в которой некоторые клетки окрашены в черный цвет, а остальные – белые. Описать и проанализировать алгоритм, определяющий, можно ли разместить на этой сетке фишки так, чтобы:
 - каждая фишка находилась на белой клетке;
 - каждый ряд сетки содержал ровно одну фишку;
 - каждый столбец сетки содержал ровно одну фишку.

Вашими входными данными является двумерный массив логических значений $IsWhite[1..n, 1..n]$, обозначающих белые клетки. Вашим выводом является одно логическое значение. Например, если задана сетка, показанная на рис. 11.14, то алгоритм должен вывести True.

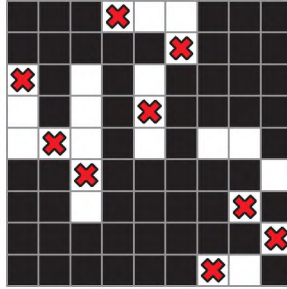


Рис. 11.14. Пометка каждого ряда и столбца в сетке

10. Предположим, что задано множество коробок, каждая из которых определяется высотой, шириной и глубиной в сантиметрах. Длины всех трех сторон каждой коробки находятся строго между 10 и 20 см. Как и следовало ожидать, одну коробку можно поместить в другую, если первую коробку можно повернуть так, чтобы ее высота, ширина и глубина были меньше соответственно высоты, ширины и глубины второй коробки. Коробки можно вкладывать друг в друга рекурсивно. Назовем коробку видимой, если она не находится внутри другой коробки.

Описать и проанализировать алгоритм вложения коробок друг в друга так, чтобы число видимых коробок было минимальным.

11. Предположим, что задана квадратная сетка $n \times n$, в которой некоторые клетки помечены. Сетка представлена массивом логических значений $M[1..n, 1..n]$, где $M[i, j] = \text{True}$, если и только если ячейка (i, j) помечена. Монотонный путь через сетку начинается в левой верхней ячейке, на каждом шаге перемещается только вправо или вниз и заканчивается в нижней правой ячейке. Наша цель – покрыть все помеченные ячейки как можно меньшим числом монотонных путей.

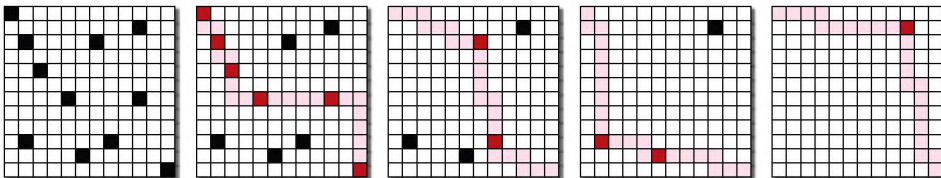


Рис. 11.15. Жадное покрытие помеченных ячеек в сетке четырьмя монотонными путями

- (а) Описать алгоритм поиска монотонного пути, покрывающего максимальное число помеченных ячеек.

- (b) Существует естественный жадный эвристический метод поиска минимального покрытия монотонными путями: если существуют какие-либо помеченные ячейки, найти монотонный путь π , покрывающий наибольшее число помеченных ячеек, снять метки с помеченных ячеек, покрываемых путем π , и применить рекурсию. Показать, что этот алгоритм не всегда вычисляет оптимальное решение.
- (c) Описать и проанализировать эффективный алгоритм вычисления наименьшего множества монотонных путей, покрывающих каждую помеченную ячейку.
12. Преподавательский совет университета Шам-Пубанана решил создать комиссию для определения, должны ли дядюшка Габби (Uncle Gabby), профессор Бобо Корнелиус (Professor Bobo Cornelius) или Мофо Психо-Горилла (Mofo the Psychic Gorilla) заменить недавно попавшего в опалу Барона Фактотума (Baron Factotum) в качестве нового официального маскота символа спортивных команд университета (The Fighting Pooh-bahs). Ровно один преподаватель должен быть выбран от каждой кафедры для участия в этой комиссии. Некоторые преподаватели имеют должности на нескольких кафедрах, но каждый член комиссии должен представлять только одну кафедру. Например, если профессор Благоевич имеет должность и на кафедре коррупции, и на кафедре глупости и выбран представителем кафедры глупости, то кафедру коррупции должен представлять кто-то другой. Наконец, политика университета требует, чтобы каждая комиссия из преподавателей непременно включала абсолютно равное число старших преподавателей, доцентов и профессоров. К счастью, число кафедр кратно трем.
- Описать и проанализировать алгоритм, определяющий подмножество преподавателей университета Шам-Пубанана для формирования комиссии пост-фактотумного выбора обезьянообразного маскота символа (The Post-Factotum Simian Mascot Symbol Committee) или корректно сообщающий о невозможности создания такой комиссии. Входные данные: двудольный граф, описывающий принадлежность преподавателей к кафедрам. Каждая вершина преподавателя помечена его должностью (старший преподаватель, доцент или профессор).
13. Кафедра обеспечения тишины в пригородных районах (Commuter Silence Department) университета Шам-Пубанана имеет гибкий учебный план с комплексным набором требований к окончанию курса. Кафедра предлагает n различных курсов, и существует m различных требований. Каждое требование определяет подмножество из n курсов и число обязательных для изучения курсов из этого подмножества. Подмножества для различных требований могут пе-

рекрываться, но каждый курс можно использовать для выполнения не более одного требования.

Например, предположим, что существует $n = 5$ курсов A, B, C, D, E и $m = 2$ требований к окончанию курса:

- вы должны изучить как минимум два курса из подмножества $\{A, B, C\}$;
- вы должны изучить как минимум два курса из подмножества $\{C, D, E\}$.

Тогда студент, изучивший курсы B, C, D , не может закончить курс обучения, но студент, изучивший курсы A, B, C, D или B, C, D, E , может закончить курс обучения.

Описать и проанализировать алгоритм, определяющий, может ли заданный студент закончить курс обучения. Входные данные для алгоритма: список m требований (каждое определяет подмножество n курсов и число курсов, обязательных для изучения, из этого подмножества) и список курсов, изученных студентом.

14. Вы организатор первой ежегодной 72-часовой танцевальной переключки на кафедре обеспечения тишины в пригородных районах (Commuter Silence) университета Шам-Пубанана непрерывно в пятницу, субботу и воскресенье. Несколько 30-минутных музыкальных фрагментов будут проигрываться во время этого мероприятия, а для исполнения приглашено большое число диджеев. Вы должны нанять диджеев в соответствии со следующими ограничениями:

- ежедневно должно проигрываться k музыкальных фрагментов, т. е. всего $3k$ фрагментов;
- каждый фрагмент непременно должен проигрываться одним диджеем в едином музыкальном жанре (эмбиент, бабблгам, дабстеп, хорроркор, К-поп, Квейто, марьячи, «прямолинейный» джаз, трип-хоп, нэшвиллское кантри, парапара, ска...);
- каждый жанр должен проигрываться не более одного раза в день;
- каждый кандидат в диджеи должен представить вам список жанров, которые он будет проигрывать;
- каждый диджей может проигрывать не более трех музыкальных фрагментов в течение всего мероприятия.

Предположим, что существует n кандидатов в диджеи и доступно g различных музыкальных жанров. Описать и проанализировать эффективный алгоритм, который распределяет диджеев и жанры для каждого из $3k$ музыкальных фрагментов или корректно сообщает о невозможности такого распределения.

15. Предположим, что вы поддерживаете веб-сайт, который ежедневно посещает одна и та же группа людей. Каждый посетитель заявляет

принадлежность к одной или нескольким демографическим группам; например, посетитель может описать себя как мужчину возраста 40–50 лет, отца семейства, проживающего в Иллинойсе, преподавателя университета, блогера и фаната Гилберта и Салливана.¹³ Ваш сайт поддерживается рекламодателями. Каждый рекламодатель сообщил вам, какие демографические группы должны видеть его рекламу и сколько рекламных объявлений вы должны демонстрировать каждый день. В общей сложности имеется n посетителей, k демографических групп и m рекламодателей.

Описать эффективный алгоритм, определяющий с учетом всех данных, описанных в предыдущем абзаце, можете ли вы показывать каждому посетителю ровно одно рекламное объявление в день так, чтобы каждый рекламодатель получил требуемое число показов своих рекламных объявлений и каждое рекламное объявление просмотрел представитель соответствующей демографической группы.

16. Предположим, что задан массив $A[1..m][1..n]$ неотрицательных действительных чисел. Мы должны выполнить *округление* A до целочисленной матрицы, заменяя каждый элемент x в A либо на $\lfloor x \rfloor$, либо на $\lceil x \rceil$ без изменения суммы элементов в каждой строке или столбце A . Например:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \mapsto \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

- (a) Описать и проанализировать эффективный алгоритм, который округляет A описанным выше способом или корректно сообщает, что такое округление невозможно.
- (b) Доказать, что корректное округление возможно, если и только если сумма элементов в каждой строке и в каждом столбце является целым числом. Другими словами, доказать, что алгоритм, полученный в части (a), возвращает корректное округление или что такое округление очевидно невозможно.
- ♥(c) Предположим, что мы гарантируем, что ни один из элементов исходной матрицы A не является целым числом. Описать и проанализировать еще более быстрый алгоритм, который округляет A или корректно сообщает, что такое округление невозможно. Для получения наивысшей оценки алгоритм должен выполняться за время $O(mn)$. [Подсказка: не используйте потоки.]

17. Децентрализованные динамические сети (ad-hoc networks) состоят из маломощных беспроводных устройств. В теории¹⁴ такие сети мо-

¹³ Я очень хороший теоретик в области информационных технологий, в частности, специалист по геометрическим алгоритмам.

¹⁴ Но не так уж часто на практике.

гут использоваться в районе боевых действий, в регионах, пострадавших от природных катастроф, и в других труднодостижимых районах. Идея состоит в том, что большой набор дешевых простых устройств можно распределить по целевой области (например, сбрасывая их с самолета). Затем эти устройства сами должны автоматически сконфигурироваться в функционирующую беспроводную сеть. Эти устройства могут обмениваться информацией только в ограниченном диапазоне. Мы предполагаем, что все устройства идентичны. Существует определенное расстояние D такое, что два устройства могут обмениваться информацией, если и только если расстояние между ними не превышает D .

Желательно, чтобы наша децентрализованная динамическая сеть была надежной, но поскольку устройства дешевые и маломощные, они часто выходят из строя. Если устройство определяет, что вскоре выйдет из строя, оно должно передать свою информацию на некоторое другое резервное устройство, находящееся в пределах достижимости по связи. Мы требуем, чтобы каждое устройство x имело k потенциальных резервных устройств, которые находятся на расстоянии, не превышающем D , от устройства x . Мы называем эти k устройств резервным комплектом (backup set) устройства x . Кроме того, мы не требуем, чтобы любое устройство находилось в резервном комплекте для слишком многих других устройств, иначе единственный сбой может повлиять на большую группу устройств в сети.

Итак, предположим, что задан предельный радиус обмена информацией D , параметры b и k , а также массив расстояний $d[1..n, 1..n]$, где $d[i, j]$ – расстояние между устройствами i и j . Описать алгоритм, который вычисляет резервный комплект размером k для каждого из n устройств такой, что ни одно из устройств не появляется в более чем b резервных комплектах или корректно сообщает, что правильный набор резервных комплектов не существует.

18. Из-за угрозы чрезвычайно жесткого сокращения бюджета потемкинский университет решил нанять актеров, которые должны изображать «студентов», присутствующих в аудиториях, создавая видимость полного набора на каждый предлагаемый учебный курс. Поскольку актеры стоят дорого, университет хотел бы нанять как можно меньше актеров.

На основе предыдущего опыта управлением ныне не существующим университетом Шам-Пубанана руководители потемкинского университета предоставили вам направленный ациклический граф $G = (V, E)$, вершины которого представляют учебные курсы, а каждое ребро $i \rightarrow j$ означает, что один и тот же «студент» может присутствовать на учебном курсе i , затем позже – на курсе j . Вам также предоставлен массив $cap[1..V]$, содержащий список максимального чис-

ла «студентов», которые могут присутствовать на каждом учебном курсе. Описать и проанализировать алгоритм, вычисляющий минимальное число «студентов», которое должно обеспечить видимость полного набора по каждому учебному курсу.

19. Квентин, Алиса и другие дети из Brakebills Physical Kids планируют экскурсию через Пустландию (Neitherlands) в Изобилию (Fillory). Пустландия – это огромный заброшенный город, состоящий из нескольких площадей, на каждой из которых есть один фонтан, способный магически переносить людей в другой мир. Соседние площади соединены воротами, которые были прокляты Зверем (Beast). Ворота между площадями открываются только на пять минут в течение каждого часа, все одновременно – с 12:00 до 12:05, потом с 1:00 до 1:05 и т. д., – а в остальное время заперты. Если за эти пять минут больше одного человека проходит через любые ворота, то Зверь узнает об их присутствии.¹⁵ Кроме того, любая попытка открыть запертые ворота или пройти через несколько ворот за один пятиминутный интервал превратит рискнувшего в ниффина.¹⁶ Но любое число людей может безопасно пройти через разные ворота одновременно и/или пройти через одни ворота, но в разное время.

У вас есть карта Пустландии, представляющая собой граф G с вершиной для каждого фонтана и ребром для каждых ворот. Фонтаны, переносящие на Землю и в Изобилию, обозначены явно.

- (a) Предположим, что также задано положительное целое число h . Описать и проанализировать алгоритм вычисления максимального количества людей, которые могут пройти от фонтана на Землю к фонтану на Изобилию не более чем за h часов, т. е. после того, как ворота открылись не более h раз, – без оповещения Зверя или превращения в ниффина. Время выполнения этого алгоритма должно зависеть от h . [Подсказка: создайте другой граф.]

♣♥(b) Описать и проанализировать алгоритм для решения части (a), время выполнения которого является полиномиальным от V и E , но не зависит от h .

- (c) С другой стороны, предположим, что также задано целое число k . Описать и проанализировать алгоритм вычисления максимального количества часов, позволяющего k людям пройти от фонтана на Землю к фонтану на Изобилию без оповещения Зверя или превращения в ниффина. [Подсказка: используйте часть (a).]

- ♥20. Пусть $G = (L \sqcup R, E)$ – двудольный граф, в котором левые вершины L проиндексированы $\ell_1, \ell_2, \dots, \ell_n$, а правые вершины – $r_1, r_2, \dots, r_n$. Па-

¹⁵ Это очень плохо.

¹⁶ Это хуже некуда.

росочетание M в G является *непересекающимся* (non-crossing), если для каждой пары ребер $\ell_i r_i$ и $\ell_j r_j$ в M мы имеем $i < i'$, если и только если $j < j'$.

- (a) Описать и проанализировать алгоритм поиска максимального непересекающегося паросочетания в G . [Подсказка: в действительности это не задача поиска потока.]
- (b) Описать и проанализировать алгоритм поиска наименьшего числа непересекающихся паросочетаний $M_1, M_2, \dots, M_k$ таких, что каждое ребро в G находится ровно в одном паросочетании M_i . [Подсказка: в действительности это задача поиска потока.]

♥21. Пусть $G = (L \sqcup R, E)$ – двудольный граф, в котором левые вершины L проиндексированы $\ell_1, \ell_2, \dots, \ell_n$ в некотором произвольном порядке.

- (a) Паросочетание M в графе G является *плотным* (dense), если не существует последовательных не связанных в пару вершин в подмножестве L , т. е. для каждого индекса i как минимум одна из вершин ℓ_i и ℓ_{i+1} инцидентна какому-либо ребру в M . Описать алгоритм, определяющий, является ли G плотным паросочетанием.
- (b) Паросочетание M в графе G является *разреженным* (sparse), если не существует последовательных связанных в пару вершин в подмножестве L , т. е. для каждого индекса i как минимум одна из вершин ℓ_i и ℓ_{i+1} не инцидентна какому-либо ребру в M . (В частности, пустое паросочетание является разреженным.) Описать алгоритм поиска максимального разреженного паросочетания в графе G .
- (c) Паросочетание M в графе G является *палиндромическим* (palindromic), если для каждого индекса i либо обе вершины ℓ_i и ℓ_{n-i+1} инцидентны какому-либо ребру в M , либо ни ℓ_i , ни ℓ_{n-i+1} не инцидентны какому-либо ребру в M . (В частности, пустое паросочетание является палиндромическим.) Описать алгоритм поиска максимального палиндромического паросочетания в графе G .

Ни в одной из описанных выше задач нет ограничения на то, какие вершины в подмножестве R являются связанными или не связанными в пару.

♥22. Дерево с корнем (rooted tree) – это направленный ациклический граф, в котором каждая вершина имеет ровно одно входящее ребро, за исключением корня, у которого нет входящих ребер. Важно: дерево с корнем состоит из вершины-корня, у которой есть ребра, указывающие на корни нуля или более деревьев с корнем меньшего размера. Описать эффективный алгоритм, вычисляющий для заданных двух деревьев с корнем A и B наибольшее дерево с корнем, изоморфное обоим подграфам A и B . Более кратко:

описать алгоритм поиска наибольшего общего поддерева этих двух деревьев с корнем.

[Подсказка: это была бы относительно простая задача динамического программирования, если бы каждый узел имел $O(1)$ потомков или потомки каждого узла были бы упорядочены слева направо. Но для неупорядоченных деревьев с большими степенями вам нужна другая методика для эффективного объединения рекурсивных подзадач.]

Глава 12

.....

NP-трудность

В своем кратком и фрагментарном трактате он дает вечный пример — не законов и даже не метода, поскольку нет иного метода, кроме как быть весьма разумным, но самого разума, который стремительно управляет анализом ощущений для решения вопросов принципа и определения.

— Т. С. Элиот (T. S. Eliot) об Аристотеле (Aristotle), «Совершенная критика» («The Perfect Critic»), «Священное дерево» («The Sacred Wood») (1921 г.)

Стандарты хороши тем, что у вас есть из чего выбирать; кроме того, если вам не понравится ни один из них, вы можете просто дождаться выпуска редакции следующего года.

— Эндрю С. Танненбаум (Andrew S. Tannenbaum), «Компьютерные сети» («Computer Networks») (1981 г.)

Это действительно редкий ум, который может сделать несуществующее до сих пор ослепляюще очевидным. Лозунг «Я мог бы подумать об этом» очень популярен и вводит в заблуждение, потому что никто этого не сделал, и это тоже очень важный и разоблачающий факт.

— Дирк Джентли (Dirk Gently) Ричарду МакДаффу (Richard McDuff) в книге Дугласа Адамса (Douglas Adams) «Детективное (холистическое) агентство Дирка Джентли» («Dirk Gently's Holistic Detective Agency») (1987 г.)

Если проблема не имеет решения, возможно, это не проблема, а факт, который нужно не решать, а справиться с ним через некоторое время.

— Шимон Перес (Shimon Peres), цитируемый Дональдом Рамсфелдом (Donald Rumsfeld) в книге «Rumsfeld's Rules: Leadership Lessons in Business, Politics, War, and Life» (2001 г.)

12.1. Игра, которую невозможно выиграть

Представьте себе, что некоторый торговец в красном костюме, подозрительно похожий на Тома Уэйтса (Tom Waits), предлагает вам черную стальную коробку с n двоичными переключателями на передней панели и лампочкой на крышке. Торговец сообщает вам, что состояние лампочки управляется слож-

ной логической схемой – набором логических вентилях AND, OR и NOT, соединенных проводами, с одним соединением для каждого переключателя и единственным проводом, присоединенным к лампочке. Затем он задает вам простой вопрос: возможно ли установить переключатели так, чтобы лампочка загорелась? Если вы сможете ответить на этот вопрос правильно, то торговец даст вам миллион 100 млрд долл. Если вы ответите неправильно или умрете раньше, чем ответите, то он заберет вашу душу.

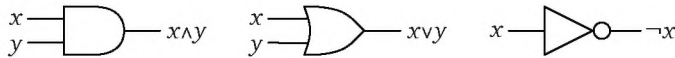


Рис. 12.1. Логические вентили AND, OR и NOT

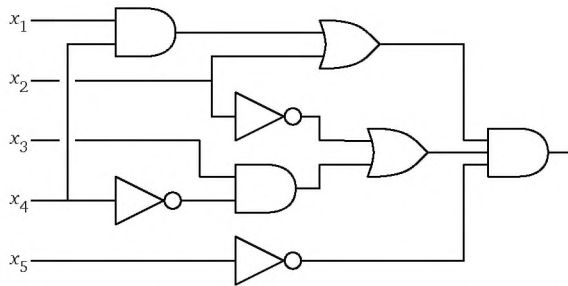


Рис. 12.2. Логическая схема. Входные сигналы поступают слева, выходной сигнал передается вправо

Насколько вы можете судить, враг рода человеческого вообще не соединил переключатели с лампочкой, поэтому не имеет значения, как вы установите переключатели, – лампочка никогда не загорится. Если вы заявите, что лампочку включить можно, то Люцифер откроет коробку и покажет, что схемы вообще нет. Но если скажете, что лампочку включить невозможно, прежде чем проверите все 2^n комбинаций, то Люцифер магически создаст внутри коробки схему, включающую лампочку, если и только если переключатели составляют одну из тех комбинаций, которую вы не проверили, затем переведет переключатели в нужное положение, и лампочка загорится. (Вы не способны уличить Люцифера в обмане, потому что не можете заглянуть в коробку до самого конца «эксперимента».) Единственный способ доказуемо ответить на вопрос Люцифера верно – проверить все 2^n возможных комбинаций. Вы сразу же понимаете, что это займет гораздо больше времени, чем вы рассчитываете прожить, поэтому вежливо отклоняете предложение Люцифера.

Люцифер улыбается и говорит хриплым голосом, как Джокер в исполнении Хита Леджера (Heath Ledger) после выкуривания пачки Мальборо: «Ах, да, конечно, у вас нет причины доверять мне. Но, возможно, я смогу успокоить вас». Он вручает вам большой свиток пергамента, – вы надеетесь, что он сделан из овечьей кожи, – с изображенной (или, возможно,

вытатуированной) на нем схемой логического контура. «Вот полная схема контура внутри коробки. Вы можете свободно покопаться в коробке, чтобы убедиться, что схема правильная. Или создать собственную схему по этим изображениям. Или написать компьютерную программу для имитации схемы. Все, что пожелаете. Если вы обнаружите, что изображения не соответствуют действительной схеме внутри коробки, то выиграете сто миллионов баксов». Несколько выборочных проверок убеждают вас, что в изображениях нет очевидных дефектов, и кажется, что скрытый обман невозможен.

Но вы должны отклонить и это «щедрое» предложение Люцифера. Задача, предлагаемая Люцифером, называется *задачей о выполнимости схемы* (circuit satisfiability), или CircuitSat: задана логическая схема, требуется определить, существует ли комбинация входных сигналов, при которых схема выводит True, или наоборот, всегда ли эта схема выводит False. Для любой конкретной комбинации входных сигналов мы можем вычислить вывод схемы за полиномиальное (в действительности линейное) время, используя метод поиска в глубину. Но никто не знает, как решить задачу CircuitSat быстрее, чем проверка всех 2^n возможных вариантов входных сигналов для такой схемы прямым перебором, для чего требуется экспоненциальное время. По общему признанию никто не смог действительно формально доказать, что мы не можем превзойти результат прямого перебора, – возможно, но только лишь возможно, что существует хитроумный алгоритм, который пока еще не открыт, – но ведь никто действительно формально не доказал и того, что не существуют антигравитирующих единорогов («сферические кони в вакууме» – более привычно для русскоговорящих читателей). Для всех практических целей осмотрительнее предполагать, что для задачи CircuitSat не существует быстрого алгоритма.

Вы отвечаете торговцу отказом. Он улыбается и говорит: «А ты умнее, чем кажешься, парень», – и улетает прочь на своем антигравитирующем единороге.

12.2. Р против NP

Минимальное требование для алгоритма, чтобы считать его «эффективным», – его время выполнения должно быть ограничено полиномиальной функцией от размера входных данных: $O(n^c)$ при некоторой константе c , где n – размер входных данных.¹ Исследователи достаточно быстро выяснили, что не все задачи можно решить так быстро, но потребовалось длительное время, чтобы точно определить, какие именно задачи можно решить быстро, а какие невозможно. Существует несколько так называемых *NP-трудных* (NP-hard) задач, которые, как уверено большинство людей, не-

¹ Эта форма записи эффективности была независимо формализована Аланом Кобхэмом (Alan Cobham) в 1965 г., Джеком Эдмондсом (Jack Edmonds) в 1965 г. и Майклом Рабином (Michael Rabin) в 1966 г., хотя похожие формы записи рассматривались более чем на десять лет раньше Куртом Гёделем (Kurt Gödel), Джоном Нэшем (John Nash) и Джоном фон Нейманом (John von Neumann).

возможно решить за полиномиальное время, даже при том, что никто не может доказать существование нижней суперполиномиальной границы.

Проблема разрешимости (decision problem) – это задача, выводом которой является единственное логическое значение: Yes (Да) или No (Нет). Определим три класса проблем разрешимости:

- P – множество проблем разрешимости, которые можно решить за полиномиальное время. Интуитивно P – это множество задач, которые можно решить быстро;
- NP – множество проблем разрешимости со следующим свойством: если ответ Yes, то существует доказательство этого факта, которое можно проверить за полиномиальное время. Интуитивно NP – это множество проблем разрешимости, для которых мы можем быстро подтвердить ответ Yes, если у нас имеется решение;
- $co-NP$ – по существу, противоположно NP . Если ответом на задачу из множества $co-NP$ является No, то существует доказательство этого факта, которое можно проверить за полиномиальное время.

Например, задача о выполнимости схемы принадлежит классу NP . Если заданная логическая схема выполнима, то любой набор из m входных значений, который на выходе дает результат True, является доказательством выполнимости этой схемы. Мы можем проверить это доказательство, выполняя схему за полиномиальное время. Широко распространено убеждение в том, что выполнимость схемы не входит ни в множество P , ни в множество $co-NP$, но в действительности этого никто точно не знает.

Каждая проблема разрешимости из множества P также входит в множество NP . Если задача находится в P , то мы можем подтвердить ответы Yes за полиномиальное время, пересчитывая ответ с нуля. Аналогично каждая задача из множества P также входит в множество $co-NP$.

Вероятно, единственный самый важный вопрос, остающийся без ответа в теоретической информатике, – если вообще не во всей информатике или даже во всей науке в целом, – действительно ли классы сложности P и NP различны. Интуитивно для большинства людей выглядит очевидным, что $P \neq NP$. Домашние задания и экзамены по курсам алгоритмов и структур данных убедили вас (как я надеюсь) в том, что задачи могут быть невероятно сложными для решения, даже если их решения просты в ретроспективе. Это абсолютно очевидно, разумеется, решение задач с нуля труднее, чем проверка корректности уже имеющегося решения. Мы можем обоснованно принять – и большинство разработчиков алгоритмов действительно принимают – утверждение « $P \neq NP$ » как закон природы, подобный другим законам природы, как, например, уравнения Максвелла, общую теорию относительности и восход солнца завтра утром, которые строго подтверждаются фактами, не имеют математического доказательства.

