ТИМ
РАФГАРДЕН

СОВЕРШЕННЫЙ АЛГОРИТМ

ЖАДНЫЕ АЛГОРИТМЫ И ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

COMPUTER
SCIENCE

Algorithms Illuminated

Part 3: Greedy Algorithms and Dynamic Programming

Tim Roughgarden

ТИМ
РАФГАРДЕН

СОВЕРШЕННЫЙ АЛГОРИТМ

ЖАДНЫЕ АЛГОРИТМЫ И ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

COMPUTER SCIENCE

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2020

ББК 32.973.2-018
УДК 004.42
Р26

Рафгарден Тим

Р26 Совершенный алгоритм. Жадные алгоритмы и динамическое программирование. — СПб.: Питер, 2020. — 256 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1445-0

Алгоритмы — это сердце и душа computer science. Без них не обойтись, они есть везде — от сетевой маршрутизации и расчетов по геномике до криптографии и машинного обучения. «Совершенный алгоритм» превратит вас в настоящего профи, который будет ставить задачи и мастерски их решать как в жизни, так и на собеседовании при приеме на работу в любую IT-компанию.

В новой книге Тим Рафгарден расскажет о жадных алгоритмах (задача планирования, минимальные остовные деревья, кластеризация, коды Хаффмана) и динамическом программировании (задача о рюкзаке, выравнивание последовательностей, кратчайшие пути, оптимальные деревья поиска).

Серия книг «Совершенный алгоритм» адресована тем, у кого уже есть опыт программирования, и основана на онлайн-курсах, которые регулярно проводятся с 2012 года. Вы перейдете на новый уровень, чтобы увидеть общую картину, разобраться в низкоуровневых концепциях и математических нюансах.

Познакомиться с дополнительными материалами и видеороликами автора (на английском языке) можно на сайте www.algorithmsilluminated.org.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018
УДК 004.42

ISBN 978-0999282946 англ.
ISBN 978-5-4461-1445-0

Оглавление

Памяти Стивена Х. Шнайдера

Предисловие

Это третья книга из серии в четырех частях, основанной на моих онлайн-курсах по алгоритмам, регулярно проводимых с 2012 года и которые, в свою очередь, основаны на курсе бакалавриата, многократно преподававшемся мною в Стэнфордском университете. Для читателей этой книги знакомство с первыми двумя частями серии не является обязательным. Тем не менее для усвоения ее содержания читателям желательно иметь хотя бы общее представление об обозначении О-большое (глава 2 *части 1* или приложение В *части 2*), алгоритмах «разделяй и властвуй» (глава 3 *части 1*) и графах (глава 7 *части 2*).

О чем эта книга

«Совершенный алгоритм» — это вводный курс (теоретическая основа и многочисленные примеры) по двум фундаментальным парадигмам проектирования алгоритмов.

Жадные алгоритмы и их применение. Жадные алгоритмы решают задачи, принимая последовательность близоруких (миопических) и необратимых решений. В большинстве случаев они легко разрабатываются и часто являются невероятно быстрыми. Правда, большинство жадных алгоритмов не гарантируют правильности, но мы по ходу изложения материала рассмотрим несколько уникальных по своим возможностям приложений, являющихся исключениями из этого правила. Примеры включают задачи планирования, оптимальное сжатие и минимальные остовные деревья графов.

Динамическое программирование и его применение. Немногие преимущества, обретенные нами вследствие серьезного изучения алгоритмов, способны соперничать с возможностями, которые дает освоение динамического программирования. Эта парадигма проектирования, впрочем, требует обширной практики. Вместе с тем она имеет бесчисленное множество приложений к задачам, которые кажутся неразрешимыми с помощью любого более простого метода. Эффективность этого своеобразного «курса молодого бойца» по динамическому программированию будет удвоена посредством тура по некоторым (см. выше) приложениям указанной парадигмы, включающего рассмотрение задачи о ранце, алгоритм выравнивания геномных последовательностей Нидлмана—Вунша, алгоритм Кнута для оптимальных бинарных деревьев поиска и алгоритмы кратчайшего пути Беллмана—Форда и Флойда—Уоршелла.

С содержанием книги можно ознакомиться подробнее, обратившись к разделам «Выводы», завершающим каждую главу и концентрирующимся на наиболее важных моментах. «Руководство по разработке алгоритмов» дает представление о том, как жадные алгоритмы и динамическое программирование вписываются в более крупную алгоритмическую картину.

Обратите внимание: разделы книги, отмеченные звездочкой, — самые сложные для усвоения, и в отсутствие у читателей достаточного времени к этим разделам можно будет вернуться позднее (на качестве усвоения наиболее доступной части материала книги это обстоятельство никак не скажется).

Темы, затронутые в трех других частях. В первой части книги «Совершенный алгоритм» освещены асимптотические обозначения (обозначение O-большое и его близких родственников), алгоритмы «разделяй и властвуй» и основной метод, рандомизированная быстрая сортировка `Quicksort` и ее анализ, а также линейно-временные алгоритмы отбора. *Часть 2* охватывает структуры данных (кучи, сбалансированные деревья поиска, хеш-таблицы, фильтры Блума), графовые примитивы (приоритетный поиск в ширину или в глубину, связность, кратчайшие пути) и их приложения (от дедупликации до анализа социальных сетей). *Часть 4* посвящена NP-полноте, возможности которой открывают для разработчика алгоритмов и стратегий решения вычислительно неразрешимых задач, включая анализ эвристик и локальный поиск.

Навыки, которые вы приобретете

Освоение алгоритмов требует времени и усилий. Ради чего все это?

Возможность стать более эффективным программистом. Вы изучите несколько невероятно быстрых подпрограмм для обработки данных и некоторые полезные структуры для организации данных, которые сможете непосредственно использовать в ваших собственных программах. Реализация и применение этих алгоритмов расширит и улучшит ваши навыки программирования. Вы также узнаете основные приемы разработки алгоритмов, актуальных для решения разнообразных задач в широких областях, получите инструменты для прогнозирования их производительности. Такие «шаблоны»

могут пригодиться для разработки новых алгоритмов решения возникающих перед вами задач.

Развитие аналитических способностей. Алгоритмические описания, мыслительная работа над алгоритмами — это весьма ценный опыт. Посредством математического анализа вы получите углубленное понимание конкретных алгоритмов и структур данных, описанных в этой и последующих книгах серии. Вы приобретете навыки работы с несколькими математическими методами, которые широко применяются для анализа алгоритмов.

Алгоритмическое мышление. Научившись разбираться в алгоритмах, вы начнете замечать, что они окружают вас повсюду: едете ли вы в лифте, наблюдаете ли за стаей птиц, управляете ли вы своим инвестиционным портфелем или даже следите за тем, как учится ребенок. Алгоритмическое мышление становится все более полезным и превалирующим в дисциплинах, не связанных с информатикой, включая биологию, статистику и экономику.

Знакомство с величайшими достижениями информатики. Изучение алгоритмов напоминает просмотр эффектного клипа с суперхитами последних шестидесяти лет развития computer science. На фуршете для специалистов в области computer science вы больше не будете чувствовать себя не в своей тарелке, если кто-то отпустит шутку по поводу алгоритма Дейкстры. Прочитав эти книги, вы точно сможете поддержать разговор.

Успех на собеседованиях. На протяжении многих лет студенты развлекали меня рассказами о том, как знания, почерпнутые из этих книг, позволяли им успешно справляться с любым вопросом, на который им требовалось ответить во время собеседования.

В чем особенность книг этой серии

Книги в составе серии преследуют только одну цель: *максимально доступным способом научить вас основам алгоритмов*. Относитесь к ним как к конспекту, который бы у вас был, если бы вы брали индивидуальные уроки у опытного наставника по алгоритмам.

Существует, впрочем, ряд прекрасных, гораздо более традиционных и энциклопедически выверенных учебников по алгоритмам. Призываю вас разузнать подробнее, что это за книги, и найти те из них, что отвечают вашему запросу. Кроме того, есть несколько книг, представляющих очевидный интерес для программистов, ищущих готовые реализации алгоритмов на конкретном языке программирования. К слову, многие такие реализации находятся в свободном доступе в интернете.

Для кого эта книга?

Весь смысл этой книги, как и онлайн-курсов, на которых она базируется, — быть широко- и легкодоступной читателю настолько, насколько это возможно. Мои курсы вызвали интерес у людей разного возраста, социального положения и профессионального опыта: среди них есть и студенты, и разработчики программного обеспечения (как состоявшиеся, так и начинающие), и ученые, и мегаспециалисты со всех уголков мира.

Однако эта книга — не введение в программирование. Для усвоения ее содержания читателям желательно обладать базовыми навыками программирования на каком-либо распространенном языке (например, Java, Python, C, Scala, Haskell). Для развития навыков программирования рекомендуем обратиться к бесплатным онлайн-курсам обучения основам программирования.

По мере необходимости мы также используем математический анализ, чтобы разобраться в том, как и почему алгоритмы действительно работают. Находящиеся в свободном доступе конспекты лекций «Математика для Computer Science» под авторством Эрика Лемана, Тома Лейтона и Альберта Мейера являются превосходным и занимательным повторительным курсом по системе математических обозначений (например, Σ и $\forall$), основам теории доказательств (метод индукции, доказательство от противного и др.), дискретной вероятности и многому другому.

Дополнительные ресурсы

Эта книга основана на онлайн-курсах, которые в настоящее время запущены в рамках проектов Coursera и Stanford Lagunita. Автор также создал несколько ресурсов в помощь читателям для повторения и закрепления опыта, который можно извлечь из онлайн-курсов.

Видео. Если вы больше настроены смотреть и слушать, нежели читать, то обратитесь к материалам YouTube, доступным на сайте www.algorithmsilluminated.org, затрагивающим все темы этой серии книг, а также некоторые более сложные темы за рамками книжной серии.

Тестовые задания. Как проверить, что вы действительно усваиваете понятия, представленные в этой книге? Тестовые задания с решениями и объяснениями разбросаны по всему тексту; когда вы сталкиваетесь с одним из них, призываю вас остановиться и подумать над ответом, прежде чем продолжать чтение.

Задачи в конце главы. В конце каждой главы вы найдете несколько относительно простых вопросов на проверку усвоения материала, а затем задачи различного уровня сложности, не ограниченные по времени выполнения. Решения этих задач в книгу не включены, но заинтересованные читатели могут обратиться ко мне или взаимодействовать между собой через дискуссионный форум книги (см. ниже).

Задачи по программированию. В конце большинства глав предлагается реализовать программный проект, целью которого является закрепление детального понимания алгоритма путем создания его рабочей реализации.

Наборы данных, а также тестовые примеры и их решения можно найти на сайте www.algorithmsilluminated.org.

Дискуссионные форумы. Основной причиной успеха онлайн-курсов является возможность общения для слушателей, реализованная через дискуссионные форумы. Это позволяет им помогать друг другу в лучшем понимании материала курса, а также отлаживать свои программы посредством взаимодействия и обсуждения. Читатели этих книг имеют такую же возможность благодаря форумам, доступным на сайте www.algorithmsilluminated.org.

Благодарности

Эти книги не существовали бы без энтузиазма и «интеллектуального голода», которые показали тысячи участников моих курсов по алгоритмам на протяжении многих лет как на кампусе в Стэнфорде, так и на онлайновых платформах. Я особенно благодарен тем, кто предоставлял подробные отзывы в отношении более раннего проекта этой книги, в их числе: Тоня Бласт, Юань Цао, Карлос Гуйя, Джим Хумельсин, Владимир Кокшенев, Байрам Кулиев и Даниэль Зингаро.

Я всегда ценю отклики читателей. О своих предложениях читателям лучше всего сообщать на упомянутом выше форуме.

Тим Рафгарден
Нью-Йорк,
апрель 2019

От издательства

Не удивляйтесь, что эта книга начинается с тринадцатой главы. С одной стороны, она является частью курса «Совершенный алгоритм» Тима Рафгардена, а с другой — самостоятельным изданием, в котором рассматриваются вопросы жадных алгоритмов и динамического программирования. Приложения А, Б и В вы можете найти в книге «Совершенный алгоритм. Основы» (часть 1) и «Совершенный алгоритм. Графовые алгоритмы и структуры данных» (часть 2).

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция). Мы будем рады узнать ваше мнение! На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

13 Введение в жадные алгоритмы

Проектирование и анализ алгоритмов строятся преимущественно на балансировании между общими принципами проектирования алгоритмов и их вариантами, адаптированными для решения конкретных вычислительных задач. В проектировании алгоритмов нет «серебряной пули», то есть такого универсального технического приема, который способен решить любую вычислительную задачу. Но есть несколько общих парадигм проектирования, способных помочь в решении самых разных прикладных задач. Научиться этим парадигмам и их наиболее известным воплощениям — одна из главных целей этой книжной серии.