Но если бы мы были математически строгими, то вынуждены были бы признать, что никто не знает, как доказать, что $P \neq NP$. Действительно, в те-

чение многих десятилетий не наблюдается почти никакого продвижения к этому доказательству.² Математический институт Клэя (Clay Mathematics Institute) включил *P-versus-NP* в список как первую из семи Задач тысячелетия (Millenium Prize Problems), вознаграждение за решение которых составляет 1 млн долл. Да, действительно, некоторые люди потеряли свои души или, по крайней мере, лишились разума, пытаясь решить эту задачу.

Менее известный, но все же остающийся открытым вопрос – являются ли классы сложности NP и co-NP различными. Даже если мы можем проверить каждый ответ Yes быстро, нет основания поверить, что мы так же быстро сможем проверить ответы No. Например, насколько нам известно, не существует короткого доказательства того, что логическая схема не является выполнимой. В общем, это позволяет верить, что $NP \neq co-NP$, но опять же никто не знает, как это доказать.

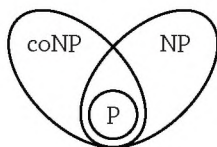


Рис. 12.3. Вот как выглядит мир в нашем представлении

12.3. NP-трудная, NP-легкая и NP-полная задача

Задача Π является *NP-трудной* (NP-hard), если алгоритм с полиномиальным временем выполнения для Π должен подразумевать алгоритм с полиномиальным временем выполнения для каждой задачи в NP. Другими словами:

Π – NP-трудная задача $\Leftrightarrow$ если Π можно решить за полиномиальное время, тогда $P = NP$.

Интуитивно, если мы могли бы решить одну конкретную NP-трудную задачу быстро, то мы могли бы быстро решить любую задачу, решение которой просто для понимания, используя решение этой одной конкретной задачи как подпрограмму. NP-трудные задачи как минимум так же трудны, как каждая задача в множестве NP.

Наконец, задача является *NP-полной* (NP-complete), если она и NP-трудная, и является элементом множества NP (или «NP-легкой» («NP-easy»)). Неформально NP-полные задачи являются самыми трудными задачами в NP. Алгоритм с полиномиальным временем выполнения даже для одной

² Вероятно, наиболее существенный прогресс в этом направлении принял форму барьерных (barrier) результатов, которые предполагают, что все направления атаки обречены на провал. В весьма реальном смысле у нас не только нет идей о том, как доказать $P \neq NP$, но в действительности мы даже не можем доказать, что у нас нет идей о том, как доказать $P \neq NP$.

NP-полной задачи должен сразу же подразумевать алгоритм с полиномиальным временем выполнения для каждой NP-полной задачи. Буквально тысячи задач продемонстрированы как NP-полные, поэтому алгоритм с полиномиальным временем выполнения для одной (а следовательно, и для всех) из них кажется весьма неправдоподобным.

Назвать задачу NP-трудной – все равно что сказать: «Если у меня есть собака, то она может свободно говорить по-английски». Вероятнее всего, вы не знаете, есть ли у меня собака или нет, но держу пари, что вы абсолютно уверены в том, что у меня нет говорящей собаки. Никто не может математически доказать, что собаки не способны говорить, – тот факт, что никто никогда не слышал, чтобы собака говорила, является доказательством, так как сотни исследований собак свидетельствуют об отсутствии у них требуемой формы рта и умственных способностей, но одно лишь подобное свидетельство не является математическим доказательством. Как бы то ни было, ни один человек в здравом уме не поверил бы мне, если бы я сказал, что я владелец собаки, свободно говорящей по-английски.⁵ Поэтому утверждение «Если у меня есть собака, то она может свободно говорить по-английски» имеет естественное следствие: никто в здравом уме не должен поверить в то, что у меня есть собака. Аналогично если задача является NP-трудной, то никто в здравом уме не должен верить, что она может быть решена за полиномиальное время.

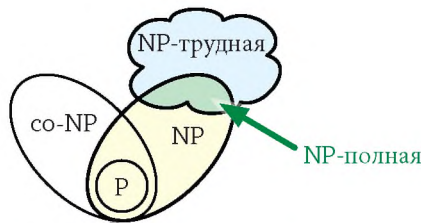


Рис. 12.4. Вот так более точно выглядит мир в нашем представлении

Не сразу очевидно, что любая задача является NP-трудной. Приведенная ниже замечательная теорема была впервые опубликована Стивеном Куком (Stephen Cook) в 1971 г. и независимо от него Леонидом Левиным в 1973 г.⁴

³ Завкафедрой печально качает головой и говорит: «Ой, да ладно вам, это только звучит как лай. Позвольте мне задать вопрос. Кто был самым великим теоретиком сложности задач всех времен?» Собака выразительно наклоняет голову в сторону, следует пауза в несколько секунд, затем говорит: «Карп!» После того, как завкафедрой выгоняет обоих из кабинета, собака поворачивается к хозяину и спрашивает: «Может быть, мне нужно было сказать Импальяццо?»

⁴ Левин сначала сообщил о своих результатах на семинарах в Москве в 1971 г., когда был аспирантом. Новости о результатах Кука не были известны в Советском Союзе как минимум до 1973 г., уже после того, как Левин объявил о своих результатах. В соответствии с законом Стиглера результат этой работы часто называют «теоремой Кука». Левин был лишен степени кандидата наук в Московском университете по политическим причинам, затем эмигрировал в США в 1978 г., а год спустя защитил кандидатскую диссертацию (PhD) в Массачусетском технологическом институте (MIT). Кук был уволен с кафедры математики университета Беркли в 1970 г., как раз за год до публикации своей основной работы. За это доказательство он (но не Левин) позже получил премию Тьюринга.

Теорема Кука–Левина. Задача о выполнимости схемы является NP-трудной.

Здесь я не буду описывать даже в общих чертах доказательство, потому что до сего момента я (умышленно) пользовался нечеткими определениями.⁵

♥ 12.4. Формальные определения (НС SVNT DRACONES – Тут [обитают] драконы)

Формально классы сложности P, NP и co-NP определяются в терминах языков и машин Тьюринга. Язык (language) – это множество строк, определенное на некотором конечном алфавите Σ , и без потери для обобщения мы можем предположить, что $\Sigma = \{0, 1\}$. Машина Тьюринга (Turing machine) – это весьма ограниченный тип компьютера, – грубо говоря, это конечный автомат с бесконечной запоминающей лентой, определение которого, как ни парадоксально, не важно. P – это множество языков, которое может быть решено за полиномиальное (Polynomial) время с помощью детерминированной одноленточной машины Тьюринга. Аналогично NP – это множество всех языков, которые могут быть решены за полиномиальное время с помощью недетерминированной машины Тьюринга. NP – это аббревиатура от Nondeterministic Polynomial-time.

Требование полиномиального времени выполнения достаточно призывное, поэтому не обязательно определять точную форму машины Тьюринга (число лент, головок, треков, размер алфавита ленты и т. д.). В действительности любой алгоритм, выполняемый на машине с прямым доступом (к памяти)⁶ за время $T(n)$, можно имитировать на машине Тьюринга с одной лентой, треком и головкой, которая выполняется за время $O(T(n)^4)$. Результат этой имитации позволяет нам формально подтвердить вычислительную сложность в терминах стандартных программных конструкций высокого уровня, таких как массивы, циклы и рекурсия, вместо описания всех подробностей непосредственно в терминах машин Тьюринга.

⁵ Интересующиеся читатели найдут доказательство в моих конспектах лекций по недетерминированным машинам Тьюринга здесь: <http://algorithms.wtf> или в превосходной книге Боаз Барак (Boaz Barak) «Introduction to Theoretical Computer Science».

⁶ Машины с прямым доступом к памяти (random-access machines) – это модель вычислений, которая более точно моделирует физические компьютеры. Стандартная машина с прямым доступом к памяти имеет неограниченную память с произвольным доступом, моделируемую неограниченным массивом $M[0..∞]$, где каждый адрес $M[i]$ содержит одно w -битовое целое число при некотором фиксированном целом значении w и может считывать или записывать данные в любом адресе памяти за постоянное время. Алгоритмы машин с прямым доступом – RAM-алгоритмы – формально записываются на языке, похожем на ассемблер, используя такие инструкции, как ADD i, j, k (означающую « $M[i] \leftarrow M[i] + M[k]$ »), INDIR i, j (означающую « $M[i] \leftarrow M[M[j]]$ ») и IF0GOTO i, ℓ (означающую «если $M[i] = 0$, перейти в строку ℓ »), но точный набор инструкций, как ни странно, не важен. По определению каждая инструкция выполняется за единицу времени. На практике RAM-алгоритмы могут быть верно описаны с использованием псевдокода более высокого уровня при условии, что мы позаботимся об арифметической точности.

Формально задача Π является NP-трудной, если и только если для каждого языка $\Pi' \in \text{NP}$ существует редукция Тьюринга (Turing reduction) от Π' до Π за полиномиальное время. Редукция Тьюринга означает редукцию (приведение), которая может быть выполнена на машине Тьюринга, т. е. машине Тьюринга M , которая может решить задачу Π' , используя другую машину Тьюринга M' для задачи Π как подпрограмму типа «черный ящик». Редукции Тьюринга часто называют редукциями оракула, а редукции Тьюринга за полиномиальное время также называют редукциями Кука (Cook reductions).

Исследователи в области теории сложности вычислений предпочитают определять NP-трудность в терминах многозначных редукций (many-one reductions) за полиномиальное время, которые также называются редукциями Карпа (Karp reductions). Многозначная редукция от одного языка $L' \subseteq \Sigma^*$ к другому языку $L \subseteq \Sigma^*$ есть функция $f: \Sigma^* \rightarrow \Sigma^*$ такая, что $x \in L'$, если и только если $f(x) \in L$. Затем мы можем определить язык L как NP-трудный, если и только если для любого языка $L' \in \text{NP}$ существует многозначная редукция от L' к L , которую можно вычислить за полиномиальное время.

Каждая редукция Карпа «есть» редукция Кука, но не наоборот. В частности, любая редукция Карпа от одного решения задачи Π к другому решению Π' равнозначна преобразованию входных данных в Π во входные данные в Π' , вызову оракула (т. е. подпрограммы) для Π' , затем возврату дословного ответа. Но, насколько нам известно, не каждую редукцию Кука можно имитировать редукцией Карпа.

Теоретики в области сложности вычислений в основном предпочитают редукции Карпа, потому что NP замкнуто относительно редукций Карпа, но не замкнуто относительно редукций Кука (если только не $\text{NP} = \text{co-NP}$, что считается нежелательным). Существуют естественные задачи, которые (1) NP-трудные с учетом редукций Кука, но (2) NP-трудные с учетом редукций Карпа, только если $P = \text{NP}$. Одним из тривиальных примеров такой задачи является UnSat: задана логическая формула, является ли она всегда ложной? С другой стороны, многозначные редукции применяются только к решениям задач (или более формально: к языкам); формально ни одна из задач оптимизации или задач на построение не является Карп-NP-трудной.

Чтобы сделать все еще более запутанным, Кук и Карп изначально определили NP-трудность в терминах редукций в логарифмическом пространстве (logarithmic-space reductions). Каждая редукция в логарифмическом пространстве является редукцией за полиномиальное время, но (насколько нам известно) не наоборот. Остается открытым вопрос: действительно ли ослабление набора допустимых редукций (Кука или Карпа) из логарифмического пространства к полиномиальному времени изменяет множество NP-трудных задач.

К счастью, на практике ни одна из этих тонкостей не дает о себе знать – в частности, каждая редукция, описанная в этой главе, может быть формализована как множественная редукция в логарифмическом пространстве, – так что пора прийти к себе.

12.5. Редукции и задача Sat

Для доказательства того, что любая задача, помимо задачи выполнимости схемы, является NP-трудной, мы используем редукционный аргумент (reduction argument). Редукция задачи A к другой задаче B означает описание алгоритма для решения задачи A с предположением, что алгоритм для решения задачи B уже существует. Вы уже использовали редукцию в течение многих лет, даже до чтения этой книги, только, вероятно, называли ее по-другому, например подпрограммы или вспомогательные функции, или модульное программирование, или применение калькулятора. Для доказательства NP-трудности чего-либо мы описываем аналогичное преобразование между задачами, но не в том направлении, в котором ожидает большинство людей.

Вы должны вытатуировать следующее правило на тыльной стороне руки, рядом с датой рождения вашей мамы и реальными правилами «Монополии»⁷.

Чтобы доказать, что задача A является NP-трудной, необходимо привести известную NP-трудную задачу к A .

Другими словами, чтобы доказать, что задача трудная, вы должны описать эффективный алгоритм решения другой задачи, о которой известно, что она трудная, используя гипотетический эффективный алгоритм для вашей задачи как подпрограмму типа «черный ящик». Определяющая логика – доказательство от противного. Редукция предполагает, что если бы ваша задача была легкой, то и другие задачи должны быть легкими, что неверно. Равнозначно поскольку вам известно, что другая задача является трудной, то редукция предполагает, что ваша задача тоже обязательно должна быть трудной; вашего гипотетического эффективного алгоритма в действительности не существует.

В качестве канонического примера рассмотрим задачу выполнимости формулы (formula satisfiability), которую обычно называют просто Sat. Входными данными для Sat является логическая формула, например:

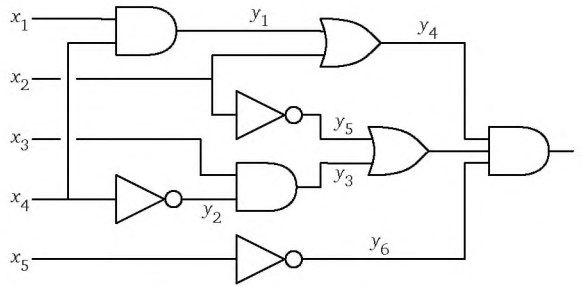
⁷ Если игрок попадает на клетку с доступным объектом собственности и отказывается (или не может) купить ее, то такой объект собственности немедленно продается на аукционе тому, кто предложил наивысшую цену. Игрок, который изначально отказался от покупки собственности, может участвовать в аукционе, а ставки могут быть произвольно выше или ниже объявленной цены. Игроки в тюрьме могут продолжать покупать и продавать собственность, покупать и продавать дома и отели, а также собирать арендную плату. В игре имеется 32 дома и 12 отелей, когда они исчезают, они исчезают. В частности, если все дома уже на игровом поле, вы не можете превратить отель в четыре дома, вы обязательно должны снести до основания все отели в группе. Игроки могут продавать или обменивать незастроенные участки собственности друг с другом, но не могут продать собственность обратно в банк. С другой стороны, игроки могут продавать здания банку (за полцены), но не могут продавать или обменивать здания друг с другом. Все штрафы выплачиваются непосредственно банку. Игрок, остановившийся на поле свободной парковки (Free Parking), ничего не выигрывает. Игрок, остановившийся на поле Go, получает ровно 200 долл., не больше. Железные дороги – это не магические телепорты. Наконец, Джефф всегда получает пинок. Нет, не тиранозавр (T-Rex) или пингвин – именно пинок, черт побери.

$$(a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee (\bar{a} \Rightarrow \bar{d}) \vee (c \neq a \wedge b)),$$

а вопрос такой: возможно ли присвоить логические значения переменным $a, b, c, \dots$ так, чтобы вычисление всей формулы дало результат True.

Для доказательства того, что Sat является NP-трудной задачей, мы должны выполнить редукцию от известной NP-трудной задачи. Единственной задачей, о которой мы до сих пор знали, что она NP-трудная, является CircuitSat, поэтому начнем с нее.

Пусть K – произвольная логическая схема. Мы можем преобразовать (или, более точно, переписать) K в логическую формулу Φ , как показано ниже. Сначала пометим каждый внутренний провод новой переменной y_i , а внешний провод – новой переменной z . Формула Φ состоит из списка равенств, по одному для каждого вентиля, разделенных операторами AND, за которыми следует завершающий элемент $\wedge z$. На рис. 12.5 показана итоговая запись для этого примера логической схемы.



$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \bar{x}_4) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge \\ (y_5 = \bar{x}_2) \wedge (y_6 = \bar{x}_5) \wedge (y_7 = y_3 \vee y_5) \wedge (z = y_4 \wedge y_7 \wedge y_6) \wedge z$$

Рис. 12.5. Запись логической схемы в виде логической формулы

Теперь мы утверждаем, что исходная схема K выполнима, если и только если полученная в итоге формула Φ выполнима. Как и любое другое утверждение «если и только если», мы доказываем это утверждение в два этапа.

- $\Rightarrow$ Если задано множество входных данных, выполняющих логическую схему K , то мы можем вывести выполнимое присваивание для формулы Φ , вычислив выход каждого логического вентиля в схеме K .
- $\Leftarrow$ Если задано выполнимое присваивание для формулы Φ , то мы можем получить выполнимые входные данные для логической схемы, просто игнорируя переменные внутренних проводов y_i и выходную переменную z .

Все преобразование в целом от логической схемы к логической формуле можно выполнить за линейное время. Кроме того, размер полученной формулы больше любого разумного представления логической схемы всего лишь с постоянным коэффициентом.

Теперь предположим, чисто теоретически, что существует алгоритм, который может определить за полиномиальное время, является ли заданная логическая формула выполнимой. Тогда при любой заданной логической схеме K мы можем решить, является ли K выполнимой схемой, сначала преобразовав K в логическую формулу Φ , как показано выше, затем спросив у нашего волшебного таинственного алгоритма Sat, является ли формула Φ выполнимой, как предлагается на рис. 12.6. Каждый прямоугольник представляет алгоритм. Красный квадрат слева – подпрограмма преобразования. Квадрат справа – гипотетический волшебный алгоритм Sat. Он непременно должен быть волшебным, потому что на нем изображена радуга⁸.

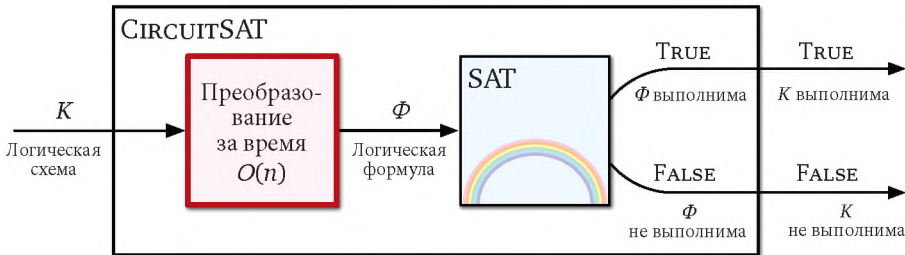


Рис. 12.6. Преобразование для определения выполнимости логической схемы

Если вместо волшебных прямоугольников-коробок вы предпочитаете волшебный псевдокод:

```

CircuitSat(K):
  transcribe K into a boolean formula Φ
  return Sat(Φ)    <<★★★★MAGIC★★★★>>
  return f
  
```

Преобразование K в Φ требует всего лишь полиномиального времени (в действительности только линейного времени, но особой разницы нет), поэтому весь алгоритм CircuitSat также выполняется за полиномиальное время.

$$T_{\text{CircuitSat}}(n) \leq O(n) + T_{\text{Sat}}(O(n)).$$

Мы приходим к выводу, что любой алгоритм с полиномиальным временем выполнения для Sat должен давать нам алгоритм с полиномиальным временем выполнения для CircuitSat, что, в свою очередь, предполагает $P = NP$. Поэтому Sat является NP-трудной задачей.

12.6. 3Sat (от CircuitSat)

Особый случай задачи Sat, который весьма полезен при доказательстве результатов NP-трудности, называется **3CNF-Sat** или чаще просто **3Sat**.

⁸ Кай Эриксон (Kay Erickson), при личном общении в 2011 г. Для тех, кто читает черно-белый печатный экземпляр книги: да, эта полукруглость изображает радугу.

Логическая формула находится в конъюнктивной нормальной форме (КНФ) (conjunctive normal form – CNF), если представляет собой конъюнкцию (AND) нескольких компонент (clauses), каждая из которых является дизъюнкцией (OR) нескольких литералов (literals), каждый из которых является переменной или ее отрицанием. Например:

$$\overbrace{(a \vee b \vee c \vee d)}^{\text{компонента}} \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b}).$$

Формула 3CNF – это формула CNF с ровно тремя литералами в каждой компоненте, поэтому приведенный выше пример не является формулой 3CNF, так как в первой компоненте содержится четыре литерала, а в последней – только два. 3Sat – это ограничение Sat по использованию только формул 3CNF: если задана формула 3CNF, существует ли присваивание ее переменным, которое при вычислении дает результат True?

Мы могли бы доказать, что 3Sat является NP-трудной задачей с помощью редукции от более общей задачи Sat, но в действительности проще начать с нуля, т. е. с редукции прямо от задачи CircuitSat.

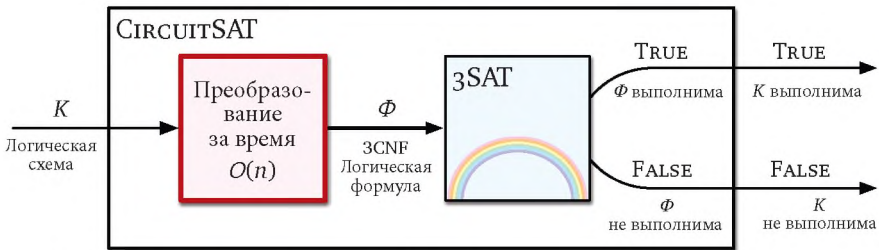


Рис. 12.7. Редукция от CircuitSat к 3Sat за полиномиальное время

Если задана произвольная логическая схема K , то мы выполняем преобразование K в равнозначную формулу 3CNF в несколько этапов. Эта редукция, за исключением самого последнего этапа, в действительности была описана Григорием Цейтиным в 1966 г., за пять лет до того, как Кук и Левин сообщили о своих доказательствах теоремы Кука–Левина. (В той же работе 1966 г. Цейтин описал задачу, которую мы теперь называем CNF-Sat, возможно, впервые.) Будем описывать каждый этап, одновременно доказывая, что он корректен.

- Убедимся, что каждый вентиль AND и OR в схеме K имеет ровно два входа. Если какой-либо вентиль имеет $k > 2$ входов, то заменим его на двоичное дерево из $k - 1$ двоичных вентилях. Назовем полученную схему K' . Схемы K и K' логически равнозначны, поэтому любой выполняемый ввод для K является выполняемым вводом для K' и наоборот.
- Перепишем схему K' в виде логической формулы Φ_1 с одной компонентой для каждого вентиля точно так же, как в предыдущей редукции к Sat. Мы уже доказали, что каждый выполнимый ввод для схемы

K' можно преобразовать в выполнимое присваивание для переменных в формуле Φ_1 , и наоборот.

- Заменим каждую компоненту в Φ_1 на формулу CNF. Существует только три типа компонент в Φ_1 – по одной для каждого типа вентилей в схеме K' :

$$\begin{aligned} a = b \wedge c &\mapsto (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c) \\ a = b \vee c &\mapsto (\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c}) \\ a = \bar{b} &\mapsto (a \vee b) \wedge (\bar{a} \vee \bar{b}). \end{aligned}$$

Назовем полученную формулу CNF Φ_2 . Поскольку Φ_1 и Φ_2 логически равнозначны, каждое выполнимое присваивание для Φ_1 также является выполнимым присваиванием для Φ_2 , и наоборот.

- Заменим каждую компоненту в Φ_2 на формулу 3CNF. Каждая компонента в Φ_2 содержит не более трех литералов, но нам нужны компоненты с ровно тремя литералами. Для получения формулы 3CNF мы разворачиваем каждую компоненту из двух литералов в Φ_2 в две компоненты из трех литералов, вводя одну новую переменную, а последнюю компоненту из одного литерала в Φ_2 – в четыре компоненты из трех литералов, вводя две новые переменные.

$$\begin{aligned} a \vee b &\mapsto (a \vee b \vee x) \wedge (a \vee b \vee \bar{x}) \\ z &\mapsto (z \vee x \vee y) \wedge (z \vee \bar{x} \vee y) \wedge (z \vee x \vee \bar{y}) \wedge (z \vee \bar{x} \vee \bar{y}). \end{aligned}$$

Назовем полученную формулу 3CNF Φ_3 . Каждое выполнимое присваивание для Φ_2 можно преобразовать в выполнимое присваивание для Φ_3 , присваивая произвольные значения новым переменным (x и y). И обратно: каждое выполнимое присваивание для Φ_3 можно преобразовать в выполнимое присваивание для Φ_2 , игнорируя новые переменные.

Рассматриваемый здесь пример логической схемы преобразуется в приведенную ниже формулу 3CNF; ср. с рис. 12.5.

$$\begin{aligned} &(y_1 \vee \bar{x}_1 \vee \bar{x}_4) \wedge (\bar{y}_1 \vee x_1 \vee z_1) \wedge (\bar{y}_1 \vee x_1 \vee \bar{z}_1) \wedge (\bar{y}_1 \vee x_4 \vee z_2) \wedge (\bar{y}_1 \vee x_4 \vee \bar{z}_2) \\ &\wedge (y_2 \vee x_4 \vee z_3) \wedge (y_2 \vee x_4 \vee \bar{z}_3) \wedge (\bar{y}_2 \vee \bar{x}_4 \vee z_4) \wedge (\bar{y}_2 \vee \bar{x}_4 \vee \bar{z}_4) \\ &\wedge (y_3 \vee \bar{x}_3 \vee \bar{y}_2) \wedge (\bar{y}_3 \vee x_3 \vee z_5) \wedge (\bar{y}_3 \vee x_3 \vee \bar{z}_5) \wedge (\bar{y}_3 \vee y_2 \vee z_6) \wedge (\bar{y}_3 \vee y_2 \vee \bar{z}_6) \\ &\wedge (\bar{y}_4 \vee y_1 \vee x_2) \wedge (y_4 \vee \bar{x}_2 \vee \bar{z}_7) \wedge (y_4 \vee \bar{x}_2 \vee \bar{z}_7) \wedge (y_4 \vee \bar{y}_1 \vee z_8) \wedge (y_4 \vee \bar{y}_1 \vee \bar{z}_8) \\ &\wedge (y_5 \vee x_2 \vee z_9) \wedge (y_5 \vee x_2 \vee \bar{z}_9) \wedge (\bar{y}_5 \vee \bar{x}_2 \vee z_{10}) \wedge (\bar{y}_5 \vee \bar{x}_2 \vee \bar{z}_{10}) \\ &\wedge (y_6 \vee x_5 \vee z_{11}) \wedge (y_6 \vee x_5 \vee \bar{z}_{11}) \wedge (\bar{y}_6 \vee \bar{x}_5 \vee z_{12}) \wedge (\bar{y}_6 \vee \bar{x}_5 \vee \bar{z}_{12}) \\ &\wedge (\bar{y}_7 \vee y_3 \vee y_5) \wedge (y_7 \vee \bar{y}_3 \vee z_{13}) \wedge (y_7 \vee \bar{y}_3 \vee \bar{z}_{13}) \wedge (y_7 \vee \bar{y}_5 \vee z_{14}) \wedge (y_7 \vee \bar{y}_5 \vee \bar{z}_{14}) \\ &\wedge (y_8 \vee \bar{y}_4 \vee \bar{y}_7) \wedge (\bar{y}_8 \vee y_4 \vee z_{15}) \wedge (\bar{y}_8 \vee y_4 \vee \bar{z}_{15}) \wedge (\bar{y}_8 \vee y_7 \vee z_{16}) \wedge (\bar{y}_8 \vee y_7 \vee \bar{z}_{16}) \\ &\wedge (y_9 \vee \bar{y}_8 \vee \bar{y}_6) \wedge (\bar{y}_9 \vee y_8 \vee z_{17}) \wedge (\bar{y}_9 \vee y_6 \vee z_{18}) \wedge (\bar{y}_9 \vee y_6 \vee \bar{z}_{18}) \wedge (\bar{y}_9 \vee y_8 \vee \bar{z}_{17}) \\ &\wedge (y_9 \vee z_{19} \vee z_{20}) \wedge (y_9 \vee \bar{z}_{19} \vee z_{20}) \wedge (y_9 \vee z_{19} \vee \bar{z}_{20}) \wedge (y_9 \vee \bar{z}_{19} \vee \bar{z}_{20}). \end{aligned}$$

Ничего себе! На первый взгляд эта формула может выглядеть гораздо более сложной, чем исходная схема, но в действительности ее размер больше только лишь на постоянный коэффициент. В частности, упрощенная схема K' содержит не более чем в два раза больше проводов по сравнению с исходной схемой K , а каждый двоичный вентиль в схеме K' преобразуется не более чем в пять компонент в Φ_3 . Даже если бы размер формулы был большой полиномиальной функцией (например, n^{374}) от размера схемы, мы все равно получили бы приемлемую редукцию.

Наша редукция выполняет преобразование произвольной логической схемы K в формулу 3CNF Φ_3 за полиномиальное время (в действительности за линейное время). Кроме того, любой выполняемый ввод для исходной схемы K можно преобразовать в выполняемое присваивание для Φ_3 , а любое выполняемое присваивание для Φ_3 можно преобразовать в выполняемый ввод для схемы K . Другими словами, схема K является выполняемой, если и только если формула Φ_3 является выполняемой. Таким образом, если задачу 3Sat можно решить за полиномиальное время, то задачу CircuitSat также можно решить за полиномиальное время, откуда следует, что $P = NP$. Мы приходим к выводу, что задача 3Sat является NP-трудной.

12.7. Максимальное независимое множество (от 3Sat)

Для нескольких рассматриваемых ниже задач входными данными является простой невзвешенный ненаправленный граф, а в задаче требуется найти размер наибольшего или наименьшего подграфа с некоторым структурным свойством.

Пусть G – произвольный граф. Независимое множество (independent set) в G – это подмножество вершин G , между которыми нет ребер. Задача поиска максимального независимого множества (maximum independent set), которую я буду сокращенно называть MaxIndSet, требует найти размер наибольшего независимого множества в заданном графе. Я докажу, что MaxIndSet является NP-трудной задачей, используя редукцию от 3Sat, как предлагается на рис. 12.8.

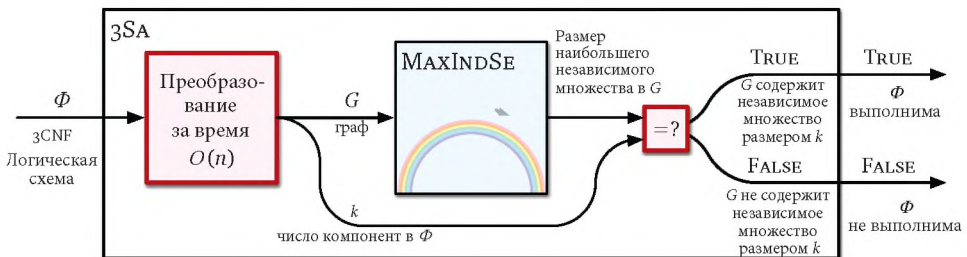


Рис. 12.8. Редукция за полиномиальное время от 3Sat к задаче MaxIndSet

Если задана произвольная формула 3CNF Φ , то мы формируем граф G , как показано ниже. Пусть k обозначает число компонент в формуле Φ . Граф G содержит ровно $3k$ вершин, по одной для каждого литерала в Φ . Две вершины в G соединены ребром, если и только если (1) они соответствуют литералам в одной компоненте или (2) они соответствуют переменной и ее инверсии. Например, формула $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ преобразуется в граф, показанный на рис. 12.9.

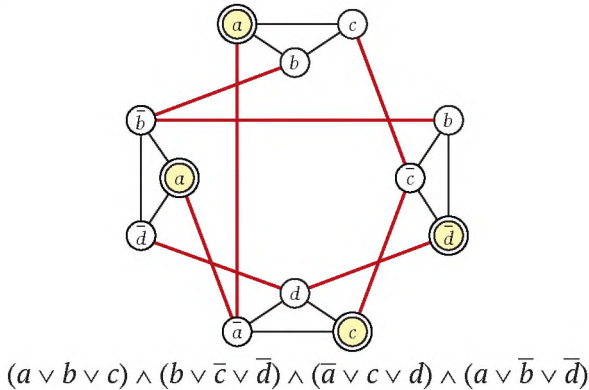


Рис. 12.9. Граф, выведенный из формулы 3CNF с четырьмя компонентами, и независимое множество размером 4

Каждое независимое множество в G содержит не более одной вершины из каждого треугольника компоненты, потому что любые две вершины в треугольнике соединены. Таким образом, наибольшее независимое множество в G имеет размер, не превышающий k . Я утверждаю, что G содержит независимое множество размером ровно k , если и только если исходная формула Φ является выполнимой. Как обычно, доказательство утверждений «если и только если» состоит из двух частей.

⇒ Предположим, что формула Φ выполнима. Зададим произвольное выполняемое присваивание. По определению каждая компонента в Φ содержит как минимум один литерал True. Следовательно, мы можем выбрать подмножество S , состоящее из k вершин в G , которое содержит ровно по одной вершине из каждого треугольника компоненты, такое, что для всех соответствующих k литералов значением является True. Поскольку каждый треугольник содержит не более одной вершины в S , никакие две вершины в S не соединены ребром треугольника. Поскольку каждый литерал, соответствующий вершине в S , имеет значение True, никакие две вершины в S не соединены отрицательным ребром. Отсюда следует, что S является независимым множеством размером k в графе G .

⇐ С другой стороны, предположим, что G содержит независимое множество S размером k . Каждая вершина в S обязательно должна находиться в отдельном треугольнике компоненты. Предположим, что

мы присваиваем значение True каждому литералу в S , а поскольку контрадикторные литералы соединены ребрами, это присваивание является логически верным. Могут существовать переменные x такие, что ни x , ни $\bar{x}$ не соответствуют какой-либо вершине в S , и мы можем установить для этих переменных любое значение по нашему выбору. Поскольку S содержит по одной вершине из каждого треугольника компоненты, каждая компонента в формуле Φ содержит (как минимум) один литерал True. Отсюда следует, что формула Φ является выполняемой.

Преобразование формулы 3CNF Φ в граф G занимает полиномиальное время, даже если мы выполняем все операции методом прямого перебора. Таким образом, если бы мы могли решить задачу MaxIndSet за полиномиальное время, то смогли бы решить задачу 3Sat также за полиномиальное время, выполнив преобразование исходной формулы Φ в граф G и сравнив размер наибольшего независимого множества в G с числом компонент в Φ . Но это должно было предполагать, что $P = NP$, а это бессмысленно. Отсюда следует, что MaxIndSet является NP-трудной задачей.

12.8. Общий шаблон

Все доказательства NP-трудности – и в более общем смысле все редукции за полиномиальное время – следуют некоторой общей схеме. Для приведения задачи X к задаче Y за полиномиальное время необходимо выполнить три действия:

- 1) описать алгоритм, выполняемый за полиномиальное время, для преобразования произвольного экземпляра x из X в специальный экземпляр y из Y ;
- 2) доказать, что если x – «правильный» экземпляр в X , то y – «правильный» экземпляр в Y ;
- 3) доказать, что если y – «правильный» экземпляр в Y , то x – «правильный» экземпляр в X . (Обычно это действие вызывает наибольшие затруднения.)

Разумеется, разработка корректной редукции не означает выполнение этих трех задач одновременно. Сначала записывается алгоритм, который кажется работающим, а затем попытки доказать, что он действительно работает, редко бывают успешными, особенно в условиях ограниченного времени, например на экзаменах. Мы обязательно должны разработать алгоритм и одновременно доказать части «если» и «только если».

Процитирую позднего великого Рики Джея (Ricky Jay)⁹: «Это приобретенное умение».

⁹ ...из его внебродвейского шоу 1996 г. «Рики Джей и 52 его ассистента» («Ricky Jay and his 52 Assistants»).

Один момент, который смущает многих студентов, – тот факт, что алгоритм редукции «работает только в одном направлении» – от X к Y , – но для доказательства корректности требуется «работа в обоих направлениях». Но доказательства корректности в действительности несимметричны. Доказательство части «если» требует обработки произвольных экземпляров X , но для доказательства части «только если» требуется обработка специальных экземпляров Y , сгенерированных алгоритмом редукции. Использование этой асимметрии – ключ к успешному проектированию корректных редукций.

Я нахожу полезным мышление в терминах сертификатов преобразования (transforming certificates) – доказательств того, что заданный экземпляр является «правильным», – совместно с самими экземплярами. Например, сертификат для CircuitSat – это множество входных сигналов, которые включают лампочку, сертификат для Sat или 3Sat – это выполнимое присваивание, сертификат для MaxIndSet – это максимальное независимое множество. Для приведения X к Y в действительности мы должны спроектировать три алгоритма, по одному для каждой следующей задачи:

- для преобразования произвольного экземпляра x из X в специальный экземпляр y из Y за полиномиальное время;
- для преобразования произвольного сертификата для x в сертификат для y ;
- для преобразования произвольного сертификата для y в сертификат для x .

Вторая и третья задачи ссылаются на входные и выходные данные для первого алгоритма. Сертификат преобразования должен быть обратимым, а не преобразованием экземпляра. Мы никогда не должны выполнять преобразования экземпляров Y , и вообще нет необходимости думать о произвольных экземплярах Y . Только первый алгоритм должен выполняться за полиномиальное время (хотя на практике второй и третий алгоритмы почти всегда проще первого).

Например, наша редукция от CircuitSat к 3Sat состоит из трех алгоритмов:

- *первый* преобразует произвольную логическую схему K в специальную логическую формулу 3CNF Φ_3 за полиномиальное время (каждый провод кодируется как переменная, а каждый вентиль – как подформула, затем каждая подформула разворачивается в 3CNF);
- *второй* преобразует произвольный выполнимый ввод для K в выполнимое присваивание для Φ_3 (трассируются входные сигналы в схеме, значения с каждого провода передаются в соответствующие переменные, а всем дополнительным переменным присваиваются произвольные значения);
- *третий* преобразует произвольное выполнимое присваивание для Φ_3 в выполнимые входные сигналы для K (значения всех переменных проводов в Φ_3 передаются в соответствующие провода в K).

Редукция работает, потому что первый алгоритм кодирует любую логическую схему K в сложноструктурированную формулу 3CNF Φ_3 . Особая структура Φ_3 определяет ограничения по тому, как она может быть выполнимой, и каждое выполнимое присваивание для Φ_3 непременно должно «происходить от» некоторого выполнимого ввода для K . Мы вообще не должны думать о произвольных формулах 3CNF.

Аналогично наша редукция от 3Sat к MaxIndSet состоит из трех алгоритмов:

- *первый* преобразует произвольную формулу 3CNF Φ в специальный граф G и специальную целочисленную переменную k за полиномиальное время;
- *второй* преобразует произвольное выполнимое присваивание для Φ в независимое множество в G размером k ;
- *третий* преобразует произвольное независимое множество в G размером k в выполнимое присваивание для Φ .

И в этом случае наше первое преобразование кодирует исходную формулу Φ в сложноструктурированный граф G и специальную переменную k . Структура G гарантирует, что каждое независимое множество размером k «происходит от» выполнимого присваивания для Φ . Мы вообще не рассматриваем произвольные графы или произвольные размеры независимых множеств.

12.9. Клика и вершинное покрытие (от независимого множества)

Клика (clique) – это альтернативное название полного графа, т. е. графа, в котором каждая пара вершин соединена ребром. Задача MaxClique требует найти число узлов в наибольшем полном подграфе заданного графа. Вершинное покрытие (vertex cover) графа – это множество вершин, касающееся каждого ребра в графе. Задача MaxVertexCover требует найти размер наименьшего вершинного покрытия в заданном графе.

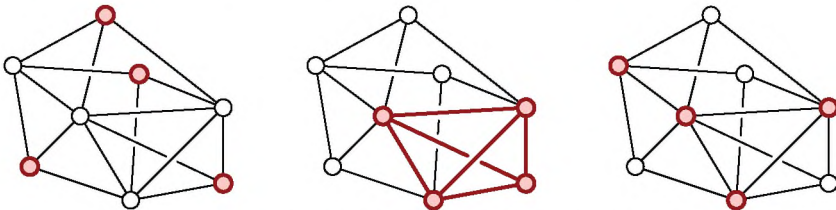


Рис. 12.10. Граф, в котором наибольшее независимое множество, наибольшая клика и наименьшее вершинное покрытие имеют размер 4

Мы можем доказать, что MaxClique является NP-трудной задачей, используя следующую простую редукцию от MaxIndSet. Любой граф имеет реберное дополнение $\bar{G}$ с теми же вершинами, но с зеркально противоположным

множеством ребер – uv является ребром в $\bar{G}$, если и только если uv не является ребром в G . Множество вершин является независимым в G , если и только если те же вершины определяют клику в $\bar{G}$. Таким образом, наибольшее независимое множество в G содержит те же вершины (следовательно, имеет тот же размер), что и наибольшая клика в дополнении G .

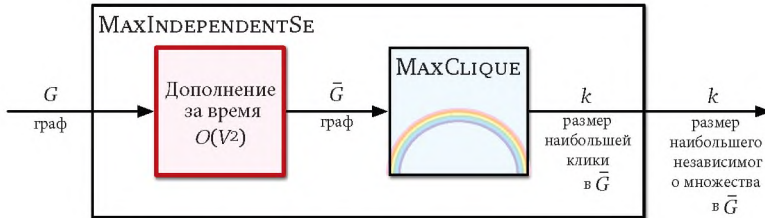


Рис. 12.11. Простая редукция от MaxIndSet к MaxClique

Доказательство того, что MinVertexCover является NP-трудной задачей, еще проще, потому что оно основано на следующем простом утверждении: I – независимое множество в графе $G = (V, E)$, если и только если его дополнение $V \setminus I$ является вершинным покрытием того же графа G . Таким образом, наибольшее независимое множество в любом графе есть дополнение наименьшего вершинного покрытия того же графа. Следовательно, если наименьшее вершинное покрытие в графе с n вершинами имеет размер k , то наибольшее независимое множество имеет размер $n - k$.

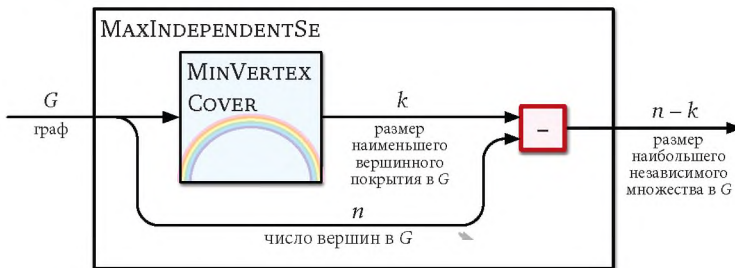


Рис. 12.12. Еще более простая редукция от MaxIndSet к MinVertexCover

12.10. Раскраска графа (от 3Sat)

Корректная k -раскраска (proper k -coloring) графа $G = (V, E)$ – это функция $C : V \rightarrow \{1, 2, \dots, k\}$, которая присваивает один из k «цветов» каждой вершине так, что каждое ребро имеет два различных цвета в своих конечных точках. (В действительности «цветами» могут быть произвольные метки, которые для упрощения мы представляем небольшими целыми числами, а не электромагнитными частотами, векторами СММК или числовыми значениями «Пантон», например.) Задача раскраски графов требует найти наименьшее возможное число цветов для корректной раскраски заданного графа.

Для доказательства того, что раскраска графа является NP-трудной задачей, достаточно рассмотреть решение задачи 3Color: если задан граф,

то имеется ли для него корректная раскраска в три цвета? Докажем, что 3Color является NP-трудной задачей, используя редукцию от 3Sat. (Почему 3Sat? Потому что в названии присутствует 3. Вероятно, вы подумали, что это шутка, но я не шучу.) Если задана формула 3CNF Φ , то мы формируем граф G , для которого возможна раскраска в три цвета, если и только если Φ является выполняемой формулой, как предлагается на рис. 12.13.

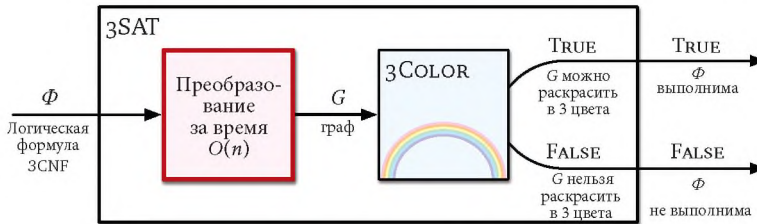


Рис. 12.13. Редукция от 3Sat к 3Color

Мы описываем редукцию, используя стандартную стратегию разбиения выходного графа G на гаджеты (gadgets), подграфы, которые поддерживают различные семантики (смысловые значения) исходной формулы Φ в языке раскраски графов. Разбиение редукции на отдельные гаджеты удобно не только для понимания существующих редукций и доказательства их корректности, но и для проектирования новых NP-трудных редукций¹⁰.

Наша редукция формула Φ граф использует три типа гаджетов:

- существует один гаджет истинности (truth gadget): треугольник с тремя вершинами T , F и X , с понятными обозначениями для True, False и Other соответственно. Поскольку все эти вершины соединены, они непременно должны иметь различные цвета в любой трехцветной раскраске. Для удобства будем называть эти цвета True, False и Other. Таким образом, когда мы говорим, что узел окрашен в цвет True, то подразумеваем, что он имеет тот же цвет, что и вершина T ;
- для каждой переменной a граф содержит гаджет переменной (variable gadget), представленный треугольником, соединяющим два новых узла, помеченные a и $\bar{a}$, с узлом X в гаджете истинности. Узел a обязательно должен быть окрашен в цвет True или False, следовательно, узел $\bar{a}$ обязательно должен быть окрашен в цвет False или True соответственно;

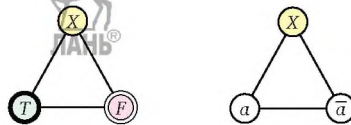


Рис. 12.14. Гаджет истинности и гаджет переменной для a

¹⁰ Наша редукция от CircuitSat к Sat кодирует каждый вентиль в исходной схеме как компоненту в итоговой формуле; эти компоненты являются «гаджетами вентиляей». Аналогично наша редукция от 3Sat к MaxIndSet использовала два типа гаджетов: «гаджеты компонент» (треугольники) и «гаджеты переменных» (ребра между противоположными литералами).

- наконец, для каждой компоненты в формуле Φ граф содержит гаджет компоненты (clause gadget). Каждый гаджет компоненты соединяет три узла литералов (из соответствующих гаджетов переменных) с узлом T (из гаджета истинности), используя пять новых непомеченных узлов и десять ребер, как показано на рис. 12.15.

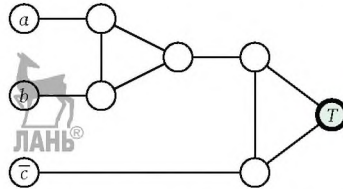


Рис. 12.15. Гаджет компоненты для $(a \vee b \vee \bar{c})$

По существу, каждый треугольник в гаджете компоненты ведет себя как «мажоритарный логический элемент». В любой допустимой схеме трехцветной раскраски если две вершины слева от треугольника имеют одинаковый цвет, то правая крайняя вершина этого треугольника обязательно должна иметь тот же цвет. С другой стороны, если две левые вершины имеют различные цвета, то цвет правой вершины можно выбрать произвольно (см. рис. 12.16).

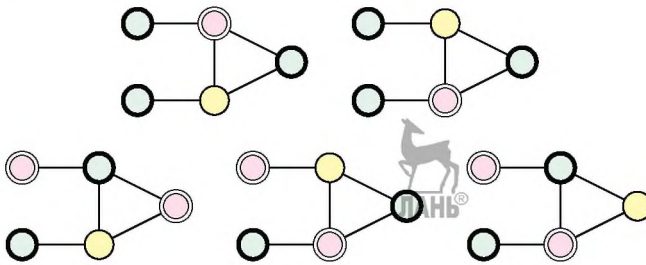
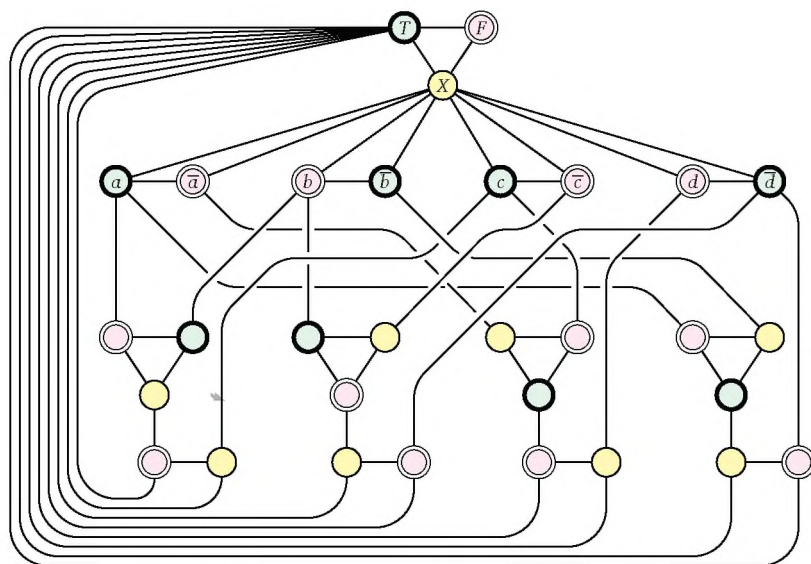


Рис. 12.16. Все допустимые схемы трехцветной раскраски «полугаджета», вплоть до перестановок цветов

Из этого следует, что не существует допустимой трехцветной раскраски гаджета компоненты, в которой все три узла литералов окрашены в цвет False. С другой стороны, любая раскраска узлов литералов более чем одним цветом может быть расширена до допустимой трехцветной раскраски гаджета компоненты. Гаджеты переменных принуждают каждый узел литерала иметь цвет True или False, следовательно, в любой допустимой трехцветной раскраске гаджета компоненты как минимум один узел литерала окрашен в цвет True.

Итоговый граф G содержит ровно один узел T , ровно один узел F и ровно два узла a и $\bar{a}$ для каждой переменной. Например, на рис. 12.17 показан граф, полученный из той же формулы 3CNF $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$, которую мы ранее использовали для иллюстрации редукции MaxIndSet на рис. 12.9. Эта трехцветная раскраска является одним

из нескольких вариантов, соответствующих выполнимому присваиванию $a = c = \text{True}$, $b = d = \text{False}$.



$$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$$

Рис. 12.17. Трехцветная раскраска графа, выведенного из выполнимой формулы 3CNF

Мы уже выполнили большую часть работы по доказательству корректности. Если формула выполнима, то мы можем раскрасить узлы литералов в соответствии с любым выполнимым присваиванием, а затем (поскольку каждая компонента выполнима) распространить раскраску на каждый гаджет компоненты. С другой стороны, если граф можно раскрасить в три цвета, то мы можем извлечь выполнимое присваивание из любой схемы трехцветной раскраски – как минимум один из трех узлов литералов в каждом гаджете компоненты окрашен в цвет True.

Поскольку 3Color является особым случаем более общей задачи раскраски графа, – каково минимальное число цветов? – эта более общая задача оптимизации также является NP-трудной.

12.11. Гамильтонов цикл

Гамильтонов цикл (Hamiltonian cycle) в графе – это цикл, который посещает каждую вершину ровно один раз. (Он абсолютно отличается от эйлера контура, представляющего собой замкнутое блуждание, проходящее по каждому ребру ровно один раз. Эйлеровы контуры легко находить и создавать за линейное время, используя поиск в глубину.) Здесь мы рассмотрим два различных доказательства того, что задача поиска гамильтонова цикла в направленных графах является NP-трудной.

От вершинного покрытия

Наше первое доказательство NP-трудности выполняет редукцию от версии решения задачи вершинного покрытия. Если задан ненаправленный граф G и целое число k , то мы формируем направленный граф H такой, что H содержит гамильтонов цикл, если и только если G содержит вершинное покрытие размером k . Как и в наших предыдущих редукциях итоговый граф H состоит из нескольких гаджетов, каждый из которых соответствует определенным свойствам исходного графа G и целого числа k .

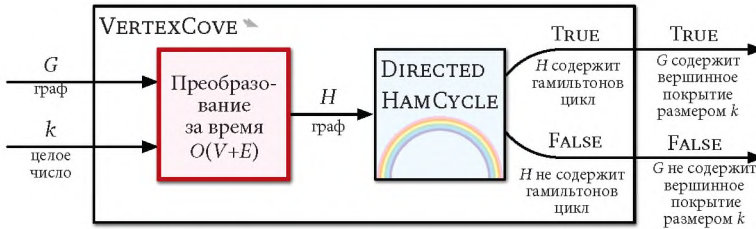


Рис. 12.18. Редукция от VertexCover к HamCycle

- Для каждого ненаправленного ребра uv в G направленный граф H содержит гаджет ребра (edge gadget), состоящий из четырех вершин $(u, v, in), (u, v, out), (v, u, in), (v, u, out)$ и шести направленных ребер:

$$\begin{aligned} (u, v, in) &\rightarrow (u, v, out) & (u, v, in) &\rightarrow (v, u, in) & (v, u, in) &\rightarrow (u, v, in) \\ (v, u, in) &\rightarrow (v, u, out) & (u, v, out) &\rightarrow (v, u, out) & (v, u, out) &\rightarrow (u, v, out), \end{aligned}$$

как показано на рис. 12.19. Каждая вершина «in» имеет дополнительное входящее ребро, а каждая вершина «out» – дополнительное исходящее ребро. Любой гамильтонов цикл в H обязательно должен проходить через гаджет ребра по одному из трех путей – либо прямо через обе стороны, либо в обход от одной стороны к другой и обратно. В итоге эти варианты будут соответствовать обоим вершинам u и v , только u или только v , принадлежащим к одному вершинному покрытию.

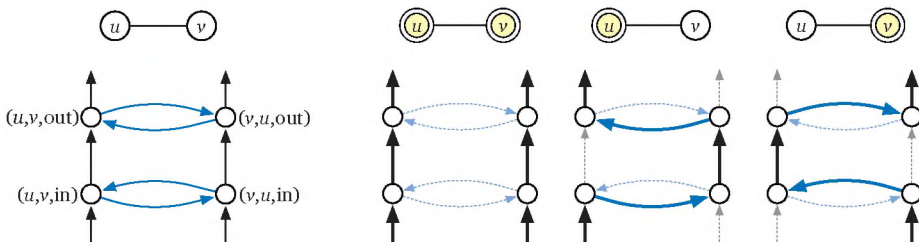


Рис. 12.19. Гаджет ребра и только возможные его пересечения с гамильтоновым циклом

- Для каждой вершины u в G все гаджеты ребер для инцидентных ребер uv соединены в H в единый направленный путь, который мы на-

зываем цепь, вершин (vertex chain). В частности, предположим, что вершина u имеет d соседей $v_1, v_2, \dots, v_d$. Тогда H содержит $d - 1$ дополнительных ребер $(u, v_i, out) \rightarrow (u, v_{i+1}, in)$ для каждого i от 1 до $d - 1$.