13.1. Парадигма проектирования жадных алгоритмов

13.1.1. Парадигмы алгоритмов

Что такое «парадигма проектирования алгоритмов»? В первой книге данной серии (далее *часть 1*) приводится канонический пример — парадигма «разделяй и властвуй»:

ПАРАДИГМА «РАЗДЕЛЯЙ И ВЛАСТВУЙ»

1. *Разделить* входные данные на более мелкие подзадачи.
2. *Рекурсивно* покорить подзадачи.
3. *Совместить* решения подзадач в решение первоначальной задачи.

В *части 1* мы видели многочисленные примеры этой парадигмы: алгоритмы сортировки слиянием `MergeSort` и быстрой сортировки `QuickSort`, алгоритм Карацубы умножения двух n-значных целых чисел за $O(n^{1,59})$-е время, алгоритм Штрассена умножения двух $n \times n$-матриц за $O(n^{2,71})$-е время и многие другие. Первая половина этой книги посвящена парадигме проектирования *жадных* алгоритмов. Что же такое жадный алгоритм на самом деле? Много

крови и чернил было пролито при поиске ответа на этот вопрос, поэтому мы удовлетворимся неофициальным определением[1].

ЖАДНАЯ ПАРАДИГМА

Итеративно конструировать решение посредством последовательности близоруких решений и надеяться, что в конце концов все получится.

Лучший способ прочувствовать жадные алгоритмы — использовать примеры, в том числе и представленные в последующих главах[2].

13.1.2. Темы жадной парадигмы

Вот несколько тем, которые следует высматривать в наших примерах (возможно, для того чтобы этот раздел казался менее абстрактным, вы захотите перечитать его после просмотра одного или нескольких примеров). Во-первых, во многих задачах удивительно легко придумать один или даже несколько жадных алгоритмов, причем вполне работоспособных. В действительности жадные алгоритмы могут быть отличным лекарством от преодоления писательского тупика, в котором вы застряли на вполне обыденной задачке, но при этом они могут быть труднопримени мыми для оценки того, какой именно жадный подход является наиболее перспективным. Во-вторых, анализ времени выполнения часто отличается афористичностью. Например, многие жадные алгоритмы сводятся к сортировке

[1] Формальные определения жадных алгоритмов можно найти в статье «(Incremental) Priority Algorithms» Аллана Бородина (Allan Borodin), Мортена Н. Нильсена (Morten N. Nielsen) и Чарльза Ракоффа (Charles Rackoff) («Алгоритмы (инкрементного) приоритета», Algorithmica, 2003).

[2] Читатели *части 2* уже видели жадный алгоритм, а именно алгоритм кратчайшего пути Дейкстры, итеративно вычисляющий кратчайшее расстояние от стартовой вершины s до любой другой вершины графа. На каждой итерации алгоритм безвозвратно и близоруко берет оценку кратчайшего расстояния до одной дополнительной вершины, никогда не пересматривая свое решение. В графах только с неотрицательными длинами ребер все в итоге срабатывает, а все полученные оценки кратчайшего пути являются правильными.

(плюс линейное количество дополнительной обработки), и в этом случае расчетное время выполнения реализации будет составлять $O(n \log n)$, где n — число объектов для сортировки[1] (обозначение O-большое подавляет постоянные множители, а различные логарифмические функции отличаются постоянным множителем, поэтому нет необходимости указывать основание логарифма). Наконец, часто трудно понять, действительно ли предлагаемый жадный алгоритм возвращает правильный выход для каждого возможного входа. Риск в том, что одно из необратимых близоруких решений алгоритма вернется и будет вас преследовать, а задним числом будет обнаружено, что алгоритм никуда не годится. И даже когда жадный алгоритм является формально правильным, доказать это бывает трудно[2].

ОСОБЕННОСТИ И НЕДОСТАТКИ ЖАДНОЙ ПАРАДИГМЫ

1. Легко придумать один или несколько жадных алгоритмов.
2. Легко проанализировать время выполнения алгоритма.
3. Трудно установить правильность алгоритма.

Одна из причин труднодоказуемости правильности жадных алгоритмов — в том, что большинство таких алгоритмов на самом деле не являются правильными, то есть существуют входы, для которых алгоритм не может получить желаемый результат.

---

[1] Например, двумя алгоритмами сортировки со временем выполнения $O(n \log n)$ являются сортировка слиянием `MergeSort` (см. главу 1 *части 1*) и кучевая сортировка `HeapSort` (см. главу 10 *части 2*). В качестве альтернативы рандомизированная быстрая сортировка `QuickSort` (см. главу 5 *части 1*) имеет среднее время выполнения $O(n \log n)$.

[2] Читатели, хорошо знакомые с *частью 1*, знают, что все три темы представляют большой контраст парадигме «разделяй и властвуй». Придумать хороший алгоритм «разделяй и властвуй», как правило, непросто, но в процессе придумывания обычно присутствует момент озарения, то есть момент интуитивной разгадки решаемой задачи. Анализировать время выполнения алгоритмов «разделяй и властвуй» бывает трудно из-за перетягивания каната между силами разрастающихся подзадач и сокращающейся работой над каждой подзадачей (см. главу 4 *части 1*). Наконец, доказательства правильности алгоритмов «разделяй и властвуй» обычно являются прямыми индукциями.

ПРЕДУПРЕЖДЕНИЕ

Большинство жадных алгоритмов не могут считаться абсолютно правильными.

Этот момент особенно трудно принять в отношении умных жадных алгоритмов, которые вы изобрели сами. В глубине души вы можете быть убеждены, что созданный вами алгоритм всегда должен решать задачу правильно. Но... чаще всего это убеждение является необоснованным[1].

Теперь, когда моя совесть чиста, давайте посмотрим на несколько тщательно отобранных примеров задач, которые *могут* быть правильно решены с помощью разумно спроектированного жадного алгоритма.

13.2. Задача планирования

Наше первое тематическое исследование касается *планирования*, в котором целью является распределение работ по одному или нескольким совместно используемым ресурсам для оптимизации некоторой цели. Например, ресурс может представлять компьютерный процессор (где работы соответствуют заданиям), учебный предмет (где работы соответствуют лекциям) или календарь на день (где работы соответствуют встречам).

13.2.1. Постановка

В области планирования подлежащие выполнению задания обычно называются *работами*, имеющими различные характеристики. Предположим, каждая работа j имеет известную продолжительность, или *длину* ℓ_j, которая является количеством времени, необходимым для выполнения работы (например, продолжительность лекции или встречи). Кроме того, каждая работа имеет *вес* w_j, причем более высокие веса соответствуют более приоритетным работам.

[1] Даже не лишенный недостатков жадный алгоритм может обеспечить сверхбыстрое разрешение задачи; к этому вопросу мы вернемся в *части 4*.

13.2.2. Сроки завершения

План, или *расписание* работ, конкретизирует порядок выполнения работ. В экземпляре задачи с n работами существует $n! = n \times (n-1) \times (n-1) \times \ldots \times 2 \times 1$ разных расписаний. Очень много! Какое из них мы должны предпочесть?

Далее нужно определить *целевую функцию*, назначающую каждому расписанию количественную отметку и квантифицирующую то, что мы хотим. Таким образом:

СРОКИ ЗАВЕРШЕНИЯ

Срок завершения $C_j(\sigma)$ работы j в расписании σ есть сумма длин работ, предшествующих j в σ, плюс длина самого j.

Другими словами, срок завершения работы в расписании — это суммарное время, которое проходит до полной обработки работы.

УПРАЖНЕНИЕ 13.1

Рассмотрим задачу, в рамках которой предлагается выполнить три работы с $\ell_1 = 1$, $\ell_2 = 2$ и $\ell_3 = 3$, начиная с первой из них. Каковы сроки выполнения всех трех работ в этом расписании? (Веса работ в этом вопросе не имеют значения, поэтому мы их не указали.)

а) 1, 2 и 3

б) 3, 5 и 6

в) 1, 3 и 6

г) 1, 4 и 6

(Решение и пояснение см. в разделе 13.2.4.)

13.2.3. Целевая функция

Что представляет собой хороший график? Мы стремимся к минимальным срокам завершения работ, но компромиссы неизбежны — в любом расписании работы, запланированные ранее, будут иметь и более ранние сроки завершения, а те, что запланированы ближе к концу, будут иметь более поздние сроки завершения.

Один из способов находить компромиссы между работами состоит в минимизации *суммы взвешенных сроков завершения*. В математике эта целевая функция транслируется в

$$\min_{\sigma} \sum_{j=1}^{n} w_j C_j(\sigma), \tag{13.1}$$

где минимизация выполняется над всеми $n!$ возможными расписаниями σ, а $C_j(\sigma)$ обозначает срок завершения работы j в расписании σ. Это эквивалентно минимизации средневзвешенных сроков завершения работ, где усредняющие веса пропорциональны значениям w_j.

Например, рассмотрим работы в упражнении 13.1 и предположим, что их веса соответственно равны $w_1 = 3$, $w_2 = 2$ и $w_3 = 1$. Если мы запланируем последовательное выполнение работ, то сумма взвешенных сроков завершения будет равна

$$\underbrace{3\times 1}_{\text{работа № 1}} + \underbrace{2\times 3}_{\text{работа № 2}} + \underbrace{1\times 6}_{\text{работа № 3}} = 15.$$

Перепроверив все 3! = 6 возможных расписаний, мы можем подтвердить, что это то расписание, которое минимизирует сумму взвешенных сроков завершения. Как решить эту задачу в общем случае с учетом произвольного множества длин и весов работ на входе?

ЗАДАЧА: МИНИМИЗАЦИЯ СУММЫ ВЗВЕШЕННЫХ СРОКОВ ЗАВЕРШЕНИЯ

Вход: множество из n работ с положительными длинами ℓ_1, ℓ_2, ..., ℓ_n и положительными весами w_1, w_2, ..., w_n.

Выход: последовательность работ, которая минимизирует сумму взвешенных сроков завершения (13.1).

С $n!$ разными расписаниями вычисление лучшего путем исчерпывающего поиска даже не обсуждается (кроме единичных исключений). Нам нужен более умный алгоритм[1].

13.2.4. Решение упражнения 13.1

Правильный ответ: (в). Мы можем визуализировать расписание, расположив работы одну над другой, с увеличением срока снизу вверх (рис. 13.1). Срок завершения работы является сроком, соответствующим ее верхнему краю. Срок завершения первой работы — это просто ее длина, или 1. Вторая работа должна дождаться завершения первой работы, поэтому ее срок завершения является суммой длин первых двух работ, то есть 3. Третья работа даже не начинается до срока 3, а затем для ее завершения требуется еще 3 единицы времени, поэтому ее срок завершения составляет 6.

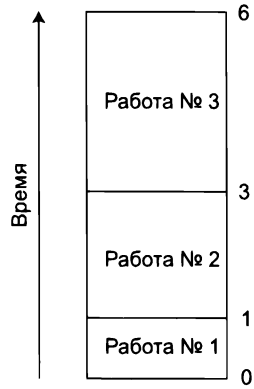


Рис. 13.1. Сроки завершения трех работ составляют 1, 3 и 6

13.3. Разработка жадного алгоритма

Жадные алгоритмы, по всей видимости, хорошо подходят для задачи планирования работ, минимизируя взвешенную сумму сроков завершения. Выход имеет итеративную структуру, где работы обрабатываются по одной. Почему бы не использовать жадный алгоритм, который итеративно решает, какая работа будет следующей?

Первым шагом нашего плана является решение двух частных случаев общей задачи. Наши решения этих случаев покажут, как может выглядеть жадный алгоритм в общем случае. Затем мы сузим область до одного алгоритма-

[1] Например, $n!$ больше 3,6 миллиона при $n = 10$, больше 2,4 квинтиллиона при $n = 20$ и больше, чем расчетное число атомов в известной Вселенной, при $n \geq 60$. Таким образом, никакое мыслимое усовершенствование компьютерной технологии не превратит исчерпывающий поиск в полезный алгоритм.

кандидата и докажем, что именно этот кандидат решает задачу правильно. Процесс, с помощью которого мы приходим к этому алгоритму, более важен для запоминания, чем сам алгоритм; этот процесс является повторяемым, и вы можете его использовать в своих собственных приложениях.

13.3.1. Два частных случая

Давайте предположим, что для задачи минимизации взвешенной суммы сроков завершения на самом деле существует правильный жадный алгоритм. Как бы он выглядел, если исходить из того, что все работы имеют одинаковую длину (но, возможно, разные веса) или, напротив, имеют одинаковый вес (но, возможно, разную длину)?

УПРАЖНЕНИЕ 13.2

(1) если все длины работ являются идентичными, должны ли мы раньше планировать работы меньшего или большего веса?

(2) если все веса работ являются идентичными, должны ли мы раньше планировать более короткие или более длинные работы?

а) большего; короткие

б) меньшего; короткие

в) большего; длинные

г) меньшего; длинные

(Решение и пояснение см. в разделе 13.3.3.)

13.3.2. Дуэльные жадные алгоритмы

В общем случае работы могут иметь разный вес и разную длину. Всякий раз, когда наши два эмпирических правила — предпочитать более короткую работу и работу с бóльшим весом — на наше счастье для пары работ совпадают, мы знаем, какую из них запланировать первой (более короткую с бóльшим весом). Но что, если эти два правила дают противоречивый совет? Что мы

должны делать с одной короткой работой с низким весом и одной длинной работой с высоким весом?

Каким будет самый простой жадный алгоритм, который сработает как надо? Каждая работа имеет два параметра, и алгоритм должен смотреть на оба. Наилучшим вариантом было бы разработать формулу, которая компилирует длину и вес каждой работы в единую отметку (вклад), вследствие чего планирование работ от самой высокой до самой низкой отметки гарантированно минимизирует сумму взвешенных сроков завершения. Если такая формула существует, то из наших двух частных случаев следует, что она должна иметь два свойства: (i) оставляя длину фиксированной, она должна увеличиваться от веса работы; (ii) оставляя вес фиксированным, она должна уменьшаться от длины работы (напомним, что чем выше отметки, тем лучше). Потратьте минуту на мозговой штурм нескольких формул, которые имеют оба этих свойства.

* * *

Пожалуй, самой простой функцией, которая увеличивается в весе и уменьшается в длине, является разница между ними:

$$\text{предложение № 1 для отметки работы } j: w_j - \ell_j.$$

Указанная отметка могла бы быть отрицательной, но это не ставит никаких препятствий для последовательного выстраивания работ от самой высокой до самой низкой отметки. Существует, впрочем, немало других вариантов. Например, отношение двух параметров является еще одним кандидатом:

$$\text{предложение № 2 для отметки работы } j: \frac{w_j}{\ell_j}.$$

Эти две функции вычисления отметки приводят к двум разным жадным алгоритмам.

ЖАДНАЯ РАЗНОСТЬ GREEDYDIFF

Планировать работы в убывающем порядке $w_j - \ell_j$

(произвольно разрывая совпадение значений).

ЖАДНОЕ ОТНОШЕНИЕ GREEDYRATIO

Планировать работы в убывающем порядке $\frac{w_j}{\ell_j}$

(произвольно разрывая совпадение значений).

Таким образом, уже наше первое тематическое исследование иллюстрирует первую тему жадной парадигмы (раздел 13.1.2): предложить для задачи несколько соперничающих жадных алгоритмов обычно не составляет труда. Но какой из двух алгоритмов, если таковые имеются, является правильным? Быстрый способ исключить один из них — найти экземпляр, в котором два алгоритма выводят разные расписания с разными значениями целевой функции. В отношении любого алгоритма, результаты которого окажутся в этом примере хуже, мы можем заключить, что он не всегда является оптимальным. В двух частных случаях с работами одинакового веса или одинаковой длины оба алгоритма поступают правильно. Простейшим возможным примером исключения одного из них может быть экземпляр задачи, в которой две работы имеют разные веса и длины, вследствие чего оба алгоритма планируют работы в противоположных порядках. То есть мы ищем две работы, порядок которых по разнице является противоположным их порядку по отношению. Пример:

	Работа #1	**Работа #2**
Длина	$\ell_1 = 5$	$\ell_2 = 2$
Вес	$w_1 = 3$	$w_2 = 1$

Первая работа имеет большее крупное отношение ($\frac{3}{5}$ против $\frac{1}{2}$), но большую разность (–2 против –1). Таким образом, алгоритм `GreedyDiff` планирует вторую работу первой, тогда как алгоритм `GreedyRatio` планирует противоположное.

УПРАЖНЕНИЕ 13.3

Какова сумма взвешенных сроков завершения в расписаниях, выводимых соответственно алгоритмами `GreedyDiff` и `GreedyRatio`?

а) 22 и 23

б) 23 и 22

в) 17 и 17

г) 17 и 11

(Решение и пояснение см. в разделе 13.3.3.)

Мы продвинулись вперед, исключив алгоритм `GreedyDiff` из дальнейшего рассмотрения. Однако результат упражнения 13.3 *не* ведет непосредственно к тому, что алгоритм `GreedyRatio` всегда будет оптимальным. Насколько нам известно, существуют другие случаи, когда этот алгоритм выдает неоптимальное расписание. Вы всегда должны скептически относиться к алгоритму, который не сопровождается доказательством его правильности, даже если этот алгоритм поступает правильно в нескольких тестовых примерах и крайне скептически относится к жадным алгоритмам.

В нашем случае алгоритм `GreedyRatio`, по сути, гарантированно *будет* минимизировать сумму взвешенных сроков завершения.

Теорема 13.1 (правильность алгоритма `GreedyRatio`). *Для каждого множества положительных весов работ w_1, w_2, ..., w_n и положительных длин работ ℓ_1, ℓ_2, ..., ℓ_n алгоритм* `GreedyRatio` *выводит расписание с минимально возможной суммой взвешенных сроков завершения.*

Это логическое утверждение не является очевидным, и вы не должны доверять ему, не получив доказательств. В соответствии с третьей темой жадной парадигмы (раздел 13.1.2) это доказательство занимает весь следующий раздел.

О ЛЕММАХ, ТЕОРЕМАХ И Т. П.

В математической записи самые важные технические высказывания имеют статус *теорем*. *Лемма* — это техническое высказывание, которое помогает с доказательством теоремы (во многом как подпрограмма помогает с реализацией более крупной программы). *Следствие* — это высказывание, которое непосредственно вытекает из уже доказанного результата, например, частного случая теоремы. Мы используем термин

«утверждение» (в иных источниках оно именуется пропозицией или высказыванием) для автономных технических высказываний, которые сами по себе не особенно важны.

Оставшейся темой жадной парадигмы является простота анализа времени выполнения (раздел 13.1.2). Здесь это, безусловно, так. Алгоритм `GreedyRatio` всего лишь сортирует работы по отношению, что занимает $O(n \log n)$ времени, где n — это число работ на входе (см. сноску 1 на с. 24).

13.3.3. Решения упражнений 13.2–13.3

Решение упражнения 13.2

Правильный ответ: (а). Сначала предположим, что все n работ имеют одинаковую длину, допустим, 1. Тогда каждое расписание имеет точно такое же множество сроков завершения — $\{1, 2, 3, ..., n\}$ — и единственный вопрос состоит в том, какая работа получает срок завершения и каков этот срок. Наша семантика для весов работ, безусловно, предполагает, что работы с бóльшим весом должны получать меньшие сроки завершения, и это действительно так. Например, вы бы не захотели запланировать работу с весом 10 третьей (со сроком завершения 3) и работу с весом 20 пятой (со сроком завершения 5); вам было бы лучше поменять позиции этих двух работ, что уменьшило бы сумму взвешенных сроков завершения на 20 (как вы можете убедиться сами).

Второй случай, в котором все работы имеют одинаковый вес, немного тоньше. Здесь вы хотите отдать предпочтение более коротким работам. Например, рассмотрим две работы единичного веса с длинами 1 и 2. Если вы сначала запланируете более короткую работу, то сроки завершения составят 1 и 3 при суммарном 4. В обратном порядке сроки завершения составят 2 и 3 с худшим итогом 5. В общем случае запланированная работа сначала вносит вклад в сроки завершения *всех* работ, так как все работы должны ждать завершения первой. При прочих равных условиях планирование самой короткой работы сначала минимизирует это негативное влияние. Вторая работа вносит свой вклад во все сроки завершения, кроме первой работы, поэтому вторая самая короткая работа должна быть запланирована следующей, и т. д.

Решение упражнения 13.3

Правильный ответ: (б). Алгоритм `GreedyDiff` планирует вторую работу первой. Срок завершения этой работы составляет $C_2 = \ell_2 = 2$, тогда как срок завершения другой работы составляет $C_1 = \ell_2 + \ell_1 = 7$. Сумма взвешенных сроков завершения тогда составит

$$w_1 \times C_1 + w_2 \times C_2 = 3 \times 7 + 1 \times 2 = 23.$$

Алгоритм `GreedyRatio` планирует первую работу первой, в результате приходя к срокам завершения $C_1 = \ell_1 = 5$ и $C_2 = \ell_1 + \ell_2 = 7$ и сумме взвешенных сроков завершения, равной

$$3 \times 5 + 1 \times 7 = 22.$$

Поскольку для этого примера алгоритму `GreedyDiff` не удается вычислить оптимальное расписание, он не всегда является правильным.

13.4. Доказательство правильности

Алгоритмы «разделяй и властвуй» обычно имеют формульные доказательства правильности, состоящие из простой индукции. Но для жадных алгоритмов доказательства правильности являются скорее искусством, чем наукой — будьте готовы приложить максимум усилий. В тех случаях, когда в доказательствах правильности жадных алгоритмов имеются повторяющиеся темы, мы будем подчеркивать их по мере продвижения.

Доказательство теоремы 13.1 включает яркий пример одной из таких тем: *обмен аргументами*. Ключевая идея — в том, чтобы доказать, что каждое возможное решение можно улучшить, модифицировав его так, чтобы оно больше походило на результат жадного алгоритма. В этом разделе представлено два варианта. В первом из них будем исходить от противного и использовать обмен аргументами для того, чтобы показать решение, которое является «слишком хорошим, чтобы быть истинным». Во втором варианте используем обмен аргументами для того, чтобы показать, что каждое возможное решение мо-

жет быть итеративно введено в выход жадного алгоритма, при этом улучшая решение только по пути[1].

13.4.1. Случай отсутствия совпадающих значений: высокоуровневый план

Перейдем к доказательству теоремы 13.1. Зададим множество работ с положительными весами $w_1, w_2, ..., w_n$ и длинами $\ell_1, \ell_2, ..., \ell_n$. Мы должны показать, что алгоритм `GreedyRatio` создает расписание, которое минимизуирует сумму взвешенных сроков завершения (13.1). Начнем с двух допущений.

ДВА ДОПУЩЕНИЯ

(1) Работы проиндексированы в невозрастающем порядке отношения веса к длине:

$$\frac{w_1}{\ell_1} \geq \frac{w_2}{\ell_2} \geq ... \geq \frac{w_n}{\ell_n}. \tag{13.2}$$

(2) Между отношениями нет совпадений значений: $\frac{w_i}{\ell_i} \neq \frac{w_j}{\ell_j}$ всякий раз, когда $i \neq j$.

Первое допущение скорее отражает договоренность с целью минимизировать наши издержки в формальных обозначениях. Переупорядочивание работ во входе не влияет на решаемую задачу. Следовательно, мы всегда можем переупорядочить и переиндексировать работы так, чтобы допущение (13.2) соблюдалось. Второе допущение накладывает нетривиальное ограничение на вход; в разделе 13.4.4 мы проделаем некоторую дополнительную работу, для того чтобы его удалить. В совокупности эти два допущения приводят

[1] Обмен аргументами (*argument exchange*) — это лишь один из многих способов доказать правильность жадного алгоритма. Например, в главе 9 *части 2* в доказательстве правильности алгоритма Дейкстры вместо обмена аргументами использовалась индукция. И индукция и обмен аргументами играют свою роль в доказательствах правильности алгоритма жадного кодирования Хаффмана (глава 14) и алгоритмов минимального остовного дерева Прима и Краскала (глава 15).

к тому, что задания индексируются в строго убывающем порядке отношения веса к длине.

Высокоуровневый план должен исходить от противного. Вспомним, что в этом типе доказательства вы исходите из *противоположного* тому, что хотите доказать, строя на этом допущении доказательство с помощью последовательности логически правильных шагов, завершающейся явно ложным утверждением. Такое противоречие приводит к тому, что допущение не может быть истинным, что и доказывает желаемое утверждение.

Для начала мы допустим, что алгоритм `GreedyRatio` создает расписание σ заданных работ, которое *не* является оптимальным. Таким образом, существует оптимальное расписание σ* этих работ со строго меньшей суммой взвешенных сроков завершения. Используем разницу между σ и σ* для явного конструирования расписания, которое *еще лучше*, чем σ*; это будет противоречить допущению о том, что σ* является оптимальным расписанием.