- Наконец, H также содержит k вершинных покрытий (cover vertices) $x_1, x_2, \dots, x_k$. Каждое вершинное покрытие содержит направленное ребро к первой вершине в каждой цепи вершин и направленное ребро от последней вершины в каждой цепи вершин.

На рис. 12.20 показан полный пример наших преобразований. Каждая двунаправленная синяя стрелка представляет пару направленных ребер.

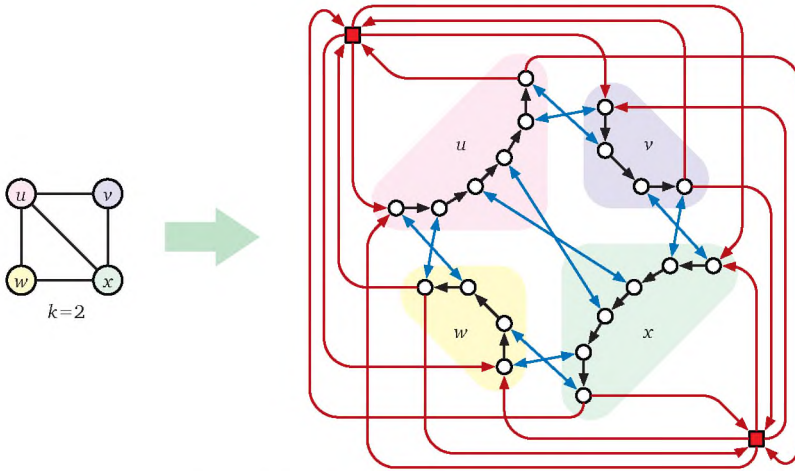


Рис. 12.20. Пример нашей редукции от VertexCover к DirectedHamiltonianCycle

Как обычно, мы доказываем, что наша редукция корректна, в два этапа.

⇒ Сначала предположим, что $C = \{u_1, u_2, \dots, u_k\}$ – вершинное покрытие графа G размером k . Мы можем создать гамильтонов цикл в H , который «кодирует» C , как показано ниже. Для каждого индекса i от 1 до k мы проходим путь из вершинного покрытия x_i через цепь вершин для u_i к вершинному покрытию x_{i+1} (или вершинному покрытию x_1 , если $i = k$). При проходе цепи для каждой вершины u_i мы определяем, как продолжить движение из каждого узла (u_i, v, in) , как показано ниже:

- если $v \in C$, то следовать по ребру $(u_i, v, in) \rightarrow (u_i, v, out)$;
- если $v \notin C$, то выполнить обход через $(u_i, v, in) \rightarrow (v, u_i, in) \rightarrow (v, u_i, out) \rightarrow (u_i, v, out)$.

Таким образом, для каждого ребра uv в G гамильтонов цикл посещает (u, v, in) и (u, v, out) как часть цепи вершин u , если $u \in C$, иначе – как часть цепи вершин v (см. рис. 12.21).

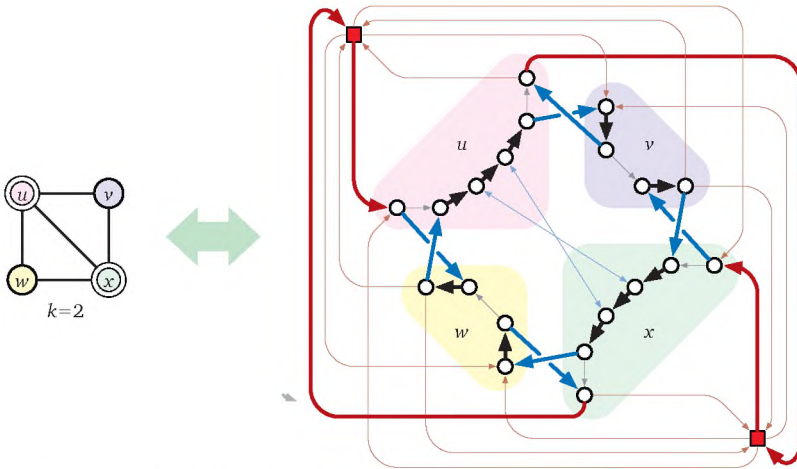


Рис. 12.21. Каждое вершинное покрытие размера k в G соответствует гамильтонову циклу в H и наоборот

$\Leftarrow$ С другой стороны, предположим, что H содержит гамильтонов цикл C . Этот цикл обязательно должен содержать ребро от каждого вершинного покрытия к началу некоторой цепи вершин. В нашем случае анализ гаджетов ребер индуктивно предполагает, что после того, как C входит в цепь вершин для некоторой вершины u , он обязательно должен пройти всю цепь вершин. В частности, в каждой вершине (u, v, in) цикл обязательно должен содержать либо одно ребро $(u, v, in) \rightarrow (u, v, out)$, либо путь обхода $(u, v, in) \rightarrow (v, u, in) \rightarrow (v, u, out) \rightarrow (u, v, out)$, за которым следует ребро к следующему гаджету ребра в цепи вершин u или к вершинному покрытию, если это последний гаджет ребра в цепи вершин u . В частности, если C содержит ребро обхода $(u, v, in) \rightarrow (v, u, in)$, то оно не может содержать ребра между любым вершинным покрытием и цепью вершин v . Из этого следует, что C проходит ровно k цепей вершин. Кроме того, эти цепи вершин описывают вершинное покрытие исходного графа G , так как C посещает вершину (u, v, in) для каждого ребра uv в графе G .

Мы заключаем, что G содержит вершинное покрытие размером k , если и только если H содержит гамильтонов цикл. Преобразование из G в H выполняется за время $O(V^2)$. Из этого следует, что задача поиска направленного гамильтонова цикла является NP-трудной.

От 3Sat

Мы также можем доказать, что задача поиска направленного гамильтонова цикла является NP-трудной, методом редукции непосредственно от задачи 3Sat. Если задана произвольная формула 3CNF Φ с n переменными $x_1, x_2, \dots, x_n$ и k компонентами $c_1, c_2, \dots, c_k$, то мы создаем направленный

граф H , который содержит гамильтонов цикл, если и только если формула Φ выполнима, как показано ниже.

Для каждой переменной x_i мы формируем гаджет переменной, состоящий из двусвязного списка $2k$ вершин $(i, 0), (i, 1), \dots, (i, 2k)$, соединенных ребрами $(i, j-1) \rightarrow (i, j)$ и $(i, j) \rightarrow (i, j-1)$ для каждого индекса j . Мы соединяем первый и последний узел в каждой смежной паре гаджетов переменных дополнительными ребрами:

$$(i, 0) \rightarrow (i+1, 0) \quad (i, 2k) \rightarrow (i+1, 0) \quad (i, 0) \rightarrow (i+1, 2k) \quad (i, 2k) \rightarrow (i+1, 2k)$$

для каждого индекса i ; мы также соединяем конечные точки первого и последнего гаджета переменной ребрами:

$$(n, 0) \rightarrow (1, 0) \quad (n, 2k) \rightarrow (1, 0) \quad (n, 0) \rightarrow (1, 2k) \quad (n, 2k) \rightarrow (1, 2k).$$

Полученный в итоге граф G содержит ровно 2^n гамильтоновых циклов, по одному для каждого присваивания логических значений n переменных в формуле Φ . В частности, для каждого i мы проходим i -й гаджет переменной слева направо, если $x_i = \text{True}$, и справа налево, если $x_i = \text{False}$ (см. рис. 12.22).

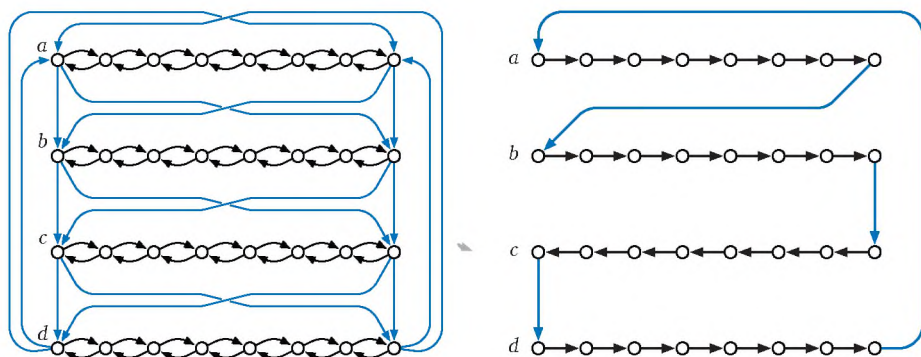


Рис. 12.22. Слева: гаджеты переменных и соединители в графе G для любой формулы с четырьмя переменными и четырьмя компонентами.

Справа: гамильтонов цикл в G , соответствующий присваиванию $a = b = d = \text{True}$ и $c = \text{False}$

Теперь мы расширяем G до более крупного графа H , добавляя вершину компоненты (clause vertex) $[j]$ для каждой компоненты c_j , соединенную с гаджетами переменных шестью ребрами, как показано на рис. 12.23. Для каждого положительного литерала x_i в c_j мы добавляем ребра $(i, 2j-1) \rightarrow [j] \rightarrow (i, 2j)$, а для каждого отрицательного литерала x_i в c_j – ребра $(i, 2j) \rightarrow [j] \rightarrow (i, 2j-1)$. Соединения с вершинами компонент гарантируют, что гамильтонов цикл в G может быть расширен до гамильтонова цикла в H , если и только если соответствующее присваивание переменным делает выполнимой формулу Φ . Теперь анализ исчерпывающего случая пред-

полагает, что H содержит гамильтонов цикл, если и только если формула Φ выполнима.

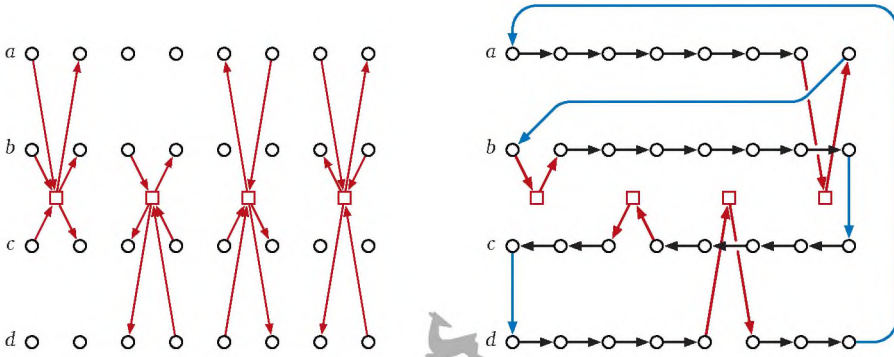


Рис. 12.23. Слева: гаджеты компонент для формулы $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee d) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$.

Справа: гамильтонов цикл в графе H , соответствующий выполнимому присваиванию $a = b = d = \text{True}$ и $c = \text{False}$

Преобразование формулы Φ в граф H выполняется за время $O(kn)$, которое не более чем квадратично относительно общей длины формулы, поэтому мы заключаем, что задача поиска направленного гамильтонова цикла является NP-трудной.

Варианты и расширения

Тривиальные модификации предыдущих редукций подразумевают, что задача поиска гамильтонова пути в направленных графах также является NP-трудной. Гамильтонов путь (Hamiltonian path) в графе G – это, конечно же, простой путь, который посещает каждую вершину G ровно один раз. В действительности существуют простые редукции, выполняемые за полиномиальное время, от задачи поиска гамильтонова цикла к задаче поиска гамильтонова пути и наоборот. Подробности этих редукций я оставляю как упражнения.

Обе предыдущие редукции имели дело с направленными графами, но соответствующий вопрос для ненаправленных графов также является NP-трудной задачей. В действительности существует относительно простая редукция от задачи поиска направленного гамильтонова цикла/пути к задаче ненаправленного гамильтонова цикла/пути. И в этом случае я оставляю подробности этой редукции как упражнение.

Наконец, имеющая дурную славу задача коммивояжера (traveling salesman problem) требует найти кратчайший гамильтонов цикл (или путь) в графе с взвешенными ребрами. Поскольку поиск кратчайшего цикла/пути очевидно труднее, чем определение существования цикла/пути вообще, – рассматривается граф, в котором каждое ребро имеет вес 1, – TravelingSalesman также является NP-трудной задачей.

12.12. Задача о сумме подмножеств (от задачи вершинного покрытия)

Следующая задача, для которой мы докажем NP-трудность, – задача о сумме подмножеств SubsetSum, рассмотренная в главе 2: задано множество X положительных целых чисел и целое число T , определить, содержит ли X подмножество, сумма элементов которого равна T .

Снова применим редукцию от задачи VertexCover. Задан граф G и целое число k , нам необходимо вычислить множество положительных целых чисел X и целое число T такое, что X содержит подмножество, сумма элементов которого равна T , если и только если G содержит вершинное покрытие размером k . Наше преобразование использует только два типа «гаджетов» – целые числа, представляющие вершины и ребра графа G .

Пронумеруем ребра в G произвольно от 0 до $E - 1$. Наше множество X содержит целое число $b_i := 4^i$ для каждого ребра i и целое число

$$a_v := 4^E + \sum_{i \in \Delta(v)} 4^i$$

для каждой вершины v , где $\Delta(v)$ – множество ребер, для которых v является конечной точкой. С другой стороны, мы можем считать каждое целое число в X $(E + 1)$ -разрядным числом, записанным по основанию 4. Каждый E -й разряд равен 1, если целое число представляет вершину, и 0 в противном случае. Для каждого $i < E$ i -й разряд равен 1, если целое число представляет ребро i или одну из его конечных точек, и 0 в противном случае. После этого мы устанавливаем целевую сумму:

$$T := k \cdot 4^E + \sum_{i=0}^{E-1} 2 \cdot 4^i.$$

Теперь докажем, что эта редукция корректна.

⇒ Сначала предположим, что G содержит вершинное покрытие C размером k . Рассмотрим подмножество

$$X' := \{a_v \mid v \in C\} \cup \{b_i \mid \text{ребро } i \text{ имеет ровно одну конечную точку в } C\}.$$

Сумма элементов X' , записанная по основанию 4, содержит самый старший разряд k , а все остальные разряды равны 2. Следовательно, сумма элементов X' в точности равна T .

⇐ С другой стороны, предположим, что существует подмножество $X' \subseteq X$, сумма элементов которого равна T . В частности, мы обязательно имеем

$$\sum_{v \in V'} a_v + \sum_{i \in E'} b_i = t$$

для некоторых подмножеств $V' \subseteq V$ и $E' \subseteq E$. И снова если мы просуммируем эти числа по основанию 4, то не существует переносов

в первых E разрядах, так как для каждого i существует только три числа в X , i -е разряды которых равны 1. Каждое число ребра b_i вносит только одну 1 в i -й разряд суммы, но i -й разряд t равен 2. Следовательно, для каждого ребра в G как минимум одна из его конечных точек обязательно должна находиться в V' . Другими словами, V' является вершинным покрытием. С другой стороны, только числа вершин больше 4^E и $\lfloor T/4^E \rfloor = k$, поэтому V' содержит не более k элементов. (В действительности легко видеть, что V' содержит ровно k элементов.)

Например, если задан граф с четырьмя вершинами $G = (V, E)$, где $V = \{u, v, w, x\}$ и $E = \{uv, uw, vx, wx\}$, то наше множество X может содержать следующие числа с основанием 4:

$$\begin{array}{ll} a_u := 111000_4 = 1344 & b_{uv} := 010000_4 = 256 \\ a_v := 110110_4 = 1300 & b_{uw} := 001000_4 = 64 \\ a_w := 101101_4 = 1105 & b_{vw} := 000100_4 = 16 \\ a_x := 100011_4 = 1029 & b_{vx} := 000010_4 = 4 \\ & b_{wx} := 000001_4 = 1. \end{array}$$

Если мы ищем вершинное покрытие размером $k = 2$, то наша целевая сумма должна быть равна $T := 222222_4 = 2730$. Разумеется, вершинное покрытие $\{v, w\}$ соответствует множеству $\{a_v, a_w, b_{uv}, b_{uw}, b_{vx}, b_{wx}\}$, сумма которого равна $1300 + 1105 + 256 + 64 + 4 + 1 = 2730$.

Эта редукция явно может быть выполнена за полиномиальное время. Мы уже доказали, что VertexCover является NP-трудной задачей, а из этого следует, что SubsetSum также является NP-трудной задачей.

Да будет осмотрителен выполняющий редукцию!

Здесь необходимо отметить одну тонкость. Около 300 страниц назад в главе 3 мы разработали алгоритм динамического программирования для решения задачи SubsetSum за время $O(nT)$. Разве этот алгоритм не выполняется за полиномиальное время? Разве мы только что не доказали, что $P = NP$? Эй, где мой миллион долларов?!

Увы, жизнь не настолько проста. Действительно, время выполнения этого алгоритма является полиномиальной функцией от переменных n и T , но для точного определения как истинного алгоритма с полиномиальным временем выполнения это время обязательно должно быть полиномиальной функцией от размера входных данных (input size), т. е. от числа битов, требуемых для представления входных данных. Значения элементов множества X и целевой суммы T могут экспоненциально превышать число битов входных данных. Разумеется, в только что описанной редукции генерируется значение T , экспоненциально превышающее размер нашего исходного графа, а это вынуждает наш алгоритм динамического программирования выполняться за экспоненциальное время.

Алгоритмы, подобные этому, называют выполняющимися за псевдополиномиальное время (pseudo-polynomial time), а любая NP-трудная задача с таким алгоритмом называется неустойчиво NP-трудной задачей (weakly NP-hard). Равнозначно неустойчиво NP-трудная задача является задачей, которую можно решить за полиномиальное время, если все входные числа представлены в унарном виде (как сумма единиц), но становится NP-трудной, когда все входные числа представлены в двоичном (бинарном) виде. Если задача является NP-трудной, даже если все входные числа представлены в унарном виде, мы говорим, что эта задача является строго NP-трудной (strongly NP-hard). Хорошим примером строго NP-трудной задачи является TravelingSalesman (задача коммивояжера), которая остается NP-трудной, даже если входной граф является полным и все веса ребер равны 1 или 2.

12.13. Другие полезные NP-трудные задачи

Буквально для тысяч задач была доказана их NP-трудность. Здесь я приведу лишь несколько NP-трудных задач, полезных при выводе редукций¹¹. Я не буду подробно описывать доказательства NP-трудности для этих задач, но большинство этих доказательств вы можете найти в классической «Черной страшной книге по NP-полноте» (Scary Black Book of NP-Completeness) Гэри (Garey) и Джонсона (Johnson)¹². Для всех рассмотренных до сего момента задач и для задач в приведенном ниже списке доказательство NP-трудности впервые было представлено в одной исторически значимой работе Ричарда Карпа (Richard Karp) в 1972 г.¹³

- **PlanarCircuitSat**: задана логическая схема, которая может быть встроена в плоскость так, что никакие два провода не пересекаются. Существуют ли входные данные, которые формируют вывод схемы True? NP-трудность этой задачи можно доказать редукцией от общей задачи выполнимости схемы, заменив каждое пересечение проводов небольшой комбинацией вентиляей.
- **1-in-3Sat**: задана формула 3CNF. Существует ли присваивание значений переменным такое, что каждая компонента содержит ровно один литерал True? NP-трудность этой задачи можно доказать редукцией от обычной задачи 3Sat.
- **NotAllEqual3Sat**: задана формула 3CNF. Существует ли присваивание значений переменным такое, что каждая компонента содержит как минимум один литерал True и как минимум один литерал False? NP-трудность этой задачи можно доказать редукцией от обычной задачи 3Sat.

¹¹ Поскольку когда-то может случиться так, что крайне необходимо найти редукцию, я привел небольшой список. Для некоторых архисложных задач мы можем использовать более низкую границу. Быстрых решений не существует. Могут помочь наши доказательства.

¹² Michael Garey and David Johnson. «Computers and Intractability: A Guide to the Theory of NP-Completeness». W. H. Freeman and Co., 1979. Да, она действительно черного цвета.

¹³ В дальнейшем исполняемой за пределами бродвейских театров как «Ричард Карп и 21 его ассистент», за которую Карп получил заслуженную премию Тони Тьюринга.

- **Planar3Sat**: задана формула 3CNF. Рассмотреть двудольный граф, вершинами которого являются компоненты и переменные, а ребра обозначают, что переменная (или ее отрицание) присутствует в компоненте. Если этот граф плоский, то формула 3CNF также называется плоской. Задача Planar3Sat требует при заданной формуле 3CNF определить, имеется ли для этой формулы выполнимое присваивание. NP-трудность этой задачи можно доказать редукцией от задачи PlanarCircuitSat¹⁴.
- **Exact3DimensionalMatching** или **X3M**: задано множество S и группа подмножеств S из трех элементов, называемых тройками. Существует ли подгруппа непересекающихся троек, которая полностью покрывает S ? NP-трудность этой задачи можно доказать редукцией от задачи 3Sat, поскольку в ней имеются три элемента.
- **Partition**: задано множество S из n целых чисел. Существуют ли подмножества A и B такие, что $A \cup B = S$, $A \cap B = \emptyset$ и

$$\sum_{a \in A} a = \sum_{b \in B} b?$$

NP-трудность этой задачи можно доказать простой редукцией от задачи SubsetSum. Как и SubsetSum, задача Partition является всего лишь неустойчиво NP-трудной.

- **3Partition**: задано множество S из $3n$ целых чисел. Можно ли разделить его на n непересекающихся подмножеств из трех элементов таких, что каждое подмножество имеет абсолютно одинаковую сумму? Несмотря на похожие названия, эта задача абсолютно отличается от задачи Partition; извините, не я придумал эти названия. NP-трудность этой задачи можно доказать редукцией от задачи X3M, так как в ней имеются три элемента. В отличие от Partition, задача 3Partition является строго NP-трудной, даже если каждое входное число не превышает n^3 .
- **SetCover**: задана группа множеств $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$. Найти наименьшую подгруппу множеств S_i , содержащую все элементы $\bigcup_i S_i$. Эта задача является обобщением двух задач: VertexCover и X3M.
- **HittingSet**: задана группа множеств $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$. Найти наименьшее число элементов в $\bigcup_i S_i$, которые попадают в каждое множество в $\mathcal{S}$. Эта задача также является обобщением задачи VertexCover.
- **LongestPath**: задан неотрицательный взвешенный граф G (направленный или ненаправленный) и две вершины u и v . Найти самый длинный простой путь от u к v в этом графе. Путь является простым, если он посещает каждую вершину не более одного раза. Эта задача является обобщением соответствующей задачи поиска гамильтонова пути. Разумеется, соответствующая задача поиска наикратчайшего пути может быть решена за полиномиальное время.

¹⁴ Удивительно, но задача PlanarNotAllEqual3Sat решается за полиномиальное время.

- **SteinerTree**: задан взвешенный ненаправленный граф G , в котором некоторые вершины помечены. Найти поддерево минимального веса в G , которое содержит каждую помеченную вершину. Если каждая вершина помечена, то минимальное дерево Штайнера является минимальным остовным деревом. Если ровно две вершины помечены, то минимальное дерево Штайнера является кратчайшим путем между этими вершинами. NP-трудность этой задачи можно доказать редукцией от задачи VertexCover.
- **Max2Sat**: задана логическая формула в конъюнктивной нормальной форме с ровно двумя литералами в каждой компоненте. Найти присваивание переменным, которое максимизирует число компонент, содержащих как минимум один литерал True. NP-трудность этой задачи можно доказать редукцией от задачи 3Sat (именно так, хотя в этой задаче не три элемента). Более простая задача 2Sat, в которой спрашивается, существует ли присваивание, которое выполняет каждую компоненту, действительно может быть решена за полиномиальное время.
- **MaxCut**: задан ненаправленный граф $G = (V, E)$. Найти подмножество $S \subset V$, которое максимизирует число ребер с ровно одной конечной точкой в S . Равнозначно: найти наибольший двудольный подграф для G . NP-трудность этой задачи можно доказать редукцией от задачи Max2Sat.

В дополнение к этим сухим, но полезным задачам было показано, что наиболее интересные головоломки и игры-пасьянсы типа «солитер» также являются NP-трудными или имеют NP-трудные обобщения. (Вероятно, если игра или головоломка не является как минимум NP-трудной, то она не интересна.) Вот несколько примеров, которые, возможно, вам знакомы:

- Minesweeper (Сапер) (от CircuitSat)¹⁵;
- Sudoku (Судоку) (по существу, от 3Sat)¹⁶;
- Tetris (Тетрис) (от 3Partition)¹⁷;
- Klondike, он же Solitaire (Солитер) (от 3Sat)¹⁸;
- Pac-Man (Пакман) (от HamiltonianCycle)¹⁹;
- Super Mario Brothers (Братья Супер-Марио) (от 3Sat)²⁰;

¹⁵ Richard Kaye. Minesweeper is NP-complete. Mathematical Intelligencer 22(2):9–15, 2000. Смотрите также: Allan Scott, Ulrike Stege, and Iris van Rooij. Minesweeper may not be NP-complete but is hard nonetheless. Mathematical Intelligencer 33(4):5–17, 2011.

¹⁶ Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E86-A(5):1052–1060, 2003. <http://www.imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf>.

¹⁷ Ron Breukelaar, Erik D. Demaine, Susan Hohenberger, Hendrik J. Hoogeboom, Walter A. Kosters, and David Liben-Nowell. Tetris is hard, even to approximate. International Journal of Computational Geometry and Applications 14:41–68, 2004.

¹⁸ Luc Longpré and Pierre McKenzie. The complexity of Solitaire. Proceedings of the 32nd International Mathematical Foundations of Computer Science, 182–193, 2007.

¹⁹ Giovanni Viglietta. Gaming is a hard job, but someone has to do it! Theory of Computing Systems, 54(4):595–621, 2014. <http://giovanniviglietta.com/papers/gaming2.pdf>.

²⁰ Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games Are (computationally) hard. Theoretical Computer Science 586:135–160, 2015. <http://arxiv.org/abs/1203.1895>.

- Candy Crush Saga (от варианта 3Sat)²¹;
- Threes/2048 (от 3Sat, разумеется)²²;
- Trainyard (Ж/д стрелки) (от DominatingSet; см. упражнение 26)²³;
- кратчайшее решение кубика Рубика $n \times n \times n$ (от 3Sat через особый случай задачи PlanarUndirectedHamCycle)²⁴;
- Cookie Clicker (от Partition или 3Partition)²⁵.

Разумеется, это далеко не полный список из-за ограниченного бюджета на носки.²⁶ К июню 2019 г. никто не опубликовал доказательство того, что обобщение Ultimate Paperclips, Line Rider, Twister или Cards Against Humanity является NP-трудной задачей, но я уверен, что это лишь вопрос времени.

12.14. Выбор правильной задачи

Одним из самых трудных этапов при доказательстве того, что задача является NP-трудной, является выбор наиболее подходящей задачи, от которой будет выполняться редукция. Теорема Кука–Левина предполагает, что если существует редукция от любой NP-трудной задачи к задаче X, то существует и редукция от каждой NP-полной задачи к задаче X, но с некоторыми задачами работать проще, чем с другими. Систематической методики для выбора правильной задачи не существует, но ниже приведено несколько полезных практических правил.

- Если в задаче спрашивается, как присвоить биты объектам, или как выбрать подмножество объектов, или как разделить объекты на два различных подмножества, то попробуйте редукцию от какой-либо версии Sat или Partition.
- Если в задаче спрашивается, как присвоить метки объектам из небольшого фиксированного множества или как разделить объекты на небольшое число подмножеств, то попробуйте редукцию от kColor или даже 3Color.
- Если в задаче предлагается разместить множество объектов в определенном порядке, то попробуйте редукцию от DirectedHamCycle, DirectedHamPath или TravelingSalesman.

²¹ Luciano Gualà, Stefano Leucci, Emanuele Natale. Bejeweled, Candy Crush and other match-three games are (NP-)hard. Proc. 2014 IEEE Conference on Computational Intelligence and Games, 2014. <http://arxiv.org/abs/1403.5830>.

²² Stefan Langerman and Yushi Uno. Threes!, Fives, 1024!, and 2048 are Hard. Proc. 8th International Conference on Fun with Algorithms, 2016. <https://arxiv.org/abs/1505.04274>.

²³ Matteo Almanza, Stefano Leucci, and Alessandro Panconesi. Trainyard is NP-Hard. Proc. 8th International Conference on Fun with Algorithms, 2016. <https://arxiv.org/abs/1603.00928>.

²⁴ Erik D. Demaine, Sarah Eisenstat, and Mikhail Rudoy. Solving the Rubik's Cube optimally is NP-complete. Proc. 35th Symposium on Theoretical Aspects of Computer Science, 2018. <https://arxiv.org/abs/1706.06708>.

²⁵ Erik D. Demaine, Hiro Ito, Stefan Langerman, Jayson Lynch, Mikhail Rudoy, and Kai Xiao. Cookie Clicker. Preprint, August 2018. <https://arxiv.org/abs/1808.07540>.

²⁶ Смотрите: <https://xkcd.com/1208/>.

- Если в задаче предлагается найти небольшое подмножество, выполняющее некоторые ограничения, то попробуйте редукцию от `MinVertexCover`.
- Если в задаче предлагается найти большое подмножество, выполняющее некоторые ограничения, то попробуйте редукцию от `MaxIndSet`, `MaxClique` или `Max2Sat`.
- Если в задаче предлагается разделить объекты на большое число малых подмножеств, то попробуйте редукцию от `3Partition`.
- Если число 3 естественным образом присутствует в задаче, то попробуйте `3Sat`, `3Color`, `X3M` или `3Partition`. (Нет, это не шутка.)
- Если ничего подходящего не нашлось, то попробуйте `3Sat` или даже `CircuitSat`.

Я не рекомендую пытаться выполнить редукцию от `Tetris`, `SuperMarioBros` или `Trainyard`. Вам действительно необходимо выбрать самую простую начальную задачу, но при этом сохранить некоторое характерное свойство вашей задачи, которое делает ее трудной для решения.

12.15. Простой пример из реальной жизни

Шашки (draughts) – это семейство настольных игр, в которые играют уже несколько тысяч лет. Большинство американцев знакомо с версией под названием *checkers* или *English draughts*, но самым известным в мире вариантом являются *международные* или *польские шашки* (international или Polish draughts), появившиеся в Нидерландах в XVI в. Полный комплект правил можно найти в «Википедии», а здесь приведено несколько важных отличий от англо-американской версии игры:

- *летучие дамки*: как и в варианте *checkers*, шашка, завершающая ход на горизонтали, ближайшей к сопернику, становится дамкой (king) и получает возможность ходить назад. Но, в отличие от *checkers*, дамка в международных шашках за один ход может перемещаться на любое расстояние по диагонали, если промежуточные клетки пусты или на одной из них находится ровно одна шашка другого цвета (которая побивается и снимается с доски);
- *обязательное взятие максимально возможного числа шашек*: при каждом ходе игрок обязательно должен побить все возможные шашки другого цвета. Это правило отличается от правила обязательного взятия в *checkers*, которое требует лишь того, что каждый игрок обязан побить шашку соперника, если это возможно, и что ход со взятием завершается, только когда ходящая шашка не может взять очередную шашку другого цвета. Другими словами, в *checkers* требуется взятие локально максимального множества шашек другого цвета при каждом ходе, тогда как международные шашки требуют глобально максимального взятия;

- *тонкости взятия*: как и в варианте checkers, побитые шашки удаляются с доски только после завершения хода. Любая шашка может быть побита не более одного раза. Таким образом, когда совершен прыжок через шашку соперника, она остается на доске, но через нее нельзя перепрыгивать во второй раз до завершения текущего хода.

Например, в первой позиции, показанной на рис. 12.24, каждый кружок обозначает шашку, а двойные кружки – дамки. Черные обязаны сделать первый обозначенный стрелками ход, побивая пять белых шашек, потому что нет возможности взять более пяти шашек и нет другого хода, при котором побивается пять шашек. Черные не могут продолжить взятие ни на северо-восток, ни на юго-восток, потому что побитые белые шашки остаются на доске до полного завершения хода. Затем белые обязаны сделать второй обозначенный стрелками ход, выигрывая партию.

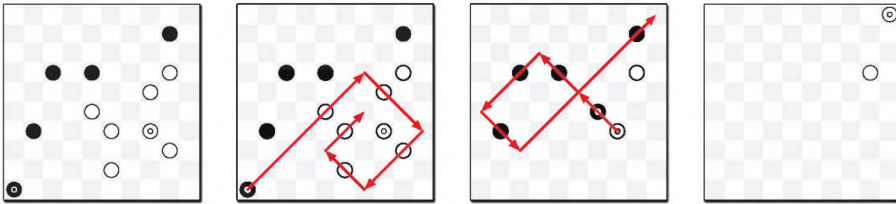


Рис. 12.24. Два обязательных (!) хода в международных шашках.
Двойными кружками обозначены дамки

Реальная игра, которая происходит на доске размером 10×10 клеток с 20 шашками каждого цвета, тривиальна с точки зрения вычислений. Мы можем предварительно вычислить оптимальный ход для обоих игроков в каждой возможной позиции на доске и жестко закодировать результаты в таблице поиска постоянного размера. Разумеется, это большая константа, но она остается константой.

Но рассмотрим естественное обобщение международных шашек на доске размером $n \times n$. При таком условии поиск правильного хода действительно становится NP-трудной задачей. Описанная ниже редукция от задачи поиска гамильтонова цикла в направленных графах была открыта Робертом Херном (Robert Hearn) в 2010 г.²⁷ В большинстве игр с двумя участниками поиск наилучшего хода является NP-трудной задачей (или сложнее). Это единственный известный мне пример игры – более того, реальной игры, в которую веками играют миллионы людей, – где строгое следование правилам является NP-трудной задачей.

Задан ненаправленный граф G с n вершинами, и мы формируем позицию на доске для международных шашек такую, что белые могут побить определенное число черных шашек за один ход, если и только если G содержит гамильтонов цикл. Мы интерпретируем G как направленный граф с двумя дугами $u \rightarrow v$ и $v \rightarrow u$ вместо каждого ненаправленного ребра 1. Число

²⁷ Смотрите: Theoretical Computer Science Stack Exchange: <http://csttheory.stackexchange.com/a/1999/111>.

вершин произвольное от 1 до n . Конечная позиция шашек имеет несколько гаджетов:

- вершины G представлены гаджетами вершин в форме головы кролика, равномерно распределенными вдоль горизонтальной прямой. Каждая дуга $i \rightarrow j$ представлена путем из двух диагональных отрезков прямой от «левого уха» гаджета вершины i к «правому уху» гаджета вершины j . Путь для дуги $i \rightarrow j$ размещен над гаджетами вершин, если $i < j$, и под гаджетами вершин, если $i > j$;

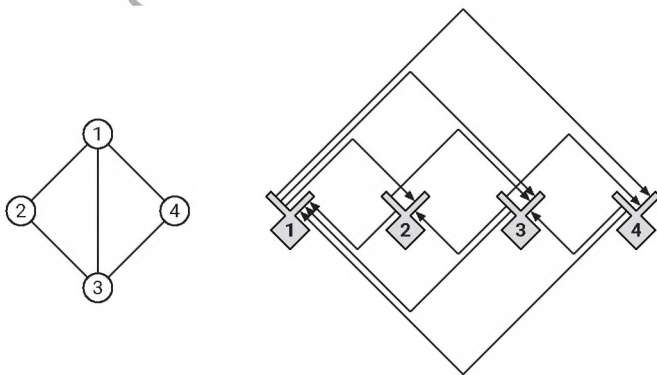


Рис. 12.25. Обзор высокого уровня для редукции от гамильтонова цикла к международным шашкам

- объем (bulk) каждого гаджета вершины – это область в форме ромба, называемая хранилищем (vault). Стены хранилища сделаны из двух прочных слоев черных шашек, которые невозможно побить; эти шашки на иллюстрациях изображены как серые кружки. Внутри каждого хранилища существует N черных шашек, которые можно побить, для некоторого большого целого числа N , которое будет определено позже. Белая дамка может войти в хранилище через «правое ухо», побить каждую внутреннюю шашку, а затем выйти через «левое ухо». Оба уха являются коридорами также со стенами толщиной в две шашки и с входами, где концы путей дуг позволяют белой дамке войти и выйти. Длины «ушей» можно легко отрегулировать, чтобы выровнять их с другими гаджетами;
- для каждой дуги $i \rightarrow j$ мы имеем угловой гаджет (corner gadget), позволяющий белой дамке выйти из гаджета вершины i с перенаправлением в гаджет вершины j ;
- наконец, когда две дуги пересекаются, мы имеем гаджет пересечения (crossing gadget). Эти гаджеты позволяют белой дамке пересечь путь другой дуги, но запрещают переходить с одного пути дуги на другой.

Одна белая дамка начинает с нижнего угла одного из хранилищ. При любом разрешенном ходе эта дамка обязана менять пути обхода всей дуги в

целом и полностью очищенные хранилища. Дамка может проходить различные гаджеты в обратном порядке, входя в каждое хранилище через выход и наоборот. Но цикл, обратный гамильтонову циклу в графе G , является другим гамильтоновым циклом в G , так что блуждание в обратном порядке штрафуются.

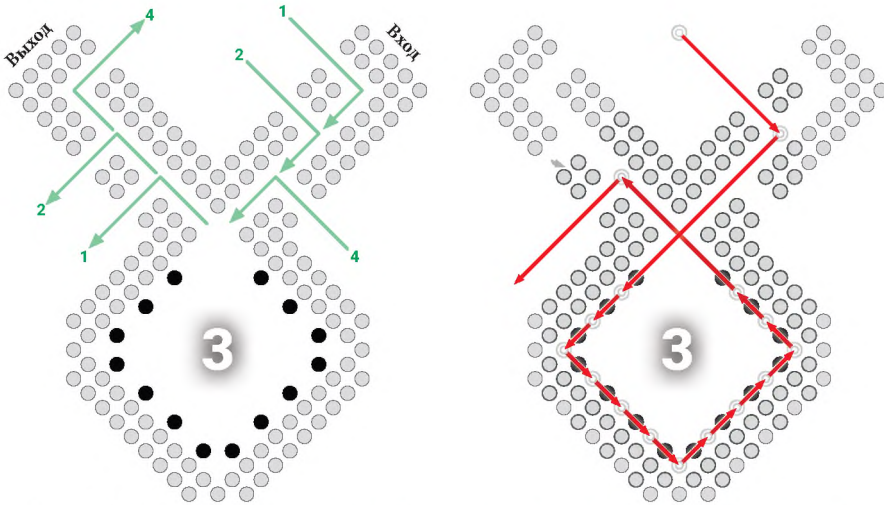


Рис. 12.26. Слева: гаджет вершины с тремя входами и тремя выходами. Справа: белая дамка опустошает хранилище. Серые кружки – это черные шашки, которые нельзя побить

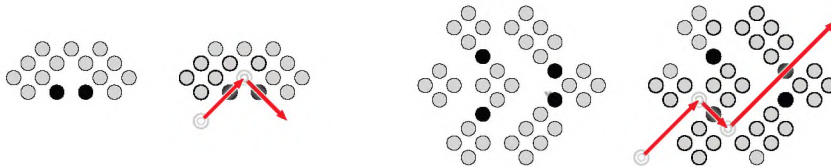


Рис. 12.27. Слева: один из двух путей через угловой гаджет. Справа: один из двух путей через гаджет пересечения

Если в графе G существует гамильтонов цикл, то белая дамка может взять как минимум nN черных шашек, посетив каждое другое хранилище и вернувшись в начальное. С другой стороны, если в G не существует гамильтонова цикла, то белая дамка может взять не более половины шашек в начальном хранилище, следовательно, в общей сложности может взять не более $(n - 1/2)N + O(n^3)$ шашек соперника. Член $O(n^3)$ учитывает угловые гаджеты и гаджеты пересечения, каждое ребро проходит через один угловой гаджет и через не более чем $1^2/2$ гаджетов пересечения.

Для завершения редукции мы устанавливаем $N = n^4$. Суммируя, получаем конфигурацию доски $O(n^5) \times O(n^5)$ с $O(n^5)$ черными шашками и одной белой дамкой. Мы можем явно создать эту конфигурацию доски прямым перебором за полиномиальное время. На рис. 12.28 показан полностью завершённый пример такого формирования.

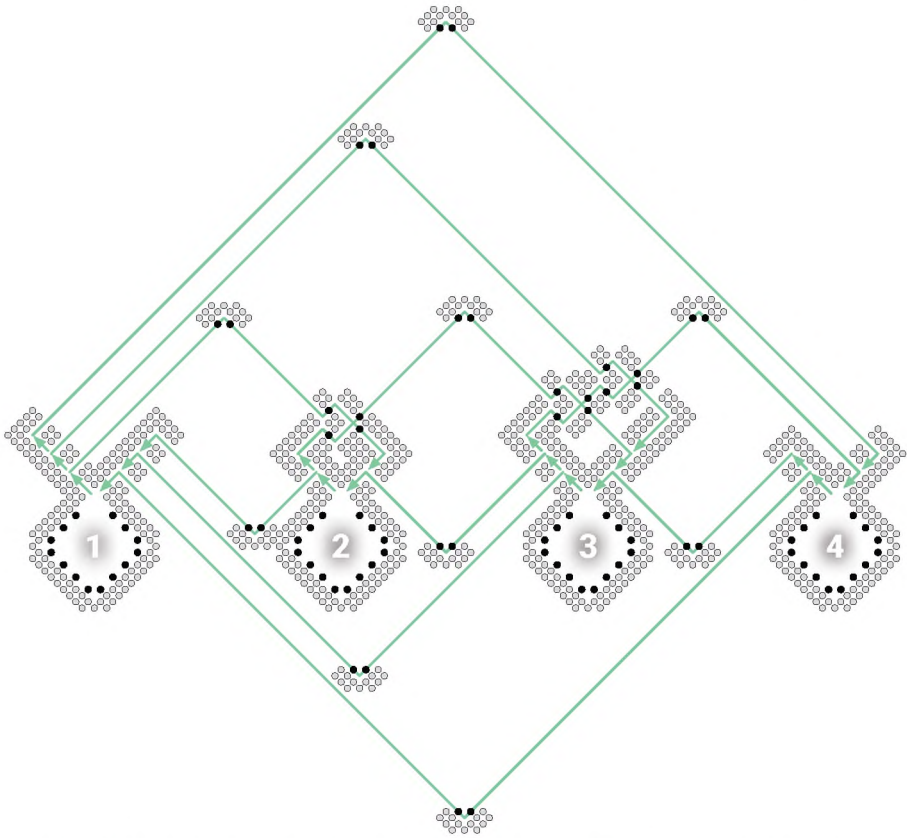


Рис. 12.28. Окончательная конфигурация шашечной доски для графа с четырьмя вершинами, показанного на рис. 12.25. (Зеленые стрелки не являются частью этой конфигурации)

Остается открытым вопрос – является ли следующая, связанная с рассматриваемой, задача NP-трудной: если задана конфигурация доски $n \times n$ для международных шашек, то могут ли (и, следовательно, обязаны) белые взять все черные шашки (т. е. выиграть партию) за один ход?

♥ 12.16. Что дальше

P и NP – это всего лишь первые два шага в огромной иерархии классов сложности. Для завершения этой главы (и всей книги) позвольте мне описать еще несколько интересных классов.

Полиномиальное пространство

PSPACE – это множество задач на принятие решений, которые могут быть решены с использованием полиномиального пространства (polynomial space). Каждая задача в NP (следовательно, и в P) также находится в

PSPACE. В общем, все верят, что $NP \neq PSPACE$, но никто не может доказать даже, что $P \neq PSPACE$. Задача Π является *PSPACE-трудной*, если для любой задачи Π' , которая может быть решена с использованием полиномиального пространства, существует множественная редукция, выполняемая за полиномиальное время, от Π' к Π . Если любая PSPACE-трудная задача находится в NP, то $PSPACE = NP$. Аналогично если любая PSPACE-трудная задача находится в P, то $PSPACE = P$.

Канонической PSPACE-трудной является задача квантифицированной логической формулы (quantified boolean formula – QBF): если задана логическая формула Φ , которая может включать любое число кванторов всеобщности или существования, но без свободных переменных, то равна ли формула Φ значению True? Например, следующее выражение представляет собой корректные входные данные для задачи QBF:

$$\exists a: \forall b: \exists c: (\forall d: a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee (\exists e: (\overline{a \Rightarrow e}) \vee (c \neq a \wedge e))).$$

Задача Sat равнозначна особому случаю задачи QBF, в котором исходная формула содержит только кванторы существования ($\exists$). Задача QBF остается PSPACE-трудной, даже если исходная формула непременно должна иметь все эти кванторы в начале со строгим чередованием кванторов $\exists$ и $\forall$ с квантифицированным утверждением в конъюнктивной нормальной форме с ровно тремя литералами в каждой компоненте, например:

$$\exists a: \forall b: \exists c: \forall d: ((a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})).$$

Эта ограниченная версия задачи QBF также может быть сформулирована как задача стратегии в игре с двумя участниками. Предположим, что для двух игроков, Алисы и Боба, задан предикат 3CNF со свободными переменными $x_1, x_2, \dots, x_n$. Игроки поочередно присваивают значения переменным в порядке возрастания индекса – Алиса присваивает значение x_1 , Боб – x_2 и т.д. В итоге Алиса присваивает значения каждой переменной с нечетным индексом, а Боб – каждой переменной с четным индексом. Алиса хочет получить результат выражения True, а Боб желает получить False. Если предположить, что Алиса и Боб играют безошибочно, что кто выиграет? Неудивительно, что большинство игр с двумя участниками²⁸, таких как крестики-нолики, реверси, шашки, го, шахматы и калах, – или более точно: соответствующие обобщения этих игр постоянного размера до досок (игровых полей) произвольного размера, – принадлежит к PSPACE-трудным задачам.

Другая каноническая PSPACE-трудная задача – NFA totality: если задан недетерминированный конечный автомат (НКА) M на некотором алфавите Σ , то принимает ли M каждую строку в Σ^* ? Тесно связанные задачи равенства НКА (принимают ли два заданные НКА один и тот же язык?) и минимизации НКА (найти наименьший НКА, который принимает тот

²⁸ Хороший (но неизбежно устаревающий) обзор известных результатов вычислительной сложности игр и головоломок см. в монографии Эрика Демейна (Erik Demaine) и Роберта Херна (Robert Hearn) «Games, Puzzles, and Computation» (CRC Press, 2009).

же язык, что и заданный НКА) также являются PSPACE-трудными, как и соответствующие вопросы о регулярных выражениях. (Соответствующие вопросы о детерминированных конечных автоматах в действительности разрешимы за полиномиальное время.)

Экспоненциальное время

Следующий значительно больший класс сложности *EXP* (также называемый EXPTIME) – это множество задач на принятие решений, которые могут быть решены за экспоненциальное время, т. е. с использованием не более 2^{nc} шагов при некоторой константе $c > 0$. Каждая задача в PSPACE (следовательно, в NP (следовательно, в P)) также находится в EXP. В общем, все верят, что $PSPACE \subsetneq EXP$, но никто не может доказать даже, что $NP \neq EXP$. Задача Π является *EXP-трудной*, если для любой задачи Π' , которую можно решить за экспоненциальное время, существует множественная редукция, выполняемая за полиномиальное время, от Π' к Π . Если любая EXP-трудная задача находится в PSPACE, то $EXP = PSPACE$. Аналогично если любая EXP-трудная задача находится в NP, то $EXP = NP$. Мы точно знаем, что $P \neq EXP$, в частности, ни одна из EXP-трудных задач не находится в P.

Естественные обобщения многих интересных игр с двумя участниками, таких как шашки, шахматы, калах и го, действительно являются EXP-трудными задачами. Граница между PSPACE-полными и EXP-трудными играми весьма тонка. Например, существует три способа сыграть вничью в шахматах (на стандартной доске 8×8): пат (ходящий игрок не находится под шахом, но не имеет допустимых ходов), повторение одной позиции на доске три раза или в течение 50 последних ходов обеих сторон не было сделано ни одного взятия и ни одного хода пешкой. Обобщение $n \times n$ игры в шахматы либо находится в PSPACE, либо является EXP-трудной задачей в зависимости от того, как мы обобщаем эти правила. Если мы объявляем вничью после (скажем) n^3 ходов без взятия, то каждая партия обязательно должна завершаться после полиномиального числа ходов, поэтому мы можем имитировать все возможные партии, начиная с любой заданной позиции, используя только полиномиальное пространство. С другой стороны, если мы полностью игнорируем правило ходов без взятия, то итоговая партия может завершиться за экспоненциальное число ходов, поэтому не существует очевидного способа определения повторяющейся позиции с использованием только полиномиального пространства; разумеется, эта версия шахмат $n \times n$ является EXP-трудной задачей.

Все выше и выше!

Само собой, даже экспоненциальное время не завершает это повествование. NEXP – это класс задач на принятие решений, которые можно решить за недетерминированное экспоненциальное время. Равнозначно: задача на принятие решений находится в NEXP, если и только если для каждого варианта Yes существует доказательство этого факта, которое можно про-

верить за экспоненциальное время. EXPSPACE – это множество задач на принятие решений, которые можно решить, используя экспоненциальное пространство. Даже эти классы большей сложности содержат трудные задачи; например, если мы добавим оператор пересечения $\cap$ в синтаксис регулярных выражений, то определение, описывают ли два таких выражения один и тот же язык, становится EXPSPACE-трудной задачей. Кроме EXPSPACE, существуют классы сложности с удвоенными экспоненциальными границами ресурсов (EEXP, NEEEXP и EEXPSPACE), затем с утроенными экспоненциальными границами ресурсов (EEEXP, NEEEXP и EEEXPSPACE) и т. д. до бесконечности.

Все эти классы сложности можно упорядочить по вложенности:

$$P \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq NEXP \subseteq EXPSPACE \subseteq EEXP \subseteq NEEEXP \subseteq \dots$$

Большинство теоретиков в области сложности вычислений твердо уверено, что каждое вложение в этой последовательности является строгим, т. е. среди этих классов сложности нет двух равных. Но самый строгий результат, который был доказан, – это тот факт, что каждый класс в этой последовательности непременно содержится в классе, расположенном на три шага позже в той же последовательности. Например, мы располагаем доказательствами, что $P \neq EXP$ и $PSPACE \neq EXPSPACE$, но не $P \neq PSPACE$ и не $NP \neq EXP$. Пределом этой последовательности классов монотонно возрастающей сложности является класс ELEMENTARY задач на принятие решений, которые могут быть решены с использованием времени или пространства, ограниченного функцией формы $2^{\uparrow^k}$ для некоторого постоянного целого числа k , где

$$2^{\uparrow^k} := \begin{cases} n & , \text{ если } k = 0, \\ 2^{2^{\uparrow^{k-1}} n} & \text{ иначе.} \end{cases}$$

Например, $2^{\uparrow^1} n = 2^n$ и $2^{\uparrow^2} n = 2^{2^n}$.

Возможно, возникает соблазн сделать предположение о том, что каждая естественная решаемая задача может быть решена за элементарное время, но в действительности это предположение некорректно. Рассмотрим обобщенные регулярные выражения, определяемые рекурсивно объединяемыми (возможно, пустыми) строками на некотором конечном алфавите с помощью операций конкатенации (сцепления) (xy), объединения ($x + y$), замыкания Клини (x^*) и отрицания ($\bar{x}$). Например, обобщенное регулярное выражение $(0 + 1)^* 00(0 + 1)^*$ представляет множество строк в алфавите $\{0, 1\}^*$, которые не содержат двух символов 0 подряд. Можно алгоритмически определить, описывают ли два обобщенных регулярных выражения идентичные языки, с помощью рекурсивного преобразования каждого выражения в равнозначный НКА с последующим преобразованием каждого НКА в детерминированный конечный автомат (ДКА), а затем выполнить минимизацию ДКА. Но время выполнения этого алгоритма имеет не элементарную границу $2^{\uparrow^{(n)}} 2$, интуитивно потому, что каждый

уровень рекурсивного отрицания может экспоненциально увеличивать число состояний. В действительности Ларри Стокмайер (Larry Stockmeyer) доказал в 1974 г., эта задача не может быть решена за чисто элементарное время, даже если мы запретим замыкание Клини.

Упражнения

1. (a) Описать и проанализировать алгоритм решения задачи Partition за время $O(nM)$, где n – размер входного множества, а M – сумма абсолютных значений его элементов.
(b) Почему этот алгоритм не предполагает, что $P = NP$?
2. Рассмотрим следующую задачу под названием BoxDepth: если задано множество n выровненных по осям прямоугольников на плоскости, то какую величину имеет наибольшее подмножество этих прямоугольников, содержащих общую точку?
 - (a) Описать редукцию, выполняемую за полиномиальное время, от BoxDepth к MaxClique.
 - (b) Описать и проанализировать алгоритм, выполняемый за полиномиальное время, для решения BoxDepth. [Подсказка: время выполнения $O(n^5)$ должно быть легко достижимым, но возможно и время выполнения $O(n \log n)$.]
 - (c) Почему эти два результата не предполагают, что $P = NP$?
3. Логическая формула находится в дизъюнктивной нормальной форме (ДНФ) (disjunctive normal form – DNF), если она состоит из дизъюнкции (OR) или нескольких членов, каждый из которых является конъюнкцией (AND) одного или нескольких литералов. Например, формула

$$(\bar{x} \wedge y \wedge \bar{z}) \vee (y \wedge z) \vee (x \wedge \bar{y} \wedge \bar{z})$$

находится в дизъюнктивной нормальной форме. Задача DNF-Sat при заданной логической формуле в дизъюнктивной нормальной форме спрашивает, является ли эта формула выполнимой.

- (a) Описать алгоритм, выполняемый за полиномиальное время, для решения задачи DNF-Sat.
- (b) Какая ошибка содержится в приведенном ниже доказательстве $P = NP$?