13.4.2. Обмен работами при последовательной инверсии

Предположим, от противного, что алгоритм `GreedyRatio` производит расписание σ и что существует оптимальное расписание σ* со строго меньшей суммой взвешенных сроков завершения. Согласно допущению (1), жадное расписание σ планирует работы в порядке индекса (сначала работа 1, затем работа 2, вплоть до работы *n*); см. рис. 13.2.

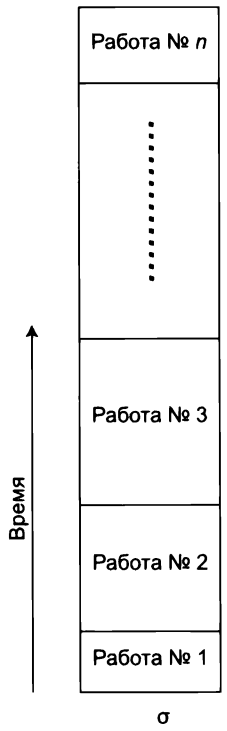


Рис. 13.2. Жадное расписание σ, в котором работы запланированы в порядке неубывания отношения веса к длине

Двигаясь снизу вверх в жадном расписании, индексы работ всегда поднимаются вверх. Это не относится ни к одному

другому расписанию. Уточняя это логическое утверждение, определим *последовательную инверсию* в расписании как пару i, j работ, таких что $i > j$ и работа i обрабатывается непосредственно перед работой j. Например, на рис. 13.2 если бы задания 2 и 3 обрабатывались в обратном порядке, то они составляли бы последовательную инверсию (с $i = 3$ и $j = 2$).

Лемма 13.2 (нежадные расписания имеют инверсии). *Каждое расписание, отличное от жадного расписания* σ, *имеет, по крайней мере, одну последовательную инверсию.*

Доказательство: мы доказываем противопоставление[1]. В отсутствие последовательных инверсий индекс каждой работы по крайней мере на 1 больше, чем у работы, ей предшествующей. Существует n работ, и максимально возможный индекс равен n, поэтому между индексами работ, расположенными подряд, не может быть скачков, равных 2 или больше. Это означает, что $\hat{\sigma}$ совпадает с расписанием, вычисленным жадным алгоритмом. *Ч. Т. Д.*[2]

Возвращаясь к доказательству теоремы 13.1, мы допускаем, что существует оптимальное расписание σ^* заданных работ со строго меньшей суммой взвешенных сроков завершения, чем жадное расписание σ. Поскольку $\sigma^* \neq \sigma$, лемма 13.2 применима к σ^*, и есть идущие подряд работы i, j в σ^* при $i > j$ (рис. 13.3(*a*)). Как использовать этот факт для выявления еще одного расписания σ', которое еще лучше, чем σ^*, тем самым создавая противоречие?

Ключевая идея заключается в выполнении *обмена*. Мы определяем новое расписание σ', идентичное σ^*, за исключением того, что работы i и j обрабатываются в обратном порядке, где теперь j обрабатывается непосредственно перед i. Работы перед i и j («дела» на рис. 13.3) совпадают с σ^* и σ' (и в том же порядке). То же касается работ, которые следуют и за i, и за j («больше дел»).

---

[1] *Противопоставлением* (contrapositive) высказыванию «если A является истинным, то B является истинным» является логически эквивалентное высказывание «если B не является истинным, то и A не является истинным». Например, противопоставлением лемме 13.2 является: если $\hat{\sigma}$ не имеет последовательных инверсий, то совпадает с жадным расписанием σ.

[2] *Ч. Т. Д.* — аббревиатура выражения «что и требовалось доказать» от латинского quod erat demonstrandum (Q.E.D.). В математической записи она используется в конце доказательства для обозначения его завершения.

13.4.3. Анализ стоимости и преимущества

Каковы последствия обмена, показанного на рис. 13.3?

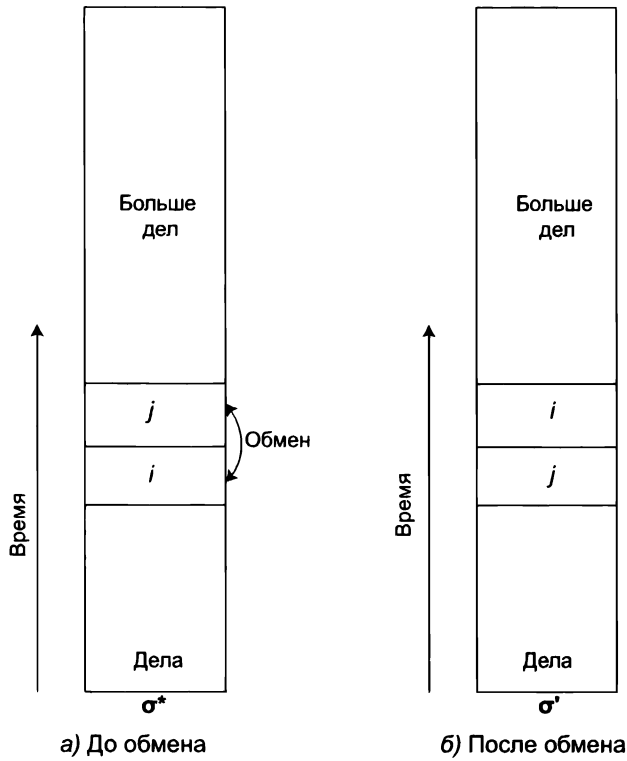


Рис. 13.3. Получение нового расписания σ′ из предположительно оптимального расписания σ* путем обмена работами в последовательной инверсии (при $i > j$)

УПРАЖНЕНИЕ 13.4

Какое влияние обмен оказывает на срок завершения: (i) работы, отличной от i или j; (ii) работы i; и (iii) работы j?

а) (i) для ответа недостаточно информации; (ii) срок увеличивается; (iii) уменьшается;

б) (i) для ответа недостаточно информации; (ii) срок уменьшается; (iii) увеличивается;

в) (i) без изменений; (ii) срок увеличивается; (iii) уменьшается;

г) (i) без изменений; (ii) срок уменьшается; (iii) увеличивается.

(Решение и пояснение см. в разделе 13.4.5.)

Решение упражнения 13.4 позволяет закончить доказательство. В результате обмена работами i и j в последовательной инверсии срок завершения C_i увеличивается на длину ℓ_j работы j, что увеличивает целевую функцию (13.1) на $w_i \times \ell_j$. Преимущество — в том, что срок завершения C_j уменьшается на длину ℓ_i задания i, что уменьшает целевую функцию (13.1) на $w_j \times \ell_i$.

В итоге приходим к следующему:

$$\underbrace{\sum_{k=1}^{n} w_k C_k(\sigma')}_{\text{значение целевой функции } \sigma'} = \underbrace{\sum_{k=1}^{n} w_k C_k(\sigma^*)}_{\text{значение целевой функции } \sigma^*} + \underbrace{w_i \ell_j - w_j \ell_i}_{\text{эффект обмена}}. \qquad (13.3)$$

Теперь самое время использовать тот факт, что σ^* запланировало i и j в «неправильном порядке» при $i > j$. Наши постоянные допущения (1) и (2) приводят к тому, что работы индексируются в строго убывающем порядке отношения веса к длине, поэтому

$$\frac{w_i}{\ell_i} < \frac{w_j}{\ell_j}.$$

После очистки знаменателей получаем

$$\underbrace{w_i \ell_j}_{\text{стоимость обмена}} < \underbrace{w_j \ell_i}_{\text{преимущество обмена}}.$$

Поскольку преимущество обмена превышает допустимую стоимость, уравнение (13.3) говорит нам, что значение целевой функции от σ' меньше значения целевой функции от σ^*.

Однако расписание σ^* должно быть оптимальным при наименьшей возможной сумме взвешенных сроков завершения! Мы пришли к желаемому противоречию, которое завершает доказательство теоремы 13.1 для случая, когда все работы имеют разные отношения веса к длине.

13.4.4. Обработка совпадений значений

Теперь докажем правильность алгоритма `GreedyRatio` (теорема 13.1) с учетом наличия совпадения значений (ties) в коэффициентах веса к длине работы и исходя из того, что (1) работы индексируются в неубывающем порядке отношения веса к длине, так как это не влечет потерю общности. Доказательство позволяет проиллюстрировать изящный поворот в обмене аргументами из предыдущего раздела, который проходит прямо, а не от противного.

Мы будем использовать бóльшую часть выполненной ранее работы, внеся в первоначальный план ряд корректив. Как и прежде, обозначим через $\sigma = 1, 2, \ldots, n$ расписание, вычисленное алгоритмом `GreedyRatio`. Рассмотрим произвольное соперничающее расписание σ^*, показав непосредственно, последовательностью обменов работами, что сумма взвешенных сроков завершения расписания σ не больше, чем σ^*. Доказав это для каждого расписания σ^*, мы заключим, что σ, по сути дела, является оптимальным расписанием.

Теперь допустим, что $\sigma^* \neq \sigma$ (учитывая, что $\sigma^* = \sigma$). По лемме 13.2 σ^* имеет последовательную инверсию — две работы i и j, причем $i > j$, а j запланирована сразу после i. Возьмем σ' из σ^*, поменяв местами i и j в расписании (рис. 13.3). Как и для уравнения (13.3), стоимость и преимущества этого обмена составляют соответственно $w_i\ell_j$ и $w_j\ell_i$. Поскольку $i > j$ и работы индексируются в неубывающем порядке отношения веса к длине,

$$\frac{w_i}{\ell_i} \leq \frac{w_j}{\ell_j}$$

и, следовательно,

$$\underbrace{w_i\ell_j}_{\text{стоимость обмена}} \leq \underbrace{w_j\ell_i}_{\text{преимущество обмена}} . \qquad (13.4)$$

Другими словами, обмен не может увеличить сумму взвешенных сроков завершения — сумма может уменьшиться или остаться прежней[1].

[1] Мы более не сталкиваемся с непосредственным противоречием в случае, когда σ^* является оптимальным графиком, поскольку σ' может быть представлен другим, столь же оптимальным графиком.

УПРАЖНЕНИЕ 13.5

Инверсия в расписании — пара k, m работ, где $k < m$, и m обрабатывается перед k. Работы k и m не обязательно должны следовать одна за другой — некоторые работы могут быть запланированы после m и до k. Предположим, что σ_1 — это расписание с последовательной инверсией i и j, где $i > j$, и получим σ_2 из σ_1, инвертировав порядок i и j. Как число инверсий в σ_2 соотносится с этим числом в σ_1?

а) σ_2 имеет на одну инверсию меньше, чем σ_1;

б) σ_2 имеет такое же число инверсий, как σ_1;

в) σ_2 имеет на одну инверсию больше, чем σ_1;

г) ни один ответ не является правильным.

(Решение и пояснение см. в разделе 13.4.5.)

В завершение доказательства возьмем произвольное соперничающее расписание σ^* и многократно поменяем работы местами, удалив последовательные инверсии[1]. Поскольку число инверсий уменьшается с каждым обменом (упражнение 13.5), этот процесс в конечном итоге завершается. По лемме 13.2 он может завершиться только на жадном расписании σ. Значение целевой функции может уменьшаться только в течение этого процесса (на (13.4)), поэтому σ по крайней мере равно σ^*. Это является истинным для каждого варианта σ^*, поэтому σ действительно является оптимальным. *Ч. Т. Д.*

13.4.5. Решения упражнений 13.4–13.5

Решение упражнения 13.4

Правильный ответ: (в). Прежде всего, работе k, кроме i и j, было все равно, что i и j поменяются местами. Это легче всего увидеть для работы k,

[1] Читатели, знакомые с алгоритмом сортировки пузырьком `BubbleSort`, могут распознать его использование здесь в анализе (но не в алгоритме).

выполненной до i и j в σ^* (как часть «дел» на рис. 13.3). Поскольку обмен происходит после завершения k, он не влияет на срок завершения k (количество времени, которое проходит до завершения k). Для работы k, обработанной после i и j в σ^* (как часть «больше дел» на рис. 13.3), множество работ, завершенных до k в точности, является одинаковым в σ^* и в σ'. Срок завершения работы зависит только от множества работ, предшествующих ей (а не от их порядка), поэтому работа k формально ничем не отличается от других и завершается одновременно в обоих расписаниях. Что касается работы i, то срок ее завершения увеличивается на σ' — в связи с ожиданием завершения тех же работ, что и раньше («дела»), плюс ожидание завершения работы j. Поэтому срок ее завершения увеличивается на ℓ_j. Схожим образом, работа j ожидает завершения тех же работ, что и раньше, за исключением того, что в σ' она больше не ждет i. Таким образом, срок завершения работы j уменьшается на ℓ_i.

Решение упражнения 13.5

Правильный ответ: (а). Если $\{k, m\} = \{i, j\}$, то k и m образуют инверсию в σ_1, но не в σ_2 (потому что обмен не отменяет их инверсию). Если хотя бы одна из k или m отличается одновременно от i и j и, следовательно, появляется либо перед i и j, либо после i и j в обоих расписаниях, то обмен не влияет на относительный порядок k и m (см. рис. 13.4). Мы заключаем, что σ_2 имеет точно такие же инверсии, как σ_1, за исключением того, что инверсия i и j удалена.

ВЫВОДЫ

- Жадные алгоритмы конструируют решения итеративно, посредством последовательности близоруких решений, и надеются, что в конце концов все получится.
- Как правило, предложить один или несколько жадных алгоритмов для задачи и проанализировать их времена выполнения не составляет большого труда.
- Самые жадные алгоритмы не всегда являются правильными.

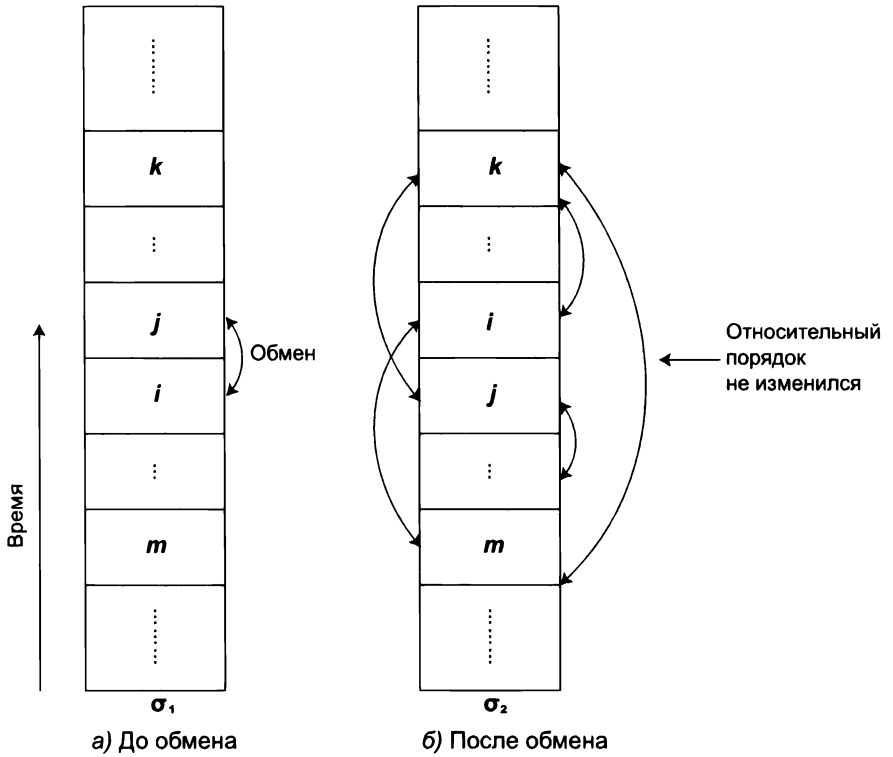


Рис. 13.4. Перестановка работ местами при последовательной инверсии уменьшает общее число инверсий на 1. Пять выделенных пар работ в (*б*) расположены в одном и том же относительном порядке в обоих расписаниях

- Даже когда жадный алгоритм является абсолютно правильным, доказать это бывает трудно. В задачах с длинами и весами жадное их упорядочивание от самого высокого отношения веса к длине к самому низкому минимизирует взвешенную сумму сроков завершения.
- Аргументы перестановки местами являются одними из наиболее распространенных методов, используемых в доказательствах правильности жадных алгоритмов. Идея состоит в том, чтобы показать, что каждое возможное решение может быть улучшено путем его модифицирования так, чтобы оно больше походило на выход жадного алгоритма.

Задачи на закрепление материала

Задача 13.1. В качестве входа задано n работ, каждая из которых имеет длину ℓ_j и крайний срок d_j. Определите запаздывание $\lambda_j(\sigma)$ работы j в расписании σ как разность $C_j(\sigma) - d_j$ между сроком завершения работы и крайним сроком либо как 0, если $C_j(\sigma) \leq d_j$ (см. с. 25 с определением срока завершения работы в расписании). В данной задаче рассматривается минимизация максимального запаздывания $\max_{j=1}^{n} \lambda_j(\sigma)$.

Какой из следующих ниже жадных алгоритмов формирует расписание, минимизирующее максимальное запаздывание? Вы можете допустить, что совпадения значений отсутствуют.

а) Запланировать работы в порядке увеличения крайнего срока d_j;

б) запланировать работы в порядке увеличения времени обработки p_j;

в) запланировать работы в порядке возрастания произведения $d_j \times p_j$;

г) ни один ответ не является правильным.

Задача 13.2. Теперь рассмотрим задачу минимизации *суммарного* запаздывания, $\sum_{j=1}^{n} \lambda_j(\sigma)$.

Какой из следующих ниже жадных алгоритмов производит расписание, которое минимизирует суммарное запаздывание? Вы можете допустить, что совпадения значений отсутствуют.

а) Запланировать работы в порядке увеличения крайнего срока d_j;

б) запланировать работы в порядке увеличения времени обработки p_j;

в) запланировать работы в порядке возрастания произведения $d_j \times p_j$;

г) ни один ответ не является правильным.

Задача 13.3. В качестве входа задано n работ, каждая из которых имеет время начала s_j и время окончания t_j. Две работы *конфликтуют*, если они накладываются во времени — если одна из них начинается между временем начала и окончания другой. В этой задаче цель состоит в том, чтобы выбрать подмножество максимального размера с работами, которые не имеют кон-

фликтов. (Например, с учетом трех работ, потребляющих интервалы [0, 3], [2, 5] и [4, 7], оптимальное решение состоит из первой и третьей работ.) План состоит в том, чтобы разработать итеративный жадный алгоритм, который на каждой итерации безвозвратно добавляет новую работу j в текущее решение и удаляет из будущего рассмотрения все работы, которые конфликтуют с j.

Какой из следующих ниже жадных алгоритмов гарантированно вычисляет оптимальное решение? Вы можете допустить, что совпадения значений отсутствуют.

а) На каждой итерации выбирать оставшуюся работу с самым ранним временем завершения;

б) на каждой итерации выбирать оставшуюся работу с самым ранним временем начала;

в) на каждой итерации выбирать оставшуюся работу, требующую наименьшего времени (то есть с наименьшим значением $t_j - s_j$);

г) на каждой итерации выбирать оставшуюся работу с наименьшим числом конфликтов с другими оставшимися работами.

Задачи по программированию

Задача 13.4. На своем любимом языке программирования реализуйте алгоритмы `GreedyDiff` и `GreedyRatio` из раздела 13.3 для минимизации взвешенной суммы сроков завершения. Выполните оба алгоритма на нескольких примерах. Насколько расписания, вычисленные алгоритмом `GreedyRatio`, лучше расписаний алгоритма `GreedyDiff`? Кейсы и наборы данных для сложных задач см. на веб-сайте www.algorithmsilluminated.org.

14 Коды Хаффмана

Все любят сжатие данных. Например, количество фотографий, которые вы можете хранить на своем смартфоне, зависит от того, насколько можно сжать файлы с малыми потерями или без потерь. Время, необходимое для загрузки файла, тоже зависит от степени его сжатия: чем «плотнее» упакован файл, тем быстрее происходит его загрузка. *Кодирование Хаффмана* является широко используемым методом сжатия без потерь, поскольку всякий раз при импорте или экспорте аудиофайла MP3 компьютер использует коды Хаффмана. В этой главе мы узнаем об оптимальности кодов Хаффмана, а также о невероятно быстром жадном алгоритме их вычисления.

14.1. Коды

14.1.1. Двоичные коды фиксированной длины

Перед тем как приступить к определению задачи или алгоритма, давайте подготовим почву. *Алфавит* Σ есть конечное непустое множество символов. Например, Σ может быть множеством из 64 символов, которое включает в себя все 26 букв латинского алфавита (как верхний, так и нижний регистр) плюс знаки препинания и некоторые специальные символы. *Двоичный код* для алфавита — это способ записи каждого его символа в виде отдельной двоичной символьной цепочки (то есть в виде последовательности *бит*, означающих нули и единицы). Например, с алфавитом из 64 символов естественным кодированием является связывание каждого символа с одной из $2^6 = 64$ двоичных цепочек длиной 6, причем каждая цепочка используется однократно. Это пример двоичного кода *фиксированной длины*, в котором для кодирования каждого символа используется одинаковое число бит. Например, примерно так работают коды ASCII.

Коды фиксированной длины являются естественным решением, но *можем ли мы добиться лучшего*?

14.1.2. Коды переменной длины

Когда некоторые символы алфавита встречаются намного чаще других, коды переменной длины могут быть эффективнее, чем коды фиксированной длины.

Однако допущение кодов переменной длины вносит усложнение, которое мы проиллюстрируем следующим примером. Рассмотрим четырехсимвольный алфавит, скажем, $\Sigma = \{A, B, C, D\}$. Естественным кодом фиксированной длины для этого алфавита является:

Символ	Кодирование
A	00
B	01
C	10
D	11

Предположим, в нашем коде мы хотели бы обойтись меньшим числом бит, используя для некоторых символов однобитное кодирование, например, имеющее следующий вид:

Символ	Кодирование
A	0
B	01
C	10
D	1

Этот короткий код может быть только лучше, верно?

УПРАЖНЕНИЕ 14.1

С приведенным выше двоичным кодом переменной длины кодировкой чего является строка «001»?

а) *AB*

б) *CD*

в) *AAD*

г) для ответа недостаточно информации

(Решение и пояснение см. в разделе 14.1.6.)

Суть упражнения 14.1 заключается в том, что при использовании кодов переменной длины и отсутствии дополнительных мер предосторожности бывает неясно, где заканчивается один символ и начинается следующий. Эта проблема не возникает с кодами фиксированной длины. Если каждый символ кодируется с использованием 6 бит, то второй символ всегда начинается с 7-го бита, третий символ — с 13-го бита и т. д. Во избежание двусмысленности с кодами переменной длины мы должны наложить ограничение.

14.1.3. Беспрефиксные коды

Мы можем устранить любую двусмысленность, настояв на применении кода *без префиксов*. Это означает, что для каждой пары разных символов $a, b \in \Sigma$ кодировка a не является префиксом b, и наоборот. Каждый код фиксированной длины по умолчанию не содержит префиксов. Код переменной длины в предыдущем разделе таковым не является: кодировка «*A*» является префиксом «*B*», и схожим образом происходит с «*D*» и «*C*».

В случае беспрефиксного кода закодированные последовательности являются однозначными и могут быть декодированы очевидным способом. Если первые 5 бит последовательности соответствуют кодировке символа a, то a определенно был первым закодированным символом — поскольку код не содержит префиксов, эти 5 бит не могут соответствовать кодировке (префиксу кодировки) любого другого символа. Если следующие 7 бит соответствуют кодировке b, то b был вторым закодированным символом, и т. д.

Вот пример беспрефиксного кода для алфавита $\Sigma = \{A, B, C, D\}$, который не имеет фиксированной длины:

Символ	Кодирование
A	0
B	10
C	110
D	111

Поскольку «0» используется для кодирования *A*, кодирования остальных трех символов должны начинаться с «1». Поскольку *B* кодируется как «10», кодирования *C* и *D* начинаются с «11».

14.1.4. Преимущества беспрефиксных кодов

Беспрефиксные коды переменной длины могут быть эффективнее, чем коды фиксированной длины, когда символы имеют очень разные частоты. Например, предположим, что у нас есть следующая статистика о частотах символов в нашем приложении (возможно, из прошлого опыта или из предварительной обработки файла, подлежащего кодированию):

Символ	Частота
A	60 %
B	25 %
C	10 %
D	5 %

Сравним производительность беспрефиксных кодов фиксированной и переменной длины:

Символ	Код фиксированной длины	Беспрефиксный код переменной длины
A	00	0
B	01	10
C	10	110
D	11	111

Под «производительностью» мы подразумеваем среднее число бит, используемых для кодирования символа, причем символы взвешены в соответствии с их частотами. Код фиксированной длины всегда использует 2 бита, то есть это и его средняя длина на символ. А как насчет кода переменной длины? Мы могли бы надеяться, что он — лучше, учитывая, что он использует только 1 бит большую часть времени (60 %) и прибегает к 3 битам только в редких случаях (15 %).

УПРАЖНЕНИЕ 14.2

Каково среднее число бит на символ, используемое приведенным выше кодом переменной длины?

а) 1,5

б) 1,55

в) 2

г) 2,5

(Решение и пояснение см. в разделе 14.1.6.)