Предположим, что задана логическая формула в конъюнктивной нормальной форме с компонентами, содержащими не более трех литералов, и мы хотим узнать, является ли она выполнимой. Мы можем использовать распределительный закон для создания равнозначной формулы в дизъюнктивной нормальной форме. Например:

$$(x \vee y \vee \bar{z}) \wedge (\bar{x} \vee \bar{y}) \Leftrightarrow (x \wedge \bar{y}) \vee (y \wedge \bar{x}) \vee (\bar{z} \wedge \bar{x}) \vee (\bar{z} \wedge \bar{y}).$$

Теперь мы можем использовать алгоритм из части (а) для определения за полиномиальное время, является ли полученная ДНФ-формула выполнимой. Мы только что решили задачу 3Sat за полиномиальное время. Поскольку 3Sat является NP-трудной задачей, мы неизбежно приходим к заключению, что $P = NP$!

4. В задаче AllOrNothing3Sat спрашивается при заданной логической формуле 3CNF, существует ли присваивание переменным такое, что каждая компонента содержит либо три литерала True, либо три литерала False.
 - (а) Описать алгоритм, выполняемый за полиномиальное время, для решения задачи AllOrNothing3Sat.
 - (b) Но 3Sat является NP-трудной задачей. Почему существование этого алгоритма не доказывает, что $P = NP$?
5. (а) Предположим, что вам дали волшебный черный ящик, который может определить за полиномиальное время при заданном произвольном взвешенном графе G длину кратчайшего гамильтонова цикла в G . Описать и проанализировать алгоритм, выполняемый за полиномиальное время, который при заданном произвольном взвешенном графе G вычисляет кратчайший гамильтонов цикл в G , используя волшебный черный ящик как подпрограмму.
- (b) Предположим, что вам дали волшебный черный ящик, который может определить за полиномиальное время при заданном произвольном взвешенном графе G число вершин в максимальном полном подграфе G . Описать и проанализировать алгоритм, выполняемый за полиномиальное время, который при заданном произвольном взвешенном графе G вычисляет полный подграф максимального размера в G , используя волшебный черный ящик как подпрограмму.
- (c) Предположим, что вам дали волшебный черный ящик, который может определить за полиномиальное время при заданном произвольном взвешенном графе G , можно ли раскрасить G в три цвета. Описать и проанализировать алгоритм, выполняемый за полиномиальное время, который вычисляет корректную раскраску в три цвета заданного графа G или корректно сообщает, что такой раскраски не существует, используя волшебный черный ящик как подпрограмму. [Подсказка: входными данными для этого волшебного черного ящика является граф. Только граф. Вершины и ребра. Ничего другого.]
- (d) Предположим, что вам дали волшебный черный ящик, который может определить за полиномиальное время при заданной

произвольной логической формуле Φ , является ли Φ вычислимой. Описать и проанализировать алгоритм, выполняемый за полиномиальное время, который вычисляет выполнимое присваивание для заданной логической формулы или корректно сообщает, что такого присваивания не существует, используя волшебный черный ящик как подпрограмму.

- (e) Предположим, что вам дали волшебный черный ящик, который может определить за полиномиальное время при заданном произвольном множестве положительных целых чисел X , можно ли разделить X на два множества A и B таких, что $\sum A = \sum B$. Описать и проанализировать алгоритм, выполняемый за полиномиальное время, который вычисляет равные части заданного множества положительных целых чисел или корректно сообщает, что такого разделения не существует, используя волшебный черный ящик как подпрограмму.

- ♣♥(f) Предположим, что вам дали волшебный черный ящик, который может определить за полиномиальное время при заданном произвольном обобщенном регулярном выражении R (которое определено в подразделе непосредственно перед разделом «Упражнения»), соответствует ли R любой строке. Описать и проанализировать алгоритм, выполняемый за полиномиальное время, который находит одну строку, соответствующую заданному обобщенному регулярному выражению или корректно сообщает, что такой строки не существует, используя волшебный черный ящик как подпрограмму.

6. Здесь кое-что особенное о числе 3.

- (a) Описать и проанализировать алгоритм, выполняемый за полиномиальное время, для решения задачи 2Partition. Если задано множество S из $2n$ положительных целых чисел, то ваш алгоритм должен определить за полиномиальное время, можно ли разделить элементы множества S на n непересекающихся пар, суммы которых равны.
- (b) Описать и проанализировать алгоритм, выполняемый за полиномиальное время, для решения задачи 2Color. Если задан ненаправленный граф G , то ваш алгоритм должен определить за полиномиальное время, существует ли для G корректная раскраска, использующая только два цвета.
- (c) Описать и проанализировать алгоритм, выполняемый за полиномиальное время, для решения задачи 2Sat. Если задана логическая формула Φ в конъюнктивной нормальной форме с ровно двумя литералами в каждой компоненте, то ваш алгоритм должен определить за полиномиальное время, существует ли для Φ выполнимое присваивание. [Подсказка: эта задача тесно связана с темами, рассмотренными в предыдущей главе.]

7. А здесь нет ничего особенного о числе 3.
 - (a) Задача 12Partition определяется следующим образом: задано множество S из $12n$ положительных целых чисел. Определить, можно ли разделить элементы S на n подмножеств по 12 элементов каждое, суммы которых равны. Доказать, что 12Partition является NP-трудной задачей. [Подсказка: выполните редукцию от 3Partition. Это может оказаться более простым, чем рассмотрение мультимножеств сначала.]
 - (b) Задача 12Color определяется следующим образом: задан ненаправленный граф G . Определить, можно ли раскрасить каждую вершину в один из 12 цветов так, чтобы каждое ребро касалось двух различных цветов. Доказать, что 12Color является NP-трудной задачей. [Подсказка: выполните редукцию от 3Color.]
 - (c) Задача 12Sat определяется следующим образом: задана логическая формула Φ в конъюнктивной нормальной форме ровно с двенадцатью литералами в каждой компоненте. Определить, существует ли для Φ выполнимое присваивание. Доказать, что 12Sat является NP-трудной задачей. [Подсказка: выполните редукцию от 3Sat.]
8. Существуют две версии задачи поиска гамильтонова цикла – для направленных и ненаправленных графов. Выше в этой главе вы можете найти два доказательства того, что задача поиска направленного гамильтонова цикла является NP-трудной.
 - (a) Описать редукцию, выполняемую за полиномиальное время, от задачи поиска ненаправленного гамильтонова цикла к задаче поиска направленного гамильтонова цикла. Доказать, что эта редукция корректна.
 - (b) Описать редукцию, выполняемую за полиномиальное время, от задачи поиска направленного гамильтонова цикла к задаче поиска ненаправленного гамильтонова цикла. Доказать, что эта редукция корректна.
 - (c) Какая из этих двух редукций предполагает, что задача поиска ненаправленного гамильтонова цикла является NP-трудной?
9.
 - (a) Описать редукцию, выполняемую за полиномиальное время, от задачи UndirectedHamiltonianCycle к задаче DirectedHamiltonianCycle.
 - (b) Описать редукцию, выполняемую за полиномиальное время, от задачи DirectedHamiltonianCycle к задаче UndirectedHamiltonianCycle.
10.
 - (a) Описать редукцию, выполняемую за полиномиальное время, от задачи HamiltonianPath к задаче HamiltonianCycle.
 - (b) Описать редукцию, выполняемую за полиномиальное время, от задачи HamiltonianCycle к задаче HamiltonianPath. [Подсказка: ре-

дукция, выполняемая за полиномиальное время, может вызывать подпрограмму черного ящика более одного раза, но это не обязательно.]

11. Рассмотрим следующие хитроумные варианты задачи CNFSat. Для каждой задачи входными данными является логическая формула Φ в конъюнктивной нормальной форме, а целью – определение, существует ли для Φ выполнимое присваивание.
 - (а) Предположим, что каждая компонента Φ содержит не более трех литералов и каждая переменная присутствует не более чем в трех компонентах. Доказать, что этот вариант CNFSat является NP-трудной задачей.
 - (б) Предположим, что каждая компонента Φ содержит ровно три литерала и каждая переменная присутствует не более чем в четырех компонентах. Доказать, что этот вариант CNFSat является NP-трудной задачей. [Подсказка: сначала решите часть (а).]
 - ♥(с) Предположим, что каждая компонента Φ содержит любое число трех литералов, но каждая переменная присутствует не более чем в двух компонентах. Описать алгоритм, выполняемый за полиномиальное время, для решения этого варианта задачи CNFSat.
 - ♥(д) Предположим, что каждая компонента Φ содержит ровно три литерала и каждая переменная присутствует не более чем в трех компонентах. Доказать, что Φ обязательно должна быть выполнимой формулой. (А этот вариант задачи 3Sat абсолютно тривиален.)
12. (а) Доказать, что задача NotAllEqual3Sat является NP-трудной.
 (б) Доказать, что задача 1-in-3Sat является NP-трудной.
13. Логическая формула в конъюнктивной нормальной форме с исключающим или (exclusive-or conjunctive normal form – XCNF) – это конъюнкция (AND) нескольких компонент, каждая из которых содержит оператор «исключающее или» (XOR), разделяющий несколько литералов, т. е. компонента истинна, если и только если она содержит нечетное число истинных литералов. Задача XCNF-Sat спрашивает, является ли заданная формула XCNF выполнимой. Описать алгоритм, выполняемый за полиномиальное время, для решения задачи XCNF-Sat или доказать, что XCNF-Sat является NP-трудной задачей. [Подсказка: не пытайтесь выполнить оба задания.]
- ♥14. Рассмотрим следующий вариант задачи 3Sat под названием Majority3Sat. Как и для 3Sat, входными данными для Majority3Sat является логическая формула Φ в конъюнктивной нормальной форме с ровно тремя литералами в каждой компоненте. Задача Majority3Sat

спрашивает, существует ли присваивание переменным Φ такое, что каждая компонента содержит как минимум два литерала True.

Описать алгоритм, который решает задачу Majority3Sat за полиномиальное время, или доказать, что Majority3Sat является NP-трудной задачей.

- ♠♥15. Для любого подмножества $X \subseteq \{0, 1, 2, 3\}$ рассмотрим следующую задачу, которую я буду называть X-3Sat. Входными данными является логическая формула Φ в конъюнктивной нормальной форме с ровно тремя литералами в каждой компоненте. Задача состоит в определении, существует ли присваивание переменным Φ такое, что в каждой компоненте Φ число литералов True находится в множестве X . Например:

- $\{1, 2, 3\}$ -3Sat – это стандартная задача 3Sat;
- $\{0, 3\}$ -3Sat – это та же задача, что и AllOrNothing3Sat (см. упражнение 4);
- $\{1, 2\}$ -3Sat обычно называют NotAllEqual3Sat (см. упражнение 12(a));
- $\{1\}$ -3Sat обычно называют 1-in-3Sat (см. упражнение 12(b));
- $\{1, 3\}$ -3Sat обычно называют XCNF-3Sat (см. упражнение 13);
- $\{2, 3\}$ -3Sat обычно называют Majority3Sat (см. упражнение 14).

Привести полный список всех подмножеств $X \subseteq \{0, 1, 2, 3\}$ такой, что задача X-3Sat является решаемой за полиномиальное время, предполагая, что $P \neq NP$. [Подсказка: не приводите 16 различных аргументов.]

16. Доказать, что описанные ниже задачи являются NP-трудными.
- (a) Задан ненаправленный граф G . Содержит ли G простой путь, который посещает все вершины, за исключением 17?
 - (b) Задан ненаправленный граф G . Содержит ли G остовное дерево, в котором каждый узел имеет степень не более 23?
 - (c) Задан ненаправленный граф G . Содержит ли G остовное дерево, в котором не более 42 листьев?
 - (d) Задан ненаправленный граф $G = (V, E)$. Каков размер максимального подмножества вершин $S \subseteq V$ такого, что не более 374 ребер в E имеют обе конечные точки в S ?
 - (e) Задан ненаправленный граф $G = (V, E)$. Каков размер максимального подмножества вершин $S \subseteq V$ такого, что каждая вершина в S имеет не более 473 соседей в S ?
 - (f) Задан ненаправленный граф G . Можно ли раскрасить вершины в G тремя различными цветами так, что не более 31 337 ребер имеют обе конечные точки одного цвета?
17. Доказать, что описанные ниже варианты задачи поиска минимального остовного дерева являются NP-трудными.

- (a) Задан граф G . Вычислить остовное дерево максимального диаметра в G . (Диаметр дерева T равен длине максимального пути в T .)
 - (b) Задан граф G с взвешенными ребрами. Вычислить остовное дерево поиска в глубину минимального веса в графе G .
 - (c) Задан граф G с взвешенными ребрами и подмножество S вершин графа G . Вычислить остовное дерево минимального веса, все листья которого находятся в S .
 - (d) Задан граф G с взвешенными ребрами и целое число ℓ . Вычислить остовное дерево минимального веса с не более чем ℓ листьями.
 - (e) Задан граф G с взвешенными ребрами и целое число Δ . Вычислить остовное дерево минимального веса, в котором каждый узел имеет степень не более Δ .
18. (a) Используя гаджет, показанный на рис. 12.29(a), доказать, что решение, определяющее, можно ли раскрасить заданный плоский граф в три цвета, является NP-трудной задачей. [Подсказка: показать, что гаджет на рис. 12.29(a) можно раскрасить в три цвета, а затем заменить все пересечения в плоском графе, встраивая соответствующим образом этот гаджет.]

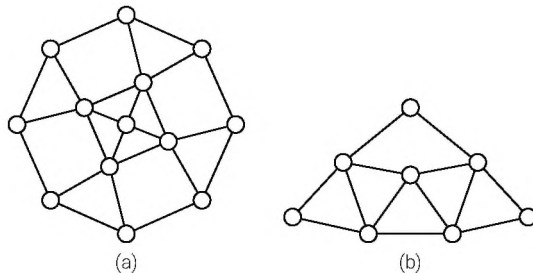


Рис. 12.29. (a) Гаджет для раскраски в три цвета плоского графа.

- (b) Гаджет для раскраски в три цвета плоского графа с максимальной степенью 4
- (b) Используя часть (a) и гаджет, показанный на рис. 12.29(b), доказать, что решение, определяющее, можно ли раскрасить заданный плоский граф с максимальной степенью 4 в три цвета, является NP-трудной задачей. [Подсказка: заменить каждую вершину со степенью больше 4 набором гаджетов, соединенных так, что ни одна из степеней не превышает четыре.]
19. Доказать, что задача PlanarCircuitSat является NP-трудной. [Подсказка: создайте гаджет для пересекающихся проводов.]
20. (a) Описать редукцию, выполняемую за полиномиальное время, от 3Sat к 4Sat.

- (b) Описать редукцию, выполняемую за полиномиальное время, от 4Sat к 3Sat.
- ♥21. Описать прямую редукцию, выполняемую за полиномиальное время, от 4Color к 3Color. (Это гораздо труднее, чем редукция в обратном направлении.)
22. Фишка домино – это прямоугольник 1×2 , разделенный на два квадрата, каждый из которых помечен некоторым целым числом²⁹. В разрешенном расположении домино фишки размещаются в линию, соединяясь по коротким сторонам так, чтобы числа на смежных квадратах совпадали, как показано на рис. 12.30.

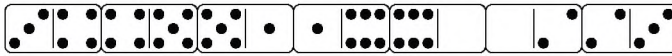


Рис. 12.30. Допустимое расположение фишек домино, в котором каждое целое число от 0 до 6 встречается дважды

Для каждой из описанных ниже задач описать алгоритм, выполняемый за полиномиальное время, или доказать, что задача является NP-трудной.

- (a) Задано произвольное мультимножество D фишек домино. Существует ли допустимое расположение всех фишек домино в D ?
- (b) Задано произвольное мультимножество D фишек домино. Существует ли допустимое расположение фишек домино в D , в котором каждое целое число от 1 до n встречается ровно два раза?
- ♥(c) Задано произвольное мультимножество D фишек домино. Каково максимальное число фишек домино, которое мы можем взять из D , чтобы составить допустимое расположение?
23. Pebbling – игра из семейства «солитер» на ненаправленном графе G , где в каждой вершине находится ноль или более камешков (pebbles). Один ход камешками состоит из удаления двух камешков из вершины v и добавление одного камешка в произвольную вершину, соседнюю с v . (Очевидно, что перед ходом в вершине v непременно должно находиться не менее двух камешков.) Задача PebbleDestruction спрашивает при заданном графе $G = (V, E)$ и счетчике камешков $p(v)$ для каждой вершины v , существует ли последовательность ходов камешками, которая удаляет все камешки, кроме одного. Доказать, что задача PebbleDestruction является NP-трудной.

²⁹ Эти целые числа обычно представлены точками (чаще всего круглыми лунками) точно так же, как на игральных кубиках. На стандартной фишке домино на каждой стороне нанесено число точек от 0 до 6, хотя встречаются комплекты с числом до 9 или даже 12 точек на каждой стороне, поэтому мы допускаем произвольные целочисленные метки. Стандартный комплект домино содержит ровно по одной фишке для каждой возможной неупорядоченной пары меток, но мы не будем предполагать, что входные данные для описанных здесь задач обладают этим свойством.

24. Напомню, что раскраска в пять цветов графа G – это функция, которая присваивает каждой вершине G «цвет» из множества $\{0, 1, 2, 3, 4\}$ такой, что для любого ребра uv вершинам u и v присвоены различные «цвета». Раскраска в пять цветов является аккуратной, если цвета, присвоенные смежным вершинам, являются не только различными, но и отличаются более чем на 1 (mod 5). Доказать, что определение существования для заданного графа аккуратной раскраски в пять цветов является NP-трудной задачей. [Подсказка: выполните редукцию от стандартной задачи 5Color.]

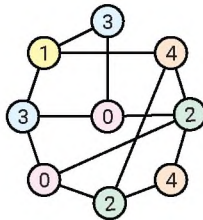


Рис. 12.31. Аккуратная раскраска в пять цветов

25. (a) Подмножество вершин S в ненаправленном графе G является полунезависимым (half-independent), если каждая вершина в S является смежной не более чем с одной другой вершиной в S . Доказать, что поиск размера максимального полунезависимого множества вершин в заданном ненаправленном графе является NP-трудной задачей.
- (b) Подмножество вершин S в ненаправленном графе G является вроде бы независимым (sort-of-independent), если каждая вершина в S является смежной не более чем с 374 вершинами в S . Доказать, что поиск размера максимального вроде бы независимого множества вершин в заданном ненаправленном графе является NP-трудной задачей.
- (c) Подмножество вершин S в ненаправленном графе G является почти независимым (almost independent), если не более 374 ребер в G имеют конечные точки, находящиеся в S . Доказать, что поиск размера максимального почти независимого множества вершин в заданном ненаправленном графе является NP-трудной задачей.
26. Пусть $G = (V, E)$ – граф. Доминирующее множество (dominating set) в G – это подмножество вершин S такое, что каждая вершина в G либо находится в S , либо является смежной с вершиной из S . Задача DominatingSet спрашивает при заданном графе G и целом числе k как входных данных, содержит ли G доминирующее множество размером k . Доказать, что эта задача является NP-трудной.

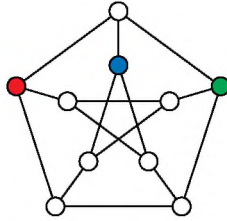


Рис. 12.32. Доминирующее множество размером 3 в графе Петерсена

27. Подмножество вершин S в ненаправленном графе G является нетриангулированным (triangle-free), если для каждой тройки вершин $u, v, w \in S$ как минимум одно из трех ребер uv, uw, vw отсутствует в G . Доказать, что поиск размера максимального нетриангулированного подмножества вершин в заданном ненаправленном графе является NP-трудной задачей.

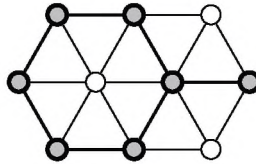


Рис. 12.33. Нетриангулированное подмножество из семи вершин.
Это не самое большое такое подмножество в этом графе

28. Задача RectangleTiling определяется следующим образом: задан один большой прямоугольник и несколько прямоугольников меньшего размера. Определить, можно ли разместить прямоугольники меньшего размера внутри большого прямоугольника без пустых промежутков и перекрытий.

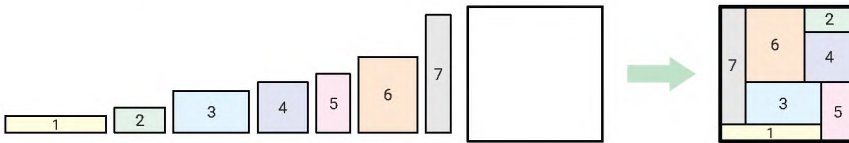


Рис. 12.34. Положительный экземпляр задачи RectangleTiling

- (а) Доказать, что задача RectangleTiling является NP-трудной.
(б) Доказать, что задача RectangleTiling является строго NP-трудной.
29. (а) Подмножество вершин B в графе G является множеством Бурра (Burr set), если удаление любой вершины в B из графа G оставляет подграф, не содержащий гамильтонов путь. Доказать, что поиск наименьшего множества Бурра в заданном графе является NP-трудной задачей.

- (b) Подмножество вершин S в графе G является множеством Скайлера (Schuyler set), если удаление любой вершины в S из графа G оставляет подграф, содержащий гамильтонов путь. Доказать, что поиск наименьшего множества Скайлера в заданном графе является NP-трудной задачей.
30. (a) Тонийский путь (tonian path) в графе G – это путь, проходящий как минимум через половину вершин в G . Показать, что определение существования тонийского пути в графе является NP-трудной задачей.
- (b) Тонийский цикл (tonian cycle) в графе G – это цикл, проходящий как минимум через половину вершин в G . Показать, что определение существования тонийского цикла в графе является NP-трудной задачей. [Подсказка: используйте часть (a). Или не используйте.]
31. Пусть G – ненаправленный граф с взвешенными ребрами. Тяжелый гамильтонов цикл – это цикл C , который проходит через каждую вершину G ровно один раз так, что суммарный вес ребер в C больше половины суммарного веса всех ребер в G . Доказать, что определение существования в графе тяжелого гамильтонова цикла является NP-трудной задачей.

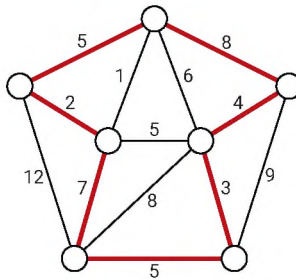
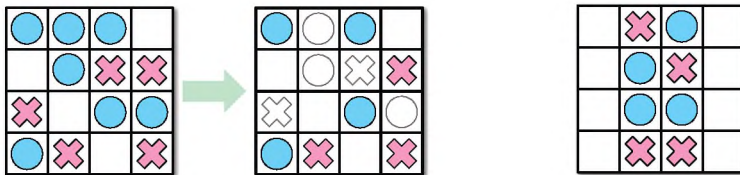


Рис. 12.35. Тяжелый гамильтонов цикл. Суммарный вес этого цикла равен 34, а граф имеет суммарный вес 67

32. Для каждой из приведенных ниже задач описать алгоритм, выполняемый за полиномиальное время, или доказать, что задача является NP-трудной.
- (a) Двойной эйлеров обход в ненаправленном графе G – это замкнутое блуждание, проходящее каждое ребро в G ровно два раза. Задан граф G . Содержит ли G двойной эйлеров обход?
- (b) Двойной гамильтонов обход в ненаправленном графе G – это замкнутое блуждание, которое посещает каждую вершину в G ровно два раза. Задан граф G . Содержит ли G двойной гамильтонов обход?

- (с) Двойной гамильтонов контур в ненаправленном графе G – это замкнутое блуждание, которое посещает каждую вершину в G ровно два раза и проходит каждое ребро в G не более одного раза. Задан граф G . Содержит ли G двойной гамильтонов обход?
- (d) Тройной эйлеров обход в ненаправленном графе G – это замкнутое блуждание, проходящее каждое ребро в G ровно три раза. Задан граф G . Содержит ли G тройной эйлеров обход?
- (е) Тройной гамильтонов обход в ненаправленном графе G – это замкнутое блуждание, которое посещает каждую вершину в G ровно три раза. Задан граф G . Содержит ли G двойной гамильтонов обход?
33. В этом упражнении вам предлагается доказать, что конкретная редукция от задачи VertexCover к задаче SteinerTree является корректной. Предположим, что нам необходимо найти наименьшее вершинное покрытие в заданном ненаправленном графе $G = (V, E)$. Мы создаем новый граф $H = (V', E')$, как показано ниже:
- $V' = V \cup E \cup \{z\}$;
 - $E' = \{ve \mid v \in V - \text{конечная точка } e \in E\} \cup \{vz \mid v \in V\}$.
- Равнозначно: мы создаем H , подразделяя каждое ребро в G с помощью новой вершины, а затем соединяя все исходные вершины в G с этой новой апексной вершиной z .
- Доказать, что G содержит вершинное покрытие размером k , если и только если существует поддерево в H с $k + |E| + 1$ вершинами, которое содержит каждую вершину в $E \cup \{z\}$.
34. Рассмотрим следующую игру типа «солитер». Головоломка состоит из поля $n \times m$ клеток, где каждая клетка может быть пустой или занятой красным или синим камнем. Цель головоломки – удалить некоторые из заданных камней так, чтобы оставшиеся камни соответствовали двум условиям: (1) каждая строка содержит хотя бы один камень и (2) ни один столбец не содержит камни обоих цветов. Для некоторых начальных конфигураций камней достижение этой цели невозможно.



Разрешимая головоломка и одно из ее многих решений Неразрешимая головоломка

Рис. 12.36. Головоломка с камнями двух цветов

Доказать, что определение разрешимости этой головоломки при заданной начальной конфигурации красных и синих камней является NP-трудной задачей.

35. В каждой из описанных ниже игр применяется игровое поле из $n \times m$ клеток, где каждая клетка может быть пустой или занятой камнем. За один ход вы можете удалить все камни в произвольном столбце.
 - (а) Доказать, что поиск наименьшего подмножества столбцов, которые можно очистить так, чтобы в каждой строке поля осталось не более одного камня, является NP-трудной задачей.
 - (б) Доказать, что поиск наибольшего подмножества столбцов, которые можно очистить так, чтобы в каждой строке поля остался хотя бы один камень, является NP-трудной задачей.
 - ♥(с) Доказать, что определение возможности очистки любого подмножества столбцов так, чтобы в каждой строке игрового поля остался ровно один камень, является NP-трудной задачей.
36. Джефф пытается сделать счастливыми своих студентов. В начале курса он выдал им опросный лист, в котором перечислен ряд стратегий курса в областях, которыми он свободно владеет. Каждому студенту предлагается ответить по каждой возможной стратегии курса одним из вариантов: «категорически поддерживаю», «в основном нейтрален» или «категорически возражаю». Каждый студент может ответить «всячески поддерживаю» или «категорически возражаю» не более чем на пять вопросов. Поскольку студенты Джеффа весьма разумны, каждый студент счастлив, если (но только если) он или она предпочитает как минимум одно из категорически поддерживаемых предложений стратегии. Описать алгоритм, выполняемый за полиномиальное время, для установки стратегии курса с максимизацией числа счастливых студентов или показать, что эта задача является NP-трудной.
37. Вы работаете в должности хореографа в местном музыкальном театре, и пришло время постановки финальной сцены с большим числом исполнителей, завершающей спектакль. («Трамвай!») Вы решили, что каждый из n приглашенных участников шоу будет размещен в длинном ряду, когда завершается песня, – все разводит руки в стороны и показывают свои наилучшие танцевальные па. Режиссер объявил, что во время финальной феерии каждый участник обязательно должен поднять руки вверх или опустить их вниз. Ваша задача – определить, кто поднимает, а кто опускает руки. Кроме того, режиссер также дал вам список композиций, которые огорчают его тонкую артистическую натуру. Каждая запрещенная ком-

позиция – это подмножество участников, положения рук которых образуют пару, например: «Мардж не должна поднимать руки вверх, когда Нед, Апу и Смитерс опускают руки вниз».