14.1.5. Определение задачи

В предыдущем примере показано, что наилучший двоичный код для задания зависит от частоты символов. Эта сложная алгоритмическая задача рассматривается ниже.

ЗАДАЧА: ОПТИМАЛЬНЫЕ БЕСПРЕФИКСНЫЕ КОДЫ

Вход: неотрицательная частота p_a для каждого символа a алфавита Σ размера $n \geq 2$.

Выход: беспрефиксный двоичный код с минимальной возможной средней длиной кодирования:

$$\sum_{a \in \Sigma} p_a \cdot (\text{число бит, используемых для кодирования } a).$$

Как узнать заранее частоты разных символов? В некоторых приложениях присутствует много данных или знаний о той или иной предметной области. Например, любой специалист по генетике может сообщить вам типичную частоту каждого нуклеооснования (аденин A, цитозин C, гуанин G и тиамин T) в ДНК человека. В случае кодирования MP3-файла кодировщик вычисляет частоты символов при подготовке начальной цифровой версии файла (возможно, после аналого-цифрового конвертирования), а затем использует оптимальный беспрефиксный код для дальнейшего сжатия файла.

На первый взгляд задача вычисления оптимального беспрефиксного кода едва ли поддается решению. Число возможных кодов растет экспоненциально вместе с n, поэтому даже при скромных значениях n нет надежды на исчер-

пывающий поиск по всем из них[1]. Впрочем, эта задача может быть решена эффективно с помощью гладкого жадного алгоритма.

14.1.6. Решения упражнений 14.1–14.2

Решение упражнения 14.1

Правильный ответ: (г). Предлагаемый код переменной длины создает неоднозначность, и более чем одна последовательность символов приведет к кодировке «001». Возможны вариации *AB* (кодированные соответственно как «0» и «01») или *AAD* (кодированные как «0», «0» и «1»). При наличии только кодирования невозможно узнать, какое вменялось значение.

Решение упражнения 14.2

Правильный ответ: (б). Расширяя средневзвешенное значение, мы имеем среднее число бит на символ = $\underbrace{1\times 0,6}_{\text{"A"}} + \underbrace{2\times 0,25}_{\text{"B"}} + \underbrace{3\times(0,1+0,05)}_{\text{"C" и "D"}} = 1,5$.

Для этого множества частот символов код переменной длины использует на 22,5 % бит меньше, чем код фиксированной длины (в среднем), что позволяет говорить о значительной экономии.

14.2. Коды в виде деревьев

«Беспрефиксное» ограничение в задаче оптимального беспрефиксного кода звучит немного пугающе. Как исключить префиксы из кода в процессе его сборки? Решающее значение для рассуждений о задаче имеет метод ассоциирования кодов с помеченными бинарными деревьями[2].

---

[1] Например, существует $n!$ разных беспрефиксных кодов, кодирующих один символ с использованием одного бита («0»), другой — с использованием двух бит («10»), третий — с использованием трех бит («110») и т. д.

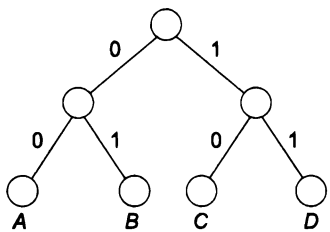
[2] Каждый узел бинарного дерева может иметь левый дочерний элемент, правый дочерний элемент, оба или ни одного. Узел без дочерних элементов называется *листом*. Не-лист также называется *внутренним узлом*. Как узлы, так и ребра могут

14.2.1. Три примера

Связь между кодами и деревьями проще всего объяснить на примерах. Наш код фиксированной длины

Символ	Кодирование
A	00
B	01
C	10
D	11

может быть представлен посредством полного бинарного дерева с четырьмя листьями:



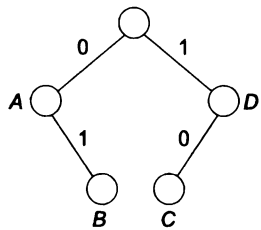
Каждое ребро, соединяющее узел с его левым или правым дочерним узлом, помечается соответственно «0» либо «1». Листья дерева помечены четырьмя символами алфавита. Каждый путь от корня до помеченного узла проходит по двум ребрам. Мы можем интерпретировать метки этих двух ребер как кодирование символа листа. Например, поскольку путь от корня к узлу с меткой «*B*» проходит по левому дочернему ребру («0»), за которым следует правое дочернее ребро («1»), мы можем интерпретировать путь как кодирование символа *b* последовательностью 01. Это соответствует кодированию *B* в нашем коде фиксированной длины. То же верно и для других трех символов.

быть помечены. По какой-то причине айтишники, похоже, думают, что деревья растут вниз, и поэтому рисуют свои деревья таким образом.

Далее вспомним наш первый (беспрефиксный) код переменной длины:

Символ	Кодирование
A	0
B	01
C	10
D	1

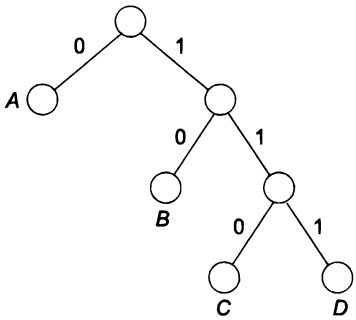
Этот код может быть представлен с помощью показанного ниже помеченного бинарного дерева



Опять-таки имеется четыре узла, помеченные символами алфавита — два листа и их родители. Это дерево определяет кодирование для каждого символа посредством последовательности меток ребер на пути от корня до узла, помеченных этим символом. Например, переход от корня к узлу с меткой «*A*» требует прохождения только одного левого дочернего ребра, соответствующего кодированию «0». Кодирования, определенные этим деревом, соответствуют кодированиям в приведенной выше таблице.

Наконец, мы можем представить наш беспрефиксный код переменной длины деревом следующей конфигурации.

Символ	Кодирование
A	0
B	10
C	110
D	111



В более общем случае *каждый* двоичный код может быть представлен в виде бинарного дерева, в котором левое и правое дочерние ребра помечены соответственно «0» и «1» и каждый символ алфавита используется в качестве метки только для одного узла[1]. И наоборот, каждое такое дерево определяет двоичный код, причем метки ребер на путях от корня до помеченных узлов обеспечивают кодирование символов. Число ребер в пути равно числу бит, используемых для кодирования соответствующего символа, что позволяет утверждать следующее.

Утверждение 14.1 (длина кодирования и глубина дерева). *Для каждого двоичного кода длина кодирования в битах символа* $a \in \Sigma$ *равна глубине узла с меткой* a *в соответствующем дереве.*

Например, в приведенном выше беспрефиксном коде лист уровня 1 соотносится с символом 1-битного кодирования (*A*), лист уровня 2 — с символом с 2-битным кодированием (*B*), листья уровня 3 — с двумя символами с 3-битными кодированиями (*C* и *D*).

14.2.2. Какие деревья представляют беспрефиксные коды?

Мы видели, что бинарные деревья могут представлять все двоичные коды, с префиксами или без. Существует неопровержимое доказательство, когда код, соответствующий дереву, не является беспрефиксным.

---

[1] Предположим, что наибольшее число бит используется для кодирования символа ℓ. Сформируйте полное бинарное дерево глубины ℓ. Кодирование каждого символа a определяет путь через дерево начиная с корня, и конечный узел этого пути должен быть помечен символом a. Наконец, многократно обрезайте непомеченные листья до тех пор, пока не останется ни одного.

Вернемся к трем рассмотренными нами примерам. Первое и третье деревья, соответствующие двум беспрефиксным кодам, выглядят совершенно поразному. Но оба имеют одно свойство: символами алфавита помечены только листья. И напротив, во втором дереве помечены два не-листа.

В общем случае кодирование символа a является префиксом кодирования другого символа b тогда и только тогда, когда узел, помеченный a, является предком узла, помеченного b. Помеченный внутренний узел является предком (помеченных) листьев в его поддереве и приводит к нарушению беспрефиксного ограничения[1]. И наоборот, поскольку ни один лист не может быть предком другого, дерево с метками только на листьях определяет беспрефиксный код. Декодирование последовательности бит сводится к следующему: пройти по дереву сверху вниз, поворачивая влево или вправо всякий раз, когда следующий входной бит равен соответственно 0 или 1. Когда лист достигнут, его метка указывает на следующий символ в последовательности, и процесс перезапускается от корня с оставшимися входными битами. Например, в третьем коде декодирование входа «010111» приводит к трем обходам от корня к листу, заканчивающимся в A, затем в B и, наконец, в D (рис. 14.1).

14.2.3. Определение задачи (в новой формулировке)

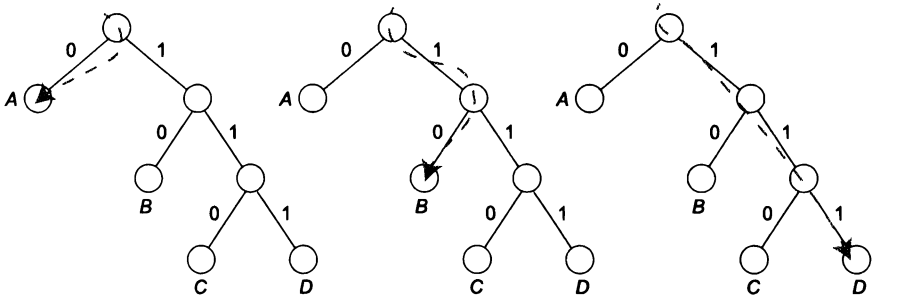
Сформулируем конкретнее задачу оптимального беспрефиксного кода. Под Σ-*деревом* мы подразумеваем бинарное дерево, в котором листья помечены в однозначном соответствии с символами алфавита Σ, при этом двоичные беспрефиксные коды для алфавита Σ соответствуют $\sum$-деревьям. Для Σ-дерева T и частоты символов $\mathbf{p} = \{p_a\}_{a\in\Sigma}$ обозначим через $L(T, \mathbf{p})$ среднюю глубину листа в T, причем вклад каждого листа взвешивается в соответствии с частотой его метки:

$$L(T,\mathbf{p}) = \sum_{a\in\Sigma} p_a \cdot (\text{глубина листа, помеченного } a \text{ в } T). \qquad (14.1)$$

Из утверждения 14.1 вытекает, что $L(T, \mathbf{p})$ является в точности средней длиной кодирования кода, который соответствует T, что мы и хотим минимизировать.

---

[1] Мы можем допустить, что каждый лист дерева имеет метку, так как удаление непомеченных листьев не изменяет код, определенный деревом.



а) Прохождение №1 (*"A"*) *б)* Прохождение № 2 (*"B"*) *в)* Прохождение № 3 (*"D"*)

Рис. 14.1. Декодирование строки «010111» в «ABD» многократными обходами от корня к листу

Следовательно, мы можем переформулировать задачу оптимального беспрефиксного кода как задачу исключительно бинарных деревьев.

ЗАДАЧА: ОПТИМАЛЬНЫЕ БЕСПРЕФИКСНЫЕ КОДЫ (В НОВОЙ ФОРМУЛИРОВКЕ)

Вход: неотрицательная частота p_a для каждого символа a алфавита Σ размера $n \geq 2$.

Выход: Σ-дерево с минимально возможной средней глубиной листа (14.1).

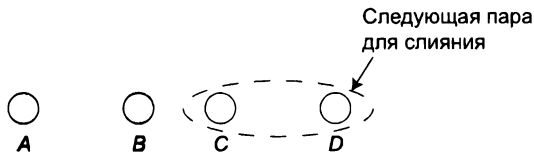
14.3. Жадный алгоритм Хаффмана

14.3.1. Построение деревьев путем последовательных слияний

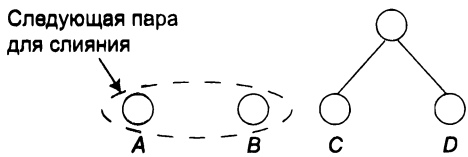
Суть идеи Хаффмана в том, чтобы решить задачу оптимального беспрефиксного кода, используя восходящий подход (снизу вверх)[1]. «Восходящий»

[1] Как в случае с курсовой работой Дэвида А. Хьюмена. Решение заменило неоптимальный алгоритм «сверху вниз, разделяй и властвуй», ранее изобретенный Робертом М. Фано.

означает начать с n узлов (где n — это размер алфавита Σ), каждый из которых помечен другим символом Σ, и строить дерево путем последовательных слияний. Например, если $\Sigma = \{A, B, C, D\}$, то построение начинается с листьев:

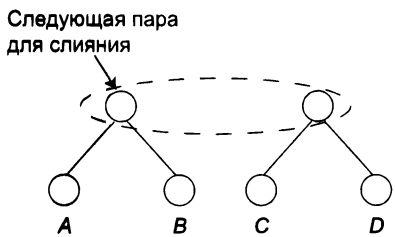


Наше первое слияние могло бы состоять из узлов, помеченных «*C*» и «*D*», реализованное путем введения одного нового немеченого внутреннего узла с левым и правым отпрысками, которые в указанном порядке соответствуют *C* и *D*:

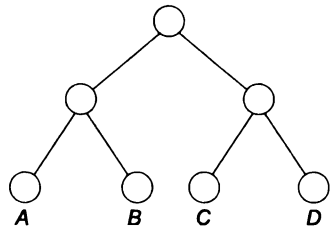


По сути, это слияние берет дерево, в котором листья, помеченные как «*C*» и «*D*», являются сестрами (то есть имеют общего родителя).

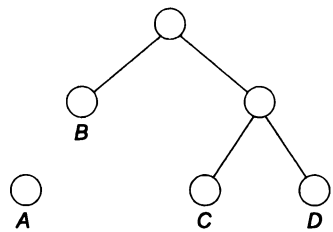
Затем мы можем сделать то же самое с *A* и *B*, используя дерево, в котором листья, помеченные как «*A*» и «*B*», являются сестрами:



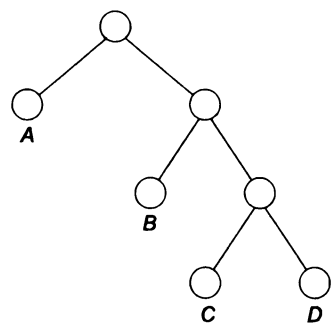
Теперь осталось слить воедино только две группы и получить полноценное бинарное дерево:



Это бинарное дерево — копия того, что использовалось для представления кода фиксированной длины в разделе 14.2.1. Как вариант, во второй итерации мы могли бы слить узел, помеченный как «*B*», с деревом, содержащим «*C*» и «*D*»:



В результате заключительного слияния образуется бинарное дерево, используемое для представления беспрефиксного кода переменной длины в разделе 14.2.1:



В общем случае жадный алгоритм Хаффмана поддерживает *лес*, то есть коллекцию одного или нескольких бинарных деревьев, листья которых находятся

в однозначном соответствии с символами $\sum$. Каждая итерация алгоритма выбирает два дерева в текущем лесу и объединяет их, делая их корни левыми и правыми дочерними элементами нового немеченого внутреннего узла. Алгоритм останавливается, когда остается только одно дерево.

УПРАЖНЕНИЕ 14.3

Сколько слияний будет выполнять жадный алгоритм Хаффмана до остановки? (Пусть $n = |\Sigma|$ обозначает число символов[1].)

а) $n - 1$

б) n

в) $\frac{(n+1)n}{2}$

г) для ответа недостаточно информации

(Решение и пояснение см. в разделе 14.3.7.)

14.3.2. Жадный критерий Хаффмана

Какую пару деревьев мы должны объединять посредством очередной итерации для заданного множества частот символов $\{p_a\}_{a\in\Sigma}$? Каждое слияние увеличивает глубину всех листьев двух участвующих деревьев и, следовательно, длины кодирования соответствующих символов. Например, в приведенном выше предпоследнем слиянии глубина узлов, помеченных как «*C*» и «*D*», увеличивается с 1 до 2, а глубина узла, помеченного как «*B*», увеличивается с 0 до 1. Таким образом, каждое слияние увеличивает целевую функцию, которую мы хотим минимизировать: среднюю глубину листа (14.1). Каждая итерация жадного алгоритма Хаффмана близоруко выполняет слияние, которое увеличивает эту целевую функцию в наименьшей степени.

ЖАДНЫЙ КРИТЕРИЙ ХАФФМАНА

Слияние пары деревьев приводит к минимально возможному увеличению средней глубины листа.

[1] $|S|$ для конечного множества S обозначает число элементов S.

Для каждого символа a в одном из двух участвующих деревьев глубина соответствующего листа увеличивается на 1, а вклад соответствующего члена в сумму (14.1) увеличивается на p_a. Таким образом, слияние двух деревьев T_1 и T_2 увеличивает среднюю глубину листа на сумму частот участвующих символов:

$$\sum_{a \in T_1} p_a + \sum_{a \in T_2} p_a, \tag{14.2}$$

где суммирование производится по всем символам алфавита, для которых соответствующий лист принадлежит соответственно T_1 или T_2. Жадный критерий Хаффмана подразумевает объединение пар деревьев, для которых сумма (14.2) будет как можно меньше.

14.3.3. Псевдокод

Алгоритм Хаффмана строит Σ-дерево снизу вверх, и на каждой итерации он объединяет два дерева, имеющие наименьшие суммы частот соответствующих символов.

HUFFMAN

Вход: неотрицательная частота p_a для каждого символа a алфавита Σ.

Выход: Σ-дерево с минимальной средней глубиной листа, представляющее двоичный беспрефиксный код с минимальной средней длиной кодирования.

```
// Инициализация
for each a ∈ Σ do
    T_a := дерево, содержащее один узел, помеченный как «a»
    P(T_a) := p_a
F := {T_a}_{a∈Σ}  // инвариант: ∀T ∈ F, P(T) = P(T) = Σ_{a∈T} p_a

// Главный цикл
while F содержит по крайне мере два дерева do
```

$T_1 := \text{argmin}_{T \in F} P(T)$ // сумма минимальных частот
$T_2 := \text{argmin}_{T \in F,\, T \neq T1} P(T)$ // вторая наименьшая
удалить T_1 и T_2 из F
// корни T_1, T_2 становятся левыми, правыми
// потомками нового внутреннего узла
$T_3 :=$ слияние T_1 и T_2
$P(T_3) := P(T_1) + P(T_2)$ // поддерживает инвариант
Добавить T_3 в F
return уникальное дерево в $\mathcal{F}$

О ПСЕВДОКОДЕ

В этой книжной серии объясняются алгоритмы, использующие комбинацию высокоуровневого псевдокода и русского языка (как показано выше).

Предполагается, что у читателей есть навыки трансляции таких высокоуровневых (общих) описаний в рабочий код на любимом языке программирования. Несколько других книг и ресурсов в Сети предлагают конкретные реализации различных алгоритмов на определенных языках программирования.

Во-первых, преимущество акцентирования высокоуровневых описаний над реализациями, специфичными для конкретного языка, заключается в гибкости: так, наличие осведомленности не увязывается с *каким-либо* конкретным языком программирования. Во-вторых, этот подход способствует пониманию алгоритмов на глубоком, концептуальном уровне, не обремененном второстепенными деталями. Опытные программисты и специалисты в области информатики обычно мыслят и обмениваются информацией об алгоритмах на столь же высоком уровне.

Тем не менее ничто не может заменить подробного понимания алгоритма, которое вытекает из разработки читателями собственной рабочей реализации. Настоятельно рекомендуется реализовать столько алгоритмов из этой книги, сколько позволяет время. Это также отличный повод познакомиться с новым языком программирования и его возможностями (см. раздел задач по программированию в конце главы и сопроводительные тестовые сценарии).

14.3.4. Пример

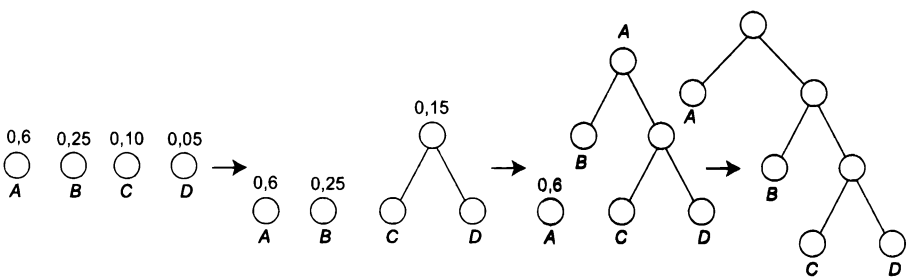
Давайте вернемся к нашему четырехсимвольному алфавиту со следующими ниже частотами:

Символ	Частота
A	0,60
B	0,25
C	0,10
D	0,05

Первоначально алгоритм `Huffman` создает лес из четырех деревьев — T_A, T_B, T_C, T_D, каждое из которых содержит один узел, помеченный другим символом алфавита. Первая итерация алгоритма объединяет узлы, соответствующие двум символам с наименьшими частотами — в данном случае «*C*» и «*D*». После итерации лес алгоритма содержит только три дерева со следующими суммами частот символов:

Символ	Сумма частот символов
Дерево, содержащее *A*	0,60
Дерево, содержащее *B*	0,25
Дерево, содержащее *C* и *D*	0,05 + 0,10 = 0,15

Вторые два дерева имеют наименьшие суммы частот символов, поэтому именно эти деревья объединяются на второй итерации. На третьей итерации лес *F* содержит только два дерева; они объединяются воедино, давая конечный результат, то есть именно то дерево, которое используется для представления беспрефиксного кода переменной длины в разделе 14.2.1:

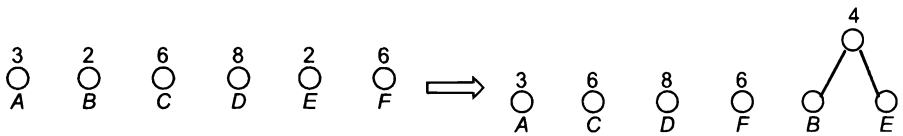


14.3.5. Более крупный пример

Для лучшего понимания функционирования жадного алгоритма Хаффмана посмотрим, как образуется окончательное дерево, на более подробном примере:

Символ	Частота
A	3
B	2
C	6
D	8
E	2
F	6

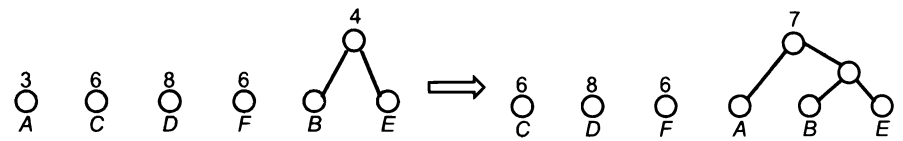
Если вас беспокоит, что частоты символов в сумме не составляют 1, то вы можете свободно поделить каждое из них на 27 — это не меняет суть задачи. Как обычно, первый шаг объединяет два символа с наименьшими частотами, а именно «*B*» и «*E*»:



В лесу останется пять деревьев:

Символ	Сумма частот символов
Дерево, содержащее *A*	3
Дерево, содержащее *C*	6
Дерево, содержащее *D*	8
Дерево, содержащее *F*	6
Дерево, содержащее *B* и *E*	2 + 2 = 4

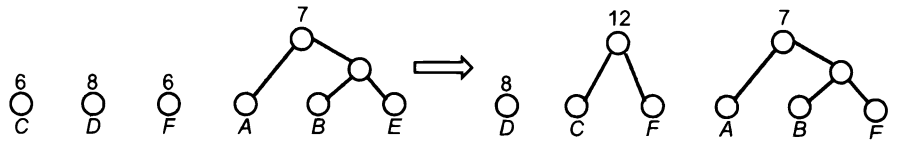
и теперь алгоритм объединяет первое и последнее из них:



Останется четыре дерева:

Символ	Сумма частот символов
Дерево, содержащее *C*	6
Дерево, содержащее *D*	8
Дерево, содержащее *F*	6
Дерево, содержащее *A*, *B* и *E*	4 + 3 = 7

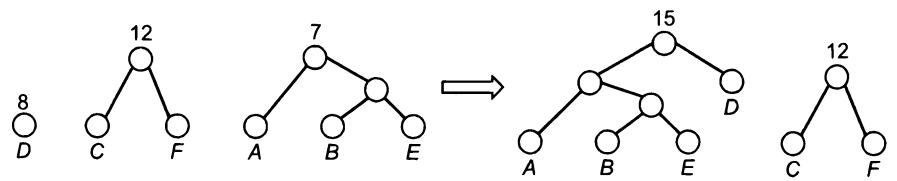
Далее алгоритм объединяет узлы, помеченные как «*C*» и «*F*»:



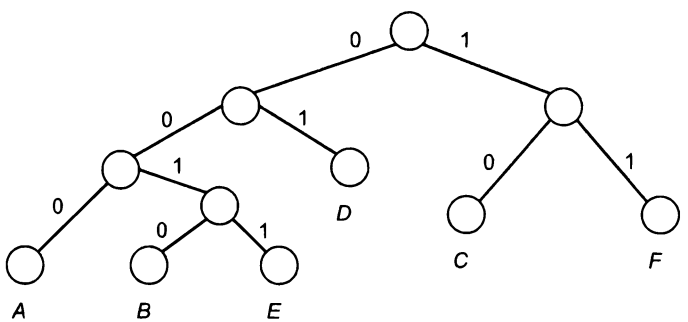
Останется три дерева:

Символ	Сумма частот символов
Дерево, содержащее *D*	8
Дерево, содержащее *C* и *F*	6 + 6 = 12
Дерево, содержащее *A*, *B* и *E*	7

Предпоследним является слияние первого и третьего деревьев:



и окончательное слияние производит результат алгоритма на выходе:



Это дерево соответствует следующему ниже беспрефиксному коду:

Символ	Кодирование
A	000
B	0010
C	10
D	01
E	0011
F	11

14.3.6. Время выполнения

Простая реализация алгоритма `Huffman` выполняется за время $O(n^2)$, где n — это число символов. Как отмечено в упражнении 14.3, каждое слияние уменьшает число деревьев в F на 1, приводя к $n-1$ итерациям главного цикла. Каждая итерация отвечает за выявление двух текущих деревьев с наименьшими суммами частот символов; это может быть сделано путем исчерпывающего поиска по $O(n)$ деревьям леса F. Остальная часть работы — инициализация, обновление F, перезапись указателей при слиянии двух деревьев — вносит в совокупную границу времени выполнения только $O(n)$ операций при суммарном значении $O(n^2)$.