Доказать, что поиск приемлемой композиции положений рук является NP-трудной задачей.

38. В очередной раз вы на вечеринке. Один из гостей предлагает всем сыграть в Three-Way Mumbletypeg, игру, требующую определенного умения и ловкости, для которой нужны три команды и нож. Официальные правила Three-Way Mumbletypeg (утвержденные консулом Священной Римской империи Three-Way Mumbletypeg в 1625 г.) требуют, чтобы (1) в каждой команде был хотя бы один человек, (2) любые два человека в одной команде обязательно должны знать друг друга и (3) каждый следящий за игрой обязательно должен быть членом одной из трех команд. Разумеется, это будет действительно веселая вечеринка, никто не захочет уходить. На вечеринке будет несколько пар людей, которые не знают друг друга. Хозяин вечеринки, слышавший захватывающие истории о вашем мастерстве во всем, что касается алгоритмов, передал вам список пар завсегдатаев, знающих друг друга, и попросил сформировать команды, пока он натирает нож.

Описать и проанализировать алгоритм, выполняемый за полиномиальное время, который определяет, можно ли разделить завсегдатаев вечеринки на три команды по правилам Three-Way Mumbletypeg, или доказать, что это NP-трудная задача.

39. Вечеринка, на которой вы присутствуете, проходит великолепно, но наступило время встать в позицию для алгоритмического марша (The Algorithm March (アルゴリズムこうしん)). Этот танец был придуман японским дуэтом комиков Ицумо Кококара (Itsumo Kokokara (いつもここから)) для детского телевизионного шоу PythagoraSwitch (ピタゴラスイッチ). Алгоритмический марш исполняется колонной людей, каждый участник в колонне начинает специальную последовательность перемещений на одно движение позже, чем участник, стоящий прямо перед ним. Таким образом, этот марш представляет собой танец, аналогичный музыкальному туру или канону, такому как «Row Row Row Your Boat» или «Frère Jacques».

Правила этикета требуют, чтобы каждый участник марша непременно знал человека, стоящего в колонне прямо перед ним, поскольку малейшая ошибка приводит к ужасной путанице среди незнакомцев. Предположим, что вам дали полный список людей на вечеринке, которые знают друг друга. Доказать, что определение наибольшего числа завсегдатаев, которые могут участвовать в Алгоритмическом марше, является NP-трудной задачей. Вы можете предположить без потери для обобщения, что на этой вечеринке нет ниндзя.

♣40. Доказать, что описанные ниже задачи о недетерминированных конечных автоматах (НКА) и регулярных выражениях являются NP-трудными.

- (a) Задан НКА M на алфавите $\Sigma = \{0, 1\}$. Существует ли строка в Σ^* , которую M не принимает?
- (b) Задан ациклический НКА M на алфавите $\Sigma = \{0, 1\}$. Какова длина кратчайшей строки в Σ^* , которую M не принимает?
- (c) Задано регулярное выражение R на алфавите $\Sigma = \{0, 1\}$. Существует ли строка в Σ^* , которая не соответствует R ?
- (d) Задано регулярное выражение без использования звездочек R на алфавите $\Sigma = \{0, 1\}$. Какова длина кратчайшей строки в Σ^* , которая не соответствует R ?

(В действительности задачи (a) и (c) являются PSPACE-полными; даже доказательство того, что эти задачи находятся в PSPACE, является нетривиальным.)

- ♣41. (a) Описать алгоритм, выполняемый за полиномиальное время, для решения следующей задачи: задан НКА M на алфавите $\Sigma = \{0, 1\}$. Существует ли строка в Σ^* , которую M принимает?
- (b) Описать алгоритм, выполняемый за полиномиальное время, для решения следующей задачи: задано регулярное выражение R на алфавите $\Sigma = \{0, 1\}$. Существует ли строка в Σ^* , которая соответствует R ?
- (c) Дополнением любого регулярного выражения является другое регулярное выражение. Тогда почему эти два алгоритма вместе с результатами доказательств NP-трудности задач в упражнении 40 не доказывают, что $P = NP$?

42. Харон должен перевезти n недавно умерших людей через реку Ахерон в Гадес. Некоторые пары являются непримиримыми врагами, которые не могут находиться вместе на любом берегу реки, если Харона нет рядом с ними. (Если двух врагов оставить наедине, то один из них украдет священную монету обол из рта другого, и оба навеки останутся бродить по берегам Ахерона как бесплотные духи. Давайте условимся, что это очень плохо.) Лодка Харона может перевезти не более k пассажиров за один раз, включая самого Харона, и только Харон может управлять ею.³⁰

³⁰ Эта задача является обобщением широко известной головоломки о волке, козе и капусте, первое известное упоминание о которой было обнаружено в знаменитом средневековом манускрипте «Propositiones ad Acuendos Juvenes» («Задачи для тренировки ума юношей»)

XVIII. Propositio De Homine et Capra et Lupo.

Homo quidam debebat ultra fluuium transferre lupum, capram, et fasciculum cauli. Et non potuit aliam nauem inuenire, nisi quae duos tantum ex ipsis ferre ualebat. Praeceptum itaque ei fuerat, ut omnia haec ultra illaesa omnino transferret. Dicat, qui potest, quomodo eis illaesis transire potuit?

Доказать, что определение возможности переправы Хароном всех n людей через Ахерон без ущерба (не считая того, что они уже мертвы, как вам известно) является NP-трудной задачей. Входные данные для задачи Харона состоят из целых чисел k и n и графа G с n вершинами, описывающими пары врагов. Выходными данными является значение True или False.

Пожалуйста, не записывайте ваше решение на классической латыни.

Solutio. Simili namque tenore duceremprius capramet dmitteremforis lupumet caulum. Tumdeinde uenirem, lupumque transferrem: lupoque forismissu capramnaui receptamultra reducerem; capramque foris missam caulum transueherem ultra; atque iterum remigassem, capramque assumptam ultra duxissem. Sicque faciendo facta erit remigatio salubris, absque uoragine lacerationis.

Для тех немногих читателей, у которых чтение на классической латыни вызывает небольшие затруднения, приводим перевод на русский язык:

XVIII. Задача о человеке, козе и волке.

Человеку нужно было переправить через реку волка, козу и мешок капусты. Однако он обнаружил, что его лодка могла нести вес только двух [объектов одновременно, включая человека]. И он должен был все передать целым и невредимым. Скажите, если сможете: как они сумели переправиться целыми и невредимыми?

Решение. Подобным образом [как и в предыдущей задаче] я сначала перевез бы козу, а волка и капусту оставил бы на противоположном берегу. Потом я бы переправил волка, оставил его на берегу, а козу перевез обратно. Далее я оставил бы козу и перевез капусту. Наконец, я вернулся бы и забрал козу. Таким образом, переправа завершается удачно, без какой-либо угрозы для жизни.

Наиболее вероятный автор этих «Задач» – плодovitый английский ученый VIII в. Алкуин из Йорка (Alcuin of York). Доказательство того, что именно Алкуин является автором этого трактата, в определенной степени зависит от обстоятельств, но нам известно из его переписки с Карлом Великим, что он посылал императору некоторые «простые арифметические задачи для развлечения». Большинство современных ученых уверено, что, даже если Алкуин действительно написал «Задачи», сам он не придумал все задачи, а собрал их из более ранних источников. Некоторые вещи никогда не меняются.

Предметный указатель

Если бы у нас был индексный файл, мы могли бы найти его в индексном файле в разделе «индексный файл»..

— Теган Джованка (Tegan Jovanka) [Джанет Филдинг (Janet Fielding)], «Castrovalva (Part 1)», «Доктор Кто» («Doctor Who»), сезон 19 (4 января 1982 г.)

Я начал с телефонной книги. Искать Менсу было непросто, учитывая необходимость следовать строгим правилам алфавита, которые так распространены в наши дни. Я предпочитаю более мягкую и нечеткую схему алфавита, которая позволяет уму свободно парить и «случайно натекать» на слово. В этом есть гордость. Словарь – прекрасный пример чрезмерной алфавитной упорядоченности, с его суровыми правилами и аккуратным расположением каждого словечка. Это почти отбивает у меня желание есть.

— Стив Мартин (Steve Martin), «Как я присоединился к Менсе» («How I Joined Mensa»), The New Yorker, 21 июля 1997 г.

Dicebat Bernardus Carnotensis nos esse quasi nanos gigantium humeris insidentes, ut possimus plura eis et remotiora videre, non utique proprii visus acumine, aut eminentia corporis, sed quia in altum subvehimur et extollimur magnitudine gigantea.

[Бернар Шартрский говорил, что мы подобны карликам, сидящим на плечах гигантов. Он указал, что мы видим больше и дальше, чем наши предшественники, не потому, что у нас более острое зрение или более высокий рост, а потому, что мы поднимаемся и возносимся вверх благодаря их гигантскому росту].

— Иоанн Солсберийский (John of Salisbury), «Металогик» («Metalogicon») (1159 г.), на англ. язык перевел Даниэл Д. МакГарри (Daniel D. McGarry) (1955 г.)

Секрет продуктивности в том, чтобы ранее умершие люди делали работу за вас.

— Роберт Дж. Ланг (Robert J. Lang) (2009 г.)

Символы

1-in-3Sat, задача 481

3Color, задача

редукция от 3Sat 471

3Partition, задача 482

3Sat, задача 462

редукция к 3Color 471

редукция к DirectedHamCycle 476

редукция к MaxIndSet 465, 469

редукция от CircuitSat 463, 468

APSP, all pairs shortest path 374

Assignment problem 427

В

Backtracking 102

Best-first search 256

Bitonic array 92

Blit block transfer (передача блоков) 90

Boston Pool, алгоритм 221

Breadth-first search 256

В-дерево 192

С

Candy Crush Saga 484

Checkers 485

А

Ad-hoc network 447

Alternating path 426

Checkers (шашки в США) 279
CircuitSat 454
CircuitSat, задача
 редукция к 3Sat 463, 468
CircuitSat, редукция к Sat 461
Clause gadget 472
Clause vertex 477
Clique 469
Conjunctive normal form – CNF 463
co-NP, противоположность NP 455
Cookie Clicker 484
Cook reduction 459
Cut vertex 303
Cycle cover 443

D

DAG directed acyclic graph 244
Dance Dance Revolution (танцевальная
 видеоигра) 176
Decision problem 455
Depth-first search 104, 281
Depth-first spanning tree 255
Dijkstras algorithm 257
Directed graph 242
DirectedHamCycle, задача
 редукция от 3Sat 476
DirectedHamPath, задача 478
Disjoint-path cover 431
Disjoint-set data structure 328
Disjunctive normal form – DNF 493
Divide and conquer 57
Domain transformation 54, 61
Draughts 485
Draughts (шашки в Великобритании) 279
Dynamic programming 139

E

Edge gadget 474
Edit distance 150
EDVAC (компьютер) 51
ELEMENTARY, класс задач 492
Escape problem 441
Exact3DimensionalMatching, задача 482
EXP-трудная задача 491
EXP, экспоненциальное время 491

F

Fast Fourier transform (FFT) 70
Fibonacci heap 324
Finite state automaton 247
Flood fill 260
Flow decomposition 404

G

Gadget 471
Game tree 106
Garbage collector 264
Greatest common divisor 87
Greedy algorithm 146

H

HamCycle, задача
 редукция от VertexCover 474
Hamiltonian cycle 473
Hamiltonian path 478
HittingSet, задача 482

I

IBM 51
Independent set 465
Inversion 80

K

Kane, Daniel 274
Karp reduction 459
Kd-дерево 97
Klondike, он же Solitaire (Солитер) 483

L

Logarithmic-space reductions 459
Longest increasing subsequence 121
LongestPath, задача 482
Looped tree 361

M

Many-one reduction 459
Max2Sat, задача 483
MaxClique, задача
 редукция от MaxIndSet 469

MaxCut, задача 483
 Maximum flow problem 395
 Maximum independent set 465
 MaxIndSet, задача 465
 редукция к MaxClique 469
 редукция от 3Sat 465, 469
 Median of three 56
 Memoization 137
 Mergesort 51
 Minesweeper (Сапёр) 483
 Minimum cut problem 396
 MinVertexCover, задача
 редукция от MaxIndSet 470
 Multi-terminal flow 440

N

National Resident Matching Program (NRMP) 219
 N. Claus (de Siam) (Н. Клаус (из Сиам)) 48
 NotAllEqual3Sat, задача 481
 NP-complete 456
 NP $\neq$ co-NP 456
 NP-easy 456
 NP-hard 454, 456
 weakly 481
 NP-легкая задача 456
 NP, недетерминированное полиномиальное
 время 455
 NP-полная задача 456
 NP-трудная задача 157, 161, 184, 292, 427,
 431, 454, 456
 выбор наиболее подходящей задачи
 для редукции при доказательстве 484
 правила 484
 неустойчиво 481
 список 481
 строго 481
 формальное определение 459

O

Optimal substructure 151

P

Рас-Ман (Пакман) 483
 Parallel assignment 307
 Pivot 54

Pixel 260
 Planar3Sat, задача 482
 PlanarCircuitSat, задача 481
 P $\neq$ NP 455
 Polygonal path 310
 Prefix-free code 212
 Prims algorithm 256
 Priority queue 256
 Proper k-coloring 470
 Pseudo-polynomial time 481
 PSPACE 489
 PSPACE-трудная задача 490
 P-versus-NP, нерешенная (открытая) задача 456
 P, полиномиальное время 455

Q

Quantified boolean formula – QBF 490
 Queue 255
 Quicksort 54

R

Racetrack, игра 276
 Random-access machine 458
 Recursion 46
 Recursion tree 57, 104
 Recursive brute force 113
 Reduction 45
 Reduction argument 460
 Replacement paths 362
 Residual capacity 398
 Residual graph 398
 Rooted tree 450
 Ruler function 73

S

Sat, NP-трудная задача 461
 Sat, задача 460
 Sat, редукция от CircuitSat 461
 Scrabble (игра на составление слов из букв) 177
 SetCover, задача 482
 Shortest path tree 335
 Shuffle 168
 Snakes and Ladders, настольная игра 268
 SSSP, single source shortest path 335
 Stable matching 219

SteinerTree, задача 483
 Strongly NP-hard 481
 Subsequence 120
 SubsetSum, задача 479
 редукция от VertexCover 479
 Substring 120
 Sudoku (Судоку) 483
 Super Mario Brothers (Братья Супер-Марио) 483

T

Tabula Peutingeriana, исторический пример
 графа 239
 Tetris (Тетрис) 483
 Threes/2048 484
 Tower of Hanoi 48
 Trainyard (Ж/д стрелки) 484
 Transforming certificate 468
 Transitive closure 304, 390
 Transitive reduction 304
 Traveling salesman problem 478
 Truth gadget 471
 Tuple selection 427
 Turing machine 458
 Turing reduction 459

U

Undirected graph 242
 UndirectedHamCycle, задача 478
 редукция к международным шашкам
 на доске $n \times n$ 486
 UndirectedHamPath, задача 478

V

Vankins Mile (американская игра) 181
 Variable gadget 471
 Vertex chain 475
 Vertex cover 469
 VertexCover, задача
 редукция к HamCycle 474
 редукция к SubsetSum 479

W

Weighted median 82

X

X3M, задача 482

A

Аддитивная цепочка 231
 Аддитивная цепочка для целого числа n 130
 Адельсон-Вельский, Георгий 189, 409
 Адельсон-Вельской, Георгий 132
 Аккермана обратная функция $\alpha(n)$ 328
 Алгоритм
 жадный 280
 разбиения потока 406
 Алгоритм Ярника
 поиск по первому наилучшему совпадению 324
 Аль-Сули, Аль-Адиль, Рудрата (al-Suli, al-Adli, Rudraa) 241
 Амортизированная стоимость операции 325
 Амортизированное время выполнения
 операции 328
 Андерссон, Арне (Andersson, Arne) 133, 190

B

Байер, Рудольф (Bayer, Rudolf) 132, 189
 Балансирование скобок в строке
 (задача) 193, 234
 Баланс скобок 234
 Беллмана–Калаба алгоритм 354
 Беллмана–Форда алгоритм 354
 бесконечный цикл 358
 версия Мура 356
 дополнительный параметр в рекуррентном
 выражении 358
 как алгоритм динамического
 программирования 358
 Беллмана–Шимбея алгоритм 354
 Беллман, Ричард (Bellman, Richard) 139, 354, 358
 Берж, Клод (Berge, Claude) 427
 Беспрефиксный двоичный код
 как двоичное дерево 212
 Битоническая последовательность 167
 Битоническая последовательность чисел 131
 Битонический массив 92
 Блинная сортировка 78
 оптимальное количество переворотов
 (не решенная проблема) 78
 переворот 78
 Блуждание прямоугольника, головоломка 275
 Блума–Флойда–Пратта–Ривеста–Тарьяна
 алгоритм 83, 84

Борувка Отакар (Bortvka, Otakar) 320
 Борувки алгоритм 320
 преимущества 322
 Брош Элли (Brosh, Allie) 321
 Брюнель Жорж (Brunel, Georges) 249
 Быстрая сортировка 54
 рекуррентное выражение 60
 рекурсивное дерево 60
 Быстрые преобразования Фурье (БПФ) 70

В

Вагнер Роберт (Wagner, Robert) 154
 Вайс Марк Аллен (Weiss, Mark Allen) 133, 190
 ван дер Хувен Йорис (van der Hoeven, Joris) 70
 Вариньон Пьер (Varignon, Pierre) 240
 Вейнбергер Арнольд (Weinberger, Arnold) 323, 326
 Величко В. М. 154
 Вершина компоненты 477
 Вершинное покрытие 475
 Вершинно-непересекающийся путь 424
 Взвешенное среднее значение 82
 Видрах Итки Леда (Vidrach Itky Leda) 268
 Винер, Кристиан (Wiener, Christian) 304
 Винцюк, Тарас 150, 154
 Вираханка (Virahāṅka) 139
 Вираханка (Virahāṅka) 135
 Вицгалль, Кристоф (Witzgall, Cristoph) 352
 Вложение коробок, задача 444
 Возведение в квадрат и усреднение (метод умножения) 72
 Возведение в степень 70
 Время выполнения арифметической операции 142
 Вудбери, Макс (Woodbury, Max) 354
 Вунш, Кристиан (Wunsch, Christian) 154
 Выбор кортежа, задача 427
 Выполнимость логической схемы 454
 Выполнимость формулы, задача 460
 Выпуклая последовательность 167
 Выпуклая последовательность чисел 131
 Выравнивание последовательностей 150
 Вырожденный частный случай задачи 50

Г

Гаджет 471
 вершины 487
 истинности 471

 компоненты 472, 477
 переменной 471, 477
 пересечения 487
 ребра 474
 угловой 487
 Галил Цви (Galił, Zvi) 383
 Гамильтонов путь 478
 Гамильтонов цикл 473, 478
 Гасфилд Дэн (Gusfield, Dan) 436
 Гаусс Карл Фридрих (Gauss, Carl Friedrich) 103
 Гаусс Карл Фридрих (Gauss, Karl Friedrich) 70
 Гёдель Курт (Gödel, Kurt) 454
 Гильотинное подразбиение (алгоритм) 190
 Гимбас Леонидас (Guibas, Leonidas) 132, 189
 Гиперкуб 263
 Гиргольцер Карл (Hierholzer, Carl) 241, 304
 Голдстейн Герман (Goldstine, Herman) 51
 Головоломка Bagueaudier 74
 Граф 238
 алгоритм обхода 304
 блуждание флага Франции 368
 вершина 239, 242
 достижимость 243
 наименьшее начальное время 299
 полустепень входа 243
 полустепень выхода 243
 предшествующая 243
 следующая 243
 смежная 243
 соседняя 243
 степень 243
 стоимость 377
 вершинное покрытие 469
 вложение 244
 гаджет (подграф) 471
 гамильтонов путь 478
 гамильтонов цикл 263, 473, 477
 двудольный 262
 дерево 244
 остовное 244
 длина пути 256
 дуга 239
 зависимостей 245, 291
 неявный 292
 задача о нахождении самого широкого пути 331
 задача поиска кратчайшего пути из одной вершины во все остальные 374

- задача поиска путей замены 362
- интервалов 245
- исторический пример 239
- карта дорог 375
- клика 469
- компонента 243
- компонента-приемник 299
- конфигурации 246
- кратчайший путь
 - в направленном графе 337
 - в ненаправленном графе 337
 - из одной вершины-источника 334
 - монотонно распространяющийся фронт
 - волны 343
 - при отсутствии отрицательных ребер 348
 - пробный 338
 - с маркером 341
 - соприкасающийся с отрицательным циклом 336
 - с отрицательными ребрами 336, 352
- лес 244
 - остовный 244
- маршрут 243
 - замкнутый 243
 - направленный 244
- маршрут Эйлера 262
- множественный 243
- наикратчайший путь 256
- наименьший разрез
 - подмножество ребер связь 416
- направленный 242, 259
 - ациклический 244, 245, 345
 - вычисление сильных компонент за линейное время 295
- гамильтонов путь 309
- динамическое программирование 308
- достижимость узла 282
- задача о максимальном пути 292
- интервал 310
- компонента-источник 296
- компонента-приемник 296
- кратчайший путь 344
- обратное представление 290
- обход в обратном порядке 283
- обход в прямом порядке 283
- остовное дерево вершин 259
- покрытие путями, не пересекающимися
 - в вершинах 312
- пометка вершины 259
- сильная компонента 294
- сильная связность 294
- сильно связная компонента 294
- сильно связный 244, 263
- сильных компонент 294
- топологическая сортировка 288
- уплотнение 294
- независимое множество 465
- ненаправленный 242
 - ациклический 244
 - достижимость узла 282
 - разделяющая вершина 303
- неориентированный 242
- несвязный 257
 - обход 257
 - обход по схеме компонента-за-компонентой 258
 - подсчет компонент 258
 - пометка компонент 258
- обнаружение отрицательных циклов 363
- ориентированный 242
- ослабление напряженного ребра 338
- остаточный 398
- остовное дерево
 - минимальное 256
 - поиск в глубину 263
 - поиск в ширину 263
- паросочетание 424
 - непересекающееся 450
 - палиндромическое 450
 - плотное 450
 - разреженное 450
- пересечений 245
- плоский 244
- подграф 243
 - собственный 243
- покрытие путей 431
- покрытие циклами 443
- полусвязный 302
- последовательно-параллельный 419
- представление 247
- простой 243
- путь 243
 - монотонный 444
 - направленный 244

путь максимальной ширины 257
 путь с максимальной пропускной способностью 257
 раскраска 470
 ребро 239, 242
 голова 243
 конечная точка 243
 напряженное 338
 потребность 416
 связанное сверху 417
 связанное снизу 417
 увеличение веса 376
 хвост 243
 редукция 259
 редуцированный 397
 связный 243
 лабиринт 304
 сжатие пути 332
 сильная компонента
 поиск за линейное время 299
 структура данных
 матрица смежности 249, 251
 неявное представление 251
 список смежных вершин 248, 251
 сравнение 250
 стягивание ребра 330
 таблица расстояний 375
 терминология 242
 транзитивное замыкание 390
 узел 239, 242
 узкое место 331
 уровень вершины в дереве поиска 409
 цепь вершин 474
 цикл 243
 направленный 244
 цикл нулевой длины 387
 чередующаяся цепь 426
 ширина пути 257, 331
 Граф зависимостей 145
 Графическая статика 241
 Графическое изображение интервалов времени 208
 Группировка (объединение в кластер) вершин дерева 200
 Гэйла–Шепли алгоритм 221
 Гэйл, Дэвид (Gale, David) 221

Д

Дабинс, Лестер (Dubins, Lester) 225
 Данциг, Джордж (Dantzig, George) 348, 354, 394
 Данциг, Джорджем (Dantzig, George) 337
 Двоичная куча 376
 Двоичное дерево 95
 алгоритм преобразования 100
 беспрефиксный двоичный код 212
 взаимная перестановка поддеревьев произвольного узла 100
 поворот произвольного узла 100
 поиска 97, 100
 совершенное (полное) 95
 центральная вершина 96
 Двоичное дерево поиска 126
 АА-дерево 133, 190
 В-дерево 189
 АВЛ-дерево 132, 189
 красно-черное 132, 189
 склоненное влево 190
 красно-черное дерево с уклоном влево 133
 оптимальное 126, 132, 189
 пустое 127
 симметричное В-дерево 132
 Двоичный код 212
 Дейкстра, Эдсгер (Dijkstra, Edsger) 264, 330, 347
 Дейкстры алгоритм 257, 348, 376
 время выполнения
 экспоненциальное 352, 364
 кратчайший путь
 при отсутствии отрицательных ребер 348
 поиск по первому наилучшему совпадению 351
 с отрицательными ребрами 352
 Дейкстр, Эдсгер (Dijkstra, Edsger) 323
 Дерево 244
 без корня 202
 с корнем 201
 неупорядоченное 202
 упорядоченное 202
 стягивание ребер 203
 Дерево игры 106, 111, 246
 Дерево кратчайшего пути 335
 Дерево решений 83
 Дерево с корнем 161, 450
 Дерево с циклами 361
 Децентрализованная динамическая сеть 447

Джонсон, Дональд (Johnson, Donald) 351, 376
 Диаграмма взаимодействия 241
 Диаграмма взаимодействующих сил 241
 Дизъюнктивная нормальная форма (ДНФ) 493
 Динамическое программирование 139, 344
 в НАГ 308
 в направленном ациклическом графе 292
 граф зависимостей 153, 158
 для деревьев 161
 как обход графа в обратном порядке 314
 как обход графа зависимостей в обратном порядке 291
 не всегда является улучшением 157
 общий шаблон разработки алгоритмов 144
 оптимизация пространства 140
 правильный порядок
 вычислений 145, 148, 153, 159
 сегментация текста 143
 Диница алгоритм 409
 Диниц, Ефим 409
 Дьюдни, Александр (Dewdney, Alexander) 213

Е

Евклид (Euclid) 87
 Евклида алгоритм 87
 Евклидово расстояние 183

Ж

Жадный алгоритм 146, 280
 бесконечный цикл 220
 динамическое программирование более предпочтительно 147
 доказательство корректности 211
 задача планирования 226
 индуктивный обмен местами аргументов 211
 корректный 206
 не работающий 221
 обмен местами аргументов 207, 215
 перестановка (обмен местами) аргументов 210
 Жадный алгоритм перестановки аргументов 317, 319
 Жадный алгоритм поиска минимального покрытия монотонными путями 445

З

Загоруйко, Н.Г. 154
 Задача-головоломка о кенигсбергских мостах 242

Задача коммивояжера 478
 Задача о n ферзях 103, 111
 Задача о выполнимости схемы 454
 Задача о марьяже 219
 Задача о минимальном количестве выстрелов 236
 Задача о скалолазах 235
 Задача о сумме подмножеств 112, 129, 479
 Зайдель, Раймунд (Seidel, Raimund) 383
 Зайделя алгоритм 391
 Золотое сечение 136

И

Индуктивное предположение 50, 53, 254
 Индукция 52, 55, 57, 68, 138, 211, 224, 254, 286, 320, 342, 344, 348, 350, 355, 400, 403, 404, 409
 Интерпункт (разделение слов) 113
 Исключение бейсбольных команд из раунда плей-офф, алгоритм 434
 Итинерарий 239

К

Калаба, Роберт (Kalaba, Robert) 354
 Карацуба, Анатолий 68
 Карацубы алгоритм 86
 быстрое умножение 68
 Карзанов, Александр 412
 Карпа редукция 459
 Карп Ричард (Karp, Richard) 376, 401, 407, 427, 481
 Карта дорог, исторический пример графа 239
 Карта пикселей 259
 связная область 260
 Квантифицированная логическая формула 490
 Кейран, Морис (Queyranne, Maurice) 419
 Кекуле, Август (Kekulé, August) 242
 Кёниг, Денеш (König, Dénes) 242, 427
 Кирхгофф, Густав (Kirchhoff, Gustav) 242
 Клиффорд, Уильям (Clifford, William) 242
 Кнут, Дональд (Knuth, Donald) 69
 Кобхэм, Алан (Alan Cobham) 454
 Колебательная последовательность чисел 131
 Колмогоров, Андрей Николаевич 68
 Коммивояжера задача 184
 Конечный автомат 247, 458
 детерминированный 247, 314

недетерминированный 247
 Конструкция подмножеств 247
 Контрольное значение 126
 Конъюнктивная нормальная форма (КНФ) 463
 Косараджу, Рао (Kosaraju, Rao) 298
 Косараджу–Шарира алгоритм 298
 Коэффициент удлинения (slop) макета абзаца 172
 Кратчайшее решение кубика Рубика $n \times n \times n$ 484
 Кремона, Луиджи (Cremona, Luigi) 241
 Крускала алгоритм 326
 Крускал, Йозеф (Kruskal, Joseph) 323
 Крускал, Йозефом (Kruskal, Joseph) 326
 Кука–Левина теорема 458
 Кука редукция 459
 Кук, Стивен (Cook, Stephen) 70, 457
 Кульман, Карл (Cullman, Carl) 241
 Кун, Харальд (Kuhn, Harald) 427
 Купер, Арчибалд (Couper, Archibald) 242
 Кэйли, Артур (Cayley, Arthur) 242

Л

Лабиринт катящегося кубика (rolling die maze), головоломка 277
 Лакьер, Эммануэль (Laquière, Emmanuel) 103
 Ландис, Евгений 132, 189
 Левенштейна расстояние 150
 Левенштейн, Владимир 150
 Левин, Леонид 457
 Лейзорец, Майкл (Leyzorek, Michael) 347, 381
 Лес 244
 Лоберман, Гарри (Loberman, Harry) 323, 326
 Логическая схема 453
 Логическая формула 461
 Логический вентиль 453
 Локальный минимум 92
 Ломаный путь 310
 длина 310
 монотонно возрастающий 310
 Ломуто, Нико (Lomuto, Nico) 54
 Лукашевичем, Йозеф (Łukaszewicz, Józef) 321
 Лэмпорт, Лесли (Lampert, Leslie) 264
 Люка, Эдуард (Lucas, Édouard) 48, 103, 141

М

Магнитная лента 205
 файл

сохранение 205
 считывание 205
 Мажоритарный логический элемент 472
 Максвелла–Кремоны диаграмма 241
 Максвелл, Джеймс Клерк (Maxwell, James Clerk) 241
 Максимальная возрастающая
 подпоследовательность 121, 147
 Максимальная возрастающая цифровая
 подпоследовательность массива 185
 Максимальная общая возрастающая
 подпоследовательность 167
 Максимальная общая
 подпоследовательность 154, 166
 Максимальная сумма элементов в непрерывном
 подмассиве 165
 Максимальное паросочетание в двудольном
 графе 424
 Максимальное произведение элементов
 в непрерывном подмассиве 165
 Марголит, Оded (Margolit, Oded) 383
 Мартел, Чарльз (Martel, Charles) 436
 Мартин, Элайн (Martin, Alain) 264
 Маск, Элон (Musk, Elon) 232
 Массе, Пьер (Massé, Pierre) 139
 Массив смежности 249
 Масштабирование пропускных способностей
 (capacity scaling) алгоритм 421
 Машина с прямым доступом к памяти 458
 Машина Тьюринга 458
 детерминированная 458
 недетерминированная 458
 Мемоизация 137, 246, 291, 332
 Метаграф 294
 Методика исключения нижних и верхних границ
 из рекуррентных выражений 61
 Метод математической индукции 53
 Миллер, Гари (Miller, Gary) 80
 Минимальная общая надпоследовательность 131
 Минимальное остовное дерево 317, 336
 неповторяемость 317, 328
 общая стратегия алгоритмов 318
 ребро
 бесполезное 319
 надежное 319
 Минимальное покрытие отрезков 227
 Минор с корнем 203

Минти алгоритм 348
 Минти, Джордж (Minty, George) 337, 348, 354
 Мичи, Дональд (Michie, Donald) 137, 157
 Многократно повторяющееся возведение
 в квадрат 141
 Мондри, Александр (M¹dry, Aleksander) 412
 Моргенштерн, Оскар (Morgenstern, Oskar) 139
 Морено, Джейкоб (Moreno, Jacob) 242
 Морзе код 135, 212
 Мультиграф 243
 Мура алгоритм 356
 с маркером 356
 Мура алгоритм А 340
 Мурена, Луция Лициния (Murena, Lucius
 Licinius) 113
 Мур, Эдвард (Moore, Edward) 259, 340, 354

Н

Наибольшее общее поддереву 451
 Наибольший общий делитель 87
 Направленный ациклический граф 287
 источник 287
 приемник 287
 Наук, Франц (Nauck, Franz) 103
 Недетерминированный конечный автомат (НКА)
 PSPACE-трудная задача 490
 Независимое множество в графе 161
 Непрерывная запись (scriptio continua) 213
 Нерешенная (открытая) задача
 преобразование (сортировка) двоичного
 дерева с помощью только лишь
 операций поворота и взаимной
 перестановки 101
 Нестабильное паросочетание 219
 Нидлмана-Вунша алгоритм 154
 Нидлман, Саул (Needleman, Saul) 154
 Нобелевская премия в области экономики 221
 Нэш, Джон (Nash, John) 454

О

Обход дерева
 в обратном порядке 162
 в обратном порядке (с отложенной
 выборкой) 96
 в прямом порядке (с предварительной
 выборкой) 96

упорядоченный (с порядковой выборкой) 97
 Общая надпоследовательность 131
 Общий шаблон
 доказательства корректности жадного
 алгоритма 211
 Округление матрицы 447
 Опорный элемент 54, 55, 59
 Оптимальная подпоследовательность 166
 Оптимальная подструктура 151
 Оптимальное двоичное дерево поиска 157
 Орлина алгоритм 412, 423
 Орлин, Джеймс (Orlin, James) 412
 Остовное дерево 244, 255, 256
 минимальное 256
 Остовный лес 244
 Отрицательное ребро 336
 Очередь 255
 с приоритетами 256
 Ошибка на единицу (ошибка неучтенной
 единицы off-by-one error) 54

П

Палиндром 114, 170, 278, 311
 Параллельное присваивание 307
 Парная перестановка 80
 Пачоли, Лука (Pacioli, Luca) 73
 Перансон, Эллиотт (Peranson, Elliott) 221
 Перемешивание 168
 Перфокарта 51
 Пиксел 259
 Пингала (Pi?gala) 71
 Пингала (Pingala) 134
 Пирс, Чарльз Сандерс (Peirce, Charles Sanders) 242
 Питт, Ленни (Pitt, Lenny) 105
 Планирование
 динамическое программирование 209
 жадный алгоритм 209, 226
 учебных курсов 208
 Планирование вечеринки
 подбор гостей 232
 Подграф 243
 собственный 243
 Поддерево 95
 Подпоследовательность 120
 максимальная возрастающая 120
 Подпрограмма 45

- Подстрока 120
 Поиск в глубину 104, 255, 281, 345, 454
 активный интервал вершины 284
 вершина
 активная 284
 завершенная 284
 новая 284
 как рекурсия с мемоизацией 291
 конечное время вершины 284
 начальное время вершины 284
 обратное ребро 285
 остовное дерево 255
 прямое ребро 285
 ребро дерева 285
 рекурсия 281
 секущее ребро 286
 Поиск в любом направлении 263, 264
 Поиск в ширину 256, 340, 409
 остовное дерево 256
 Поиск кратчайшего пути из одной вершины
 графа во все остальные 335
 Поиск максимального независимого множества,
 задача 465
 Поиск максимального потока (задача) 395
 Поиск максимального потока, задача 434
 Поиск наименьшего разреза (задача) 396
 Поиск покрытия непересекающихся путей,
 задача 433
 Поиск по первому наилучшему совпадению 256,
 257, 407
 Дейкстры алгоритм 351
 Поиск с возвратом 102, 104, 128
 последовательность принятия решений 111
 Пойтингерова таблица (Peutinger Table),
 исторический пример графа 239
 Покрытие непересекающихся путей 431
 Последовательность принятия решений 151
 обзор задач 111
 Построение индекса 117
 Поток 394
 ациклический 406
 векторное пространство 415
 величина 395
 выполнимый 395
 источник 394
 максимальный
 ациклический 414
 с пропускными способностями вершин 424
 многополюсный 440
 ограничение сохранения 394
 пропускная способность 395
 иррациональное число 402
 остаточная 398
 целое число 401
 путь 404
 разбиение 404, 428
 алгоритм 406
 теорема 404
 ребро
 избегание 395
 насыщение 395
 увеличивающий путь 399
 кратчайший 409
 насыщенный 408
 цель 394
 цикл 404
 циркуляция 405
 Поэтическая метрика или просодия (prosody) 134
 Предложение, которое описывает само себя 213
 Преобразование области 54, 61
 Префикс 109
 Префиксный двоичный код 212
 Прима алгоритм 256, 323
 Прим, Роберт (Prim, Robert) 319, 323
 Присваивания задача 427
 Проблема разрешимости 455
 Простое арифметическое выражение 185
 Прямой перебор 121
 Псевдо-полиномиальное время 481
- ## Р
- Рабин, Майкл (Rabin, Michael) 454
 Разделение абзаца на несколько строк 172
 Разделяй и властвуй 57, 67, 71, 382, 390
 Разрез 396
 пропускная способность 396
 Раскраска графа 228
 Расстояние редактирования 292
 Реберно-непересекающийся путь 422
 Ребоди, Овидио (Rebaudi, Ovidio) 105
 Регулярное выражение 196
 обобщенное 492
 Редакторский радиус 184

Редакторский центр 184
 Редакторское расстояние 184, 195, 245
 операция вставки 152
 операция замены 152
 операция удаления 152
 Редукционный аргумент 460
 Редукция 459
 в логарифмическом пространстве 459
 многозначная 459
 Рекуррентное выражение 53
 Рекурсивная стратегия
 поиск с возвратом 102
 Рекурсивное дерево
 54, 55, 57, 77, 104, 110, 136
 все уровни равны 59
 глубина 58, 66
 графическое представление 58
 значение узла 57
 корень 57
 лист 58
 экспоненциальное возрастание 59
 экспоненциальное убывание 59, 66
 Рекурсивный поиск с возвратом 126
 Рекурсивный прямой перебор 113
 Рекурсия 46, 134, 344, 354, 423
 с мемоизацией
 как поиск в глубину 291
 Ремзен, Айра (Remsen, Ira) 105
 Росс, Фрэнк (Ross, Frank) 393
 Рот, Элвин (Roth, Alvin) 221
 Роуз Болл, Уолтер Уильям (Rouse Ball, W. W.) 48
 Рукопожатия лемма 274

С

Саксел, Индржих (Saxeľ, Jindřich) 320
 Саллоуз, Ли (Sallows, Lee) 213
 Самуэл, Артур (Samuel, Arthur) 137
 Санкофф, Давид (Sankoff, David) 154
 Сборщик мусора 264
 Сведение 45, 50, 86
 Связный подграф 95
 Связный список 248
 Сегментация текста 113, 129, 143, 164
 Седжвик, Роберт (Sedgwick, Robert) 133, 189
 Селлерс, Питер (Sellers, Peter) 154
 Сен-Лагë, Андре (Sainte-Laguë, André) 242

Сертификат преобразования 468
 Сильвестр, Джеймс (Sylvester, James) 242
 Скиена, Стив (Skiena, Steve) 21
 Смаллиан, Реймонд (Smullyan, Raymond) 90
 Снелл, Виллеброрд (Snellius, Willebrord) 241
 Совершенное (полное) двоичное дерево 110
 Согласование матрицы 232
 Соллина алгоритм 321
 Соллин, Джордж (Sollin, George) 321
 Сортировка слиянием 51
 рекуррентное выражение 59
 рекурсивное дерево 59
 Состояние игры 106
 Спасения
 задача 441
 Сплошная заливка 260
 Сплошной блок в массиве 182
 максимальная площадь 182
 Среднее значение блока средних значений 65
 Среднее из трех (значений) 56, 79
 Стабильное паросочетание 229
 Стандартная непрерывная запись (scriptio continua) в классической латыни 113
 Стевин, Саймон (Stevin, Simon) 240
 Стек 255
 Стеффенс, Элизабет (Steffens, Elisabeth) 264
 Стиглера закон об эпонимии 135, 154
 Стиглер, Стивен (Stigler, Stephen) 135
 Стокмайер, Ларри (Stockmeyer, Larry) 493
 Стратегия разделяй и властвуй 126
 Структура данных Union-Find 328
 Структура данных непересекающихся множеств 328, 332
 Сульпиций Руф, Сервий (Sulpicius Rufus, Servius) 113
 Сумма подмножеств 155
 Суффикс 116
 Схольтен, Карел (Scholten, Carel) 264

Т

Тарри алгоритм 304
 Тарри, Гастон (Tarry, Gaston) 304
 Тарьяна алгоритм 298
 вспомогательный стек вершин 300
 Тарьян, Роберт (Tarjan, Robert) 299, 324
 Теорема о максимальном потоке и наименьшем разрезе (Maxflow-Mincut) 397

Теорема целочисленности (для максимального потока) 400
 Томицава, Нобуаки (Tomizawa, Nobuaki) 376
 Тоома–Кука алгоритм 70
 Тоом, Андрей 69
 Транзитивное замыкание 304, 390
 Транзитивное сокращение 304
 Транспортная сеть
 редуцированная 414
 Тремо, Шарль (Trémaux, Charles) 304
 Тьюринг, Алан (Turing, Alan) 139
 Тьюринга редукция 459

У

Уайтинг, Питер (Whiting, Peter) 348
 Улама расстояние 150
 Улам, Станислав (Ulam, Stanislaw) 150
 Умножение матриц 382
 логическое 389
 минимум-плюс 382, 389, 391
 повторяющееся возведение в квадрат 382
 стандартное 389, 392
 странное 382
 умножение расстояний 382
 Умножение целых чисел методом быстрых преобразований Фурье 70

Ф

Файнштейн, Эмиел (Feinstein, Amiel) 397
 Факториал 86
 Фалкерсон, Делберт (Fulkerson, Delbert) 376, 397
 Фальберг, Константин (Fahlberg, Constantin) 105
 Фано, Роберт (Fano, Robert) 213
 Фернандес-Бака, Давид (Fernández-Baca, David) 436
 Фея Рекурсия 50, 53, 121, 216, 405
 Фибоначчиева куча 324, 351, 376
 Фибоначчи, Леонардо (Fibonacci, Leonardo) 75, 135
 Фибоначчи число 245, 307
 Фишер, Майкл (Fischer, Michael) 154
 Флойда–Уоршелла алгоритм 390
 фон Нейман, Джон (von Neumann, John) 51, 139, 454
 Фонтана, Джованни (Fontana, Giovanni) 241

Форда алгоритм 354
 время выполнения
 экспоненциальное 364
 Форда релаксационный алгоритм 337
 Форда–Фалкерсона алгоритм 401, 418, 423
 время выполнения
 экспоненциальное 401
 Форд, Лестер (Ford, Lester) 337, 397
 Фридман, Дэвид (Freedman, David) 225
 Фридман, Майкл (Fredman, Michael) 324
 Фризиус, Гемма (Frisius, Gemma) 241
 Фронт волны 346, 348
 Функция линейки 73
 Функция Римана (ТФДП) 73
 Фюрер, Мартин (Fürier, Martin) 70

Х

Ханойские башни
 вариант 185
 рекурсивный алгоритм 50
 Ханойские башни (головоломка) 48, 72
 варианты 75
 нерекурсивное решение 72
 Хантингтона–Хилла алгоритм 46
 Харви, Дэвид (Harvey, David) 70
 Харрис, Теодор (Harris, Theodor) 393
 Хаффмана алгоритм 213
 кодовое дерево 214
 объединение символов 214
 Хаффмана код 213, 214, 228
 Хаффман, Дэвид (Huffman, David) 213
 Херн, Роберт (Hearn, Robert) 486
 Хиллер, Джоном (Hillier, John) 348
 Хоар, Тони (Hoare, Tony) 54
 Хопкрофт, Джон (Hopcroft, John) 427

Ц

Цвик, Ури (Zwick, Uri) 402
 Цегловский, Мацей (Cegłowski, Maciej) 232
 Цейтин, Григорий 463
 Цепь вершин 475
 Цзинь Ян Ли (Chin Yang Lee) 340, 343
 Цифровая подпоследовательность массива 184
 Цицерон (Cicero) 113
 Цузе, Конрад (Zuse, Konrad) 259, 340

Ч

Частный случай задачи 46
Черный ящик 45
Числовой лабиринт 267
Число Фибоначчи 135

Ш

Шарир, Миха (Sharir, Micha) 298
Шашки 485
 дамка 485
 международные 485
 на доске $n \times n$
 NP-трудная задача 486
 редукция от UndirectedHamCycle 486
Шашки, игра 279
Шварц, Бенджамин (Schwartz, Benjamin) 436
Шеннон, Клод (Shannon, Claude) 137, 213, 242, 340, 397
Шёнхаге, Арнольд (Schönhage, Arnold) 70
Шёнхаге-Штрассена алгоритм 70
Шепли, Ллойд (Shapley, Lloyd) 221
Шимбель, Альфонсо (Shimbel, Alfonso) 354, 380
Шир, Дуглас (Shier, Douglas) 352
Шоке, Густав (Choquet, Gustav) 321
Шрейвер, Александр (Schrijver, Alexander) 394
Штаудт, Карл фон (Staudt, Karl von) 242
Штрассена алгоритм 382
Штрассен, Фолькер (Strassen, Volker) 70, 382
Шумахер, Генрих (Schumacher, Heinrich) 103

Э

Эвристический алгоритм 56
Эдмондса–Карпа алгоритм 407, 417, 419, 420
Эдмондс, Джек (Edmonds, Jack) 376, 401, 407, 454
Эйлера маршрут 280
Эйлер, Леонард (Euler, Leonhard) 241, 304
Эйлеров контур 473
Элиас, Питер (Elias, Peter) 397
Эрикссон, Кай (Erickson, Kay) 462
Эрудит (игра на составление слов из букв) 177

Ю

Юваль, Гидеон (Yuval, Gideon) 391
Юваля алгоритм 392

Я

Язык, как множество строк 458
Якоби, Карл (Jacobi, Carl) 427
Ян Шэнь (Yang Shen) 73
Ярника алгоритм 323
 улучшенный 324
Ярник, Войтех (Jarník, Vojtěch) 323

Список алгоритмов на псевдокоде

Прежде чем продолжить, мы должны объяснить, что наша цель – рассматривать эту программу не со ссылкой на фактическое расположение данных в переменных этой машины, а просто как абстрактный вопрос о природе и количестве операций, которые необходимо выполнить во время полного решения этой задачи.

— Ада Августа Байрон Кинг, графиня Лавлейс (Ada Augusta Byron King, Countess of Lovelace),
замечания переводчика для Луиджи Ф. Менабреа (Luigi F. Menabrea),
«Конспект лекции об аналитической машине, изобретенной Чарльзом Бэббиджем»
(«Sketch of the Analytical Engine invented by Charles Babbage, Esq.») (1843 г.)

*Как играть на флейте. [берет флейту] А вот как.
Вы дуете сюда и двигаете пальцами вверх и вниз вот здесь.*

— Алан [Джон Клиз] (Alan [John Cleese]), «Как это делается» («How to Do It»),
«Летающий цирк Монти Пайтона» (Monty Python's Flying Circus),
эпизод 28 (выход в эфир 26 октября 1972 г.)

Глава 0

BottlesOfBeer(n)
FibonacciMultiply(X[0..m-1], Y[0..n-1])
PeasantMultiply(x, y)
MultiplyOrDivide(A, B, C, D)
ApportionCongress(Pop[1..n], R)
BeAMillionaireAndNeverPayTaxes()
NDaysOfChristmas(gifts[2..n])
Alouette(lapart[1..n])
BarleyMow(n)
HNGuess(Pop[1..n], R, D)

Глава 1

PeasantMultiply(x, y)
Hanoi(n, src, dst, tmp)
MergeSort(A[1..n])
Merge(A[1..n], m)
QuickSort(A[1..n])
Partition(A[1..n], p)
QuickSelect(A[1..n], k)
MomSelect(A[1..n], k)
SplitMultiply(x, y, n)
FastMultiply(x, y, n)

SlowPower(a, n)
PingalaPower(a, n)
PeasantPower(a, n)
RulerHanoi(n)
StoogeSort(A[0..n-1])
Cruel(A[1..n])
Unusual(A[1..n])
MombSelect(A[1..n], k)
MomomSelect(A[1..n], k)
Factorial(n)
Falling(n, m)
EuclidGCD(x, y)
FastEuclidGCD(x, y)
BinaryGCD(x, y)
WhoTargetsWhom(Ht[1..n])
PreOrder(v)
InOrder(v)
PostOrder(v)

Глава 2

PlaceQueens(Q[1..n], r)
PlayAnyGame(X, player)
SubsetSum(X, T)
SubsetSum(X, i, T)

ConstructSubset(X, i, T)
Splittable($A[1..n]$)
Splittable(i), 84
LISbigger($prev, A[1..n]$)
LISbigger(i, j)
LIS($A[1..n]$)
LISfirst(i)
LIS($A[1..n]$)

Глава 3

RecFibo(n)
MemFibo(n)
IterFibo(n)
IterFibo2(n)
FastRecFibo(n)
FastSplittable($A[1..n]$)
FastLIS($A[1..n]$)
FastLIS2($A[1..n]$)
EditDistance($A[1..m], B[1..n]$)
FastSubsetSum($X[1..n], T$)
InitF($f[1..n]$)
ComputeOptCost(i, k)
OptimalBST($f[1..n]$)
OptimalBST2($f[1..n]$)
OptimalBST3($f[1..n]$)
TreeMIS(v)
TreeMIS2(v)

Глава 4

GreedySchedule($S[1..n], F[1..n]$)
Huffman
BuildHuffman($f[1..n]$)
HuffmanEncode($A[1..k]$)
HuffmanEncodeOne(x)
HuffmanDecode($B[1..n]$)

Глава 5

RecursiveDFS(v)
IterativeDFS(s)
WhateverFirstSearch(s)
WFSAll(G)
MarkEveryVertexDuh(G)
CountComponents(G)
CountAndLabel(G)

LabelOne($v, count$)
WhateverFirstSearch(s)
EagerWFS(s)
ThreeColorSearch(s)
ThreeColorStep(s)
ThreeColorStackSearch(s)
ThreeColorStackStep(s)
ThreeColorQueueSearch(s)
ThreeColorQueueStep(s)
GarbageCollect(s)
CollectStep(s)
Mutate(s)

Глава 6

DFS(v)
DFSAll(G)
Preprocess(G)
PreVisit(v)
PostVisit(v)
DFSAll(G)
DFS($v, clock$)
IsAcyclic(G)
IsAcyclicDFS(v)
PostProcess(G)
PostProcessDFS(v)
PostProcessDag(G)
PostProcessDagDFS(v)
PostProcessDag(G)
TopologicalSort(G)
Memoize(x)
DFS(v)
DynamicProgramming(G)
LongestPath(v, t)
LongestPath(s, t)
StrongComponents(G)
KosarajuSharir(G)
PushPostRevDFS(v, S)
LabelOneDFS(v, r)
FindLow(G)
FindLowDFS(v)
Tarjan(G)
TarjanDFS(v)
Tarry(G)
RecTarry(v)
Tarry2(G)
RecTarry2($v, clock$)

Глава 7

ShorterEdge(i, j, k, l)
 Borůvka(V, E)
 AddAllSafeEdges(E, F, count)
 Jarník(V, E, s)
 JarníkInit(V, E, s)
 JarníkLoop(V, E, s)
 Kruskal(V, E)
 Find(v)
 Union(u, v)
 FindSafeEdges(V, E)
 AddSafeEdges(V, E, F)
 Borůvka(V, E)

Глава 8

InitSSSP(s)
 Relax($u \rightarrow v$)
 FordSSSP(s)
 BFS(s)
 BFSWithToken(s)
 DagSSSP(s)
 PushDagSSSP(s)
 Dijkstra(s)
 NonnegativeDijkstra(s)
 BellmanFord(s)
 BellmanFord(s)
 Moore(s)
 BellmanFordDP(s)
 BellmanFordDP2(s)
 BellmanFordDP3(s)
 BellmanFordFinal(s)
 BellmanBored(s)

Глава 9

ObviousAPSP(V, E, w)
 JohnsonAPSP(V, E, w)
 ShimbelAPSP(V, E, w)
 AllPairsBellmanFord(V, E, w)
 FischerMeyerAPSP(V, E, w)
 MatrixSquare(A)
 FischerMeyerInnerLoop(D)
 KleeneAPSP(V, E, w)
 FloydWarshall(V, E, w)
 RecursiveAPSP(V, E, w)
 RecAPSP(D, n, i, j, k)

Глава 10

GreedyFlow(G, c, s, t)
 QueyranneFatPaths

Глава 11

MaxMultiFlow($G, s[1..σ], t[1..τ]$)

Глава 12

CircuitSat(K)

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
Тел.: +7(499) 782-38-89. Электронная почта: books@alians-kniga.ru.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине:
www.galaktika-dmk.com.

Джефф Эриксон

Алгоритмы

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Зам. главного редактора	<i>Сенченкова Е. А.</i>
Научный редактор	<i>Иванов П. Б., Бронников С. В.</i>
Перевод	<i>Снастин А. В., Иванов П. Б.</i>
Корректор	<i>Абросимова Л. А.</i>
Верстка	<i>Паранская Н. В.</i>
Дизайн обложки	<i>Мовчан А. Г.</i>

Формат 70×90^{1/16}.
Печать цифровая. Усл. печ. л. 42,74
Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com