Читатели, знакомые с кучевой структурой данных (см. главу 10 *части 2* и раздел 15.3), должны найти возможность добиться лучшего результата. Смысл кучевой структуры данных состоит в том, чтобы ускорять многократные минимальные вычисления так, чтобы каждое вычисление занимало логарифмическое, а не линейное время. В работе, выполняемой на каждой итерации главного цикла в алгоритме `Huffman`, доминируют два минимальных вычисления, поэтому вас должно осенить: этот алгоритм требует кучи! Использование кучи для ускорения минимальных вычислений сокращает время выполнения с $O(n^2)$ до $O(n \log n)$, что квалифицируется как невероятно быстрая реализация[1].

[1] Объекты в куче соответствуют деревьям леса F. Ключ, ассоциированный с объектом, представляет собой сумму частот символов, соответствующих листьям дерева. На каждой итерации деревья T_1 и T_2 могут быть удалены из кучи с помощью двух подряд операций Извлечь минимум, а объединенное дерево T_3 добавляется одной операцией Вставить (с ключом дерева T_3, установленным равным сумме ключей T_1 и T_2).

Впрочем, алгоритм `Huffman` может быть реализован путем сортировки символов в порядке возрастания частоты с последующим выполнением линейного объема дополнительной обработки. Эта реализация избегает кучи в пользу еще более простой структуры данных: очереди (фактически двух очередей, подробнее см. в задаче 14). n символов могут быть отсортированы по частоте за время $O(n \log n)$ (см. сноску 1 на с. 24), поэтому время выполнения реализации равно $O(n \log n)$. Кроме того, в частных случаях, когда сортировка возможна за линейное время, эта реализация алгоритма `Huffman` также будет выполняться за линейное время[1].

14.3.7. Решение упражнения 14.3

Правильный ответ: (а). Первоначальный лес имеет n деревьев, где n — это число символов алфавита. Каждое слияние заменяет пару деревьев одним деревом и, следовательно, уменьшает число деревьев на 1. Алгоритм останавливается, как только остается одно дерево, то есть после $n - 1$ слияний.

*14.4. Доказательство правильности

Алгоритм `Huffman` правильно решает задачу оптимального беспрефиксного кода[2].

Теорема 14.2 (правильность алгоритма `Huffman`). *Для каждого алфавита* Σ *и неотрицательных частот символов* $\{p_a\}_{a \in \Sigma}$ *алгоритм* `Huffman` *выводит беспрефиксный код с минимально возможной средней длиной кодирования.*

---

[1] Для алгоритма сортировки «общего назначения», свободного от предположений относительно данных, подлежащих сортировке, наилучшим достижимым временем выполнения задания следует считать $O(n \log n)$. Впрочем, некоторые специализированные алгоритмы сортировки могут работать быстрее. Например, алгоритм RadixSort способен справиться с сортировкой массива данных, состоящего только из целых чисел величиной не более n^{10}, за время, равное $O(n)$. Подробнее см. раздел 5.6 *части 1*.

[2] Разделы, отмеченные здесь и далее *, более сложны для усвоения и могут быть пропущены при первом чтении.

Аналогичным образом указанный алгоритм выводит и Σ-дерево с минимально возможной средней глубиной листа (14.1).

14.4.1. Высокоуровневый план

Доказательство теоремы 14.2 смешивает две общепринятые стратегии доказательства правильности жадных алгоритмов, упомянутые в разделе 13.4: индукцию и обмен аргументами.

Мы начнем с индукции по размеру алфавита, сконцентрировавшись на двух идеях, необходимых для реализации индукционного шага. Зафиксируем с этого момента вход с алфавитом Σ и символьными частотами **p**. Пусть a и b обозначают символы с наименьшими и вторыми наименьшими частотами соответственно. Рассмотрим первую итерацию алгоритма `Huffman`, в которой он объединяет листья, соответствующие a и b. Затем указанный алгоритм практически взял Σ-дерево, в котором листья, соответствующие a и b, являются элементами одного уровня. Прежде всего следует доказать, что среди всех таких деревьев алгоритм `Huffman` на выходе выдает наилучшее.

ГЛАВНАЯ ИДЕЯ № 1

Доказать, что выход алгоритма `Huffman` минимизирует среднюю глубину листа по всем Σ-деревьям, в которых a и b являются элементами одного уровня.

Этот шаг сводится к тому, чтобы показать, что задача вычисления лучшего Σ-дерева, в котором a и b являются элементами одного уровня, эквивалентна задаче вычисления лучшего Σ′-дерева, где Σ′ совпадает с Σ, за исключением того, что a и b слиты в единый символ. Поскольку алфавит Σ′ меньше, чем Σ, мы можем завершить доказательство с помощью индукции.

Этой идеи, впрочем, недостаточно. Если каждое Σ-дерево с a и b в качестве элементов одного уровня является субоптимальным, то и не стоит их оптимизировать.

Вторая главная идея решает эту проблему и доказывает, что всегда безопасно брать дерево, в котором два символа с самой низкой частотой соответствуют одноуровневым листьям.

ГЛАВНАЯ ИДЕЯ № 2

Доказать, что существует оптимальное Σ-дерево, в котором a и b являются элементами одного уровня.

В данном случае необходимо показать, что каждое Σ-дерево может быть введено в одинаково хорошее или лучшее Σ-дерево, в котором a и b являются элементами одного уровня, путем перестановки местами меток a и b с метками x и y двух листьев на самом глубоком уровне дерева. Интуитивно это воспринимается как чистый выигрыш, понижающий символы a и b с меньшей частотой до самого глубокого уровня дерева и продвигающий высокочастотные символы x и y ближе к корню.

Если обе главные идеи могут быть реализованы, то индукционный шаг и теорема 14.2 легко выполняются. Из первой идеи следует, что алгоритм `Huffman` оптимально решает проблему для ограниченного семейства Σ-деревьев, в которых a и b являются элементами одного уровня. Вторая гарантирует, что оптимальное дерево этого ограниченного типа фактически является оптимальным для первоначальной задачи.

14.4.2. Подробности

Обзор индукции

Для формального доказательства обратимся к нашему старому другу (или Немезиде?), то есть к индукции[1]. Напомним, доказательства по индукции следуют довольно жесткому шаблону с целью подтверждения того, что логическое утверждение $P(k)$ соблюдается для сколь угодно больших натуральных чисел k. В доказательстве теоремы 14.2 мы принимаем $P(k)$ как утверждение:

[1] Для ознакомления с вводным курсом обучения см. приложение А (*часть 1*) или книгу «Математика для computer science», упомянутую в предисловии.

«Алгоритм `Huffman` правильно решает задачу оптимального беспрефиксного кода, когда размер алфавита не превышает k».

Аналогично рекурсивному алгоритму, доказательство по индукции состоит из двух частей: *базового случая* и *индукционного шага*. Для нас естественным базовым случаем является утверждение $P(2)$ (задача оптимального беспрефиксного кода неинтересна с односимвольным алфавитом). На индукционном шаге мы допускаем, что $k > 2$. Мы также допускаем, что все $P(2)$, $P(3)$, ..., $P(k-1)$ являются истинными — это называется *индукционной гипотезой*, — и используем это допущение для доказательства того, что $P(k)$, следовательно, также является истинным. Если мы докажем и базовый случай, и индукционный шаг, то $P(k)$ действительно будет истинным для каждого положительного целого числа $k \geq 2$: $P(2)$ истинно по базовому случаю и, как падающие домино, применение индукционного шага снова и снова показывает, что $P(k)$ является истинным для сколь угодно больших значений k.

Алгоритм `Huffman` является оптимальным для двухсимвольных алфавитов: указанный алгоритм использует 1 бит для кодирования каждого символа (0 для одного символа и 1 для другого), что является минимально возможным. Это доказывает базовый случай.

Для индукционного шага допустим, что $k > 2$, задав алфавит Σ размера k и неотрицательные частоты символов $\mathbf{p} = \{p_x\}_{x \in \Sigma}$. Для остальной части доказательства обозначим через a и b два символа Σ, соответственно с наименьшей и второй наименьшей частотами, разрывая совпадения значений произвольным образом.

Первая главная идея в новой формулировке

Чтобы реализовать первую, более сложную главную идею из раздела 14.4.1, определим

$$\mathcal{T}_{ab} = \left\{ \begin{array}{c} \Sigma\text{-деревья, в которых } a \text{ и } b \text{ являются соответственно} \\ \text{левым и правым потомком общего родителя} \end{array} \right\}.$$

Алгоритм `Huffman` на выходе выдает дерево $\mathcal{T}_{ab}$, и мы хотим доказать, что это дерево является лучшим таким деревом:

(*) среди всех деревьев в $\mathcal{T}_{ab}$ алгоритм `Huffman` выдает дерево с минимально возможной средней глубиной листа.

Напомним, средняя глубина листа Σ-дерева T относительно частот символов $\mathbf{p}$ равна

$$L(T,\mathbf{p}) = \sum_{x\in\Sigma} p_x \cdot \{\text{глубина помеченного в листьях } x \text{ в } T\}.$$

Эта величина совпадает со средней длиной кодирования соответствующего беспрефиксного кода.

Применение индукционной гипотезы к остаточной задаче

Индукционная гипотеза применима только к алфавитам с менее чем k символами. Выведем производную Σ′ из Σ, объединив символы a и b — символы с наименьшими и вторыми наименьшими частотами — в единый «метасимвол» ab. Существует взаимно однозначное соответствие между Σ′-деревьями и ограниченным множеством $\mathcal{T}_{ab}$ Σ-деревьев (рис. 14.2). Каждое Σ′-дерево

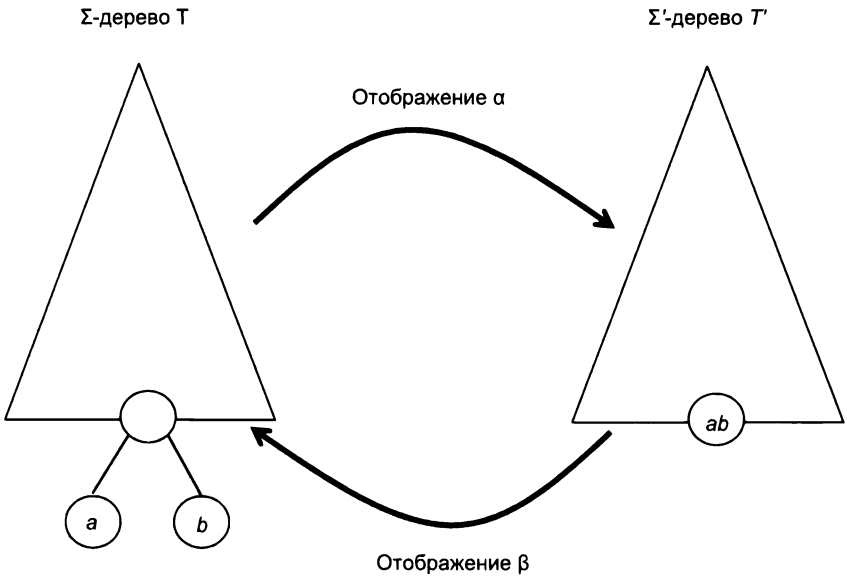


Рис. 14.2. Существует взаимно однозначное соответствие между Σ′-деревьями и Σ-деревьями, в которых a и b являются левыми и правыми дочерними элементами общего родителя

T' может быть преобразовано в Σ-дерево $T \in \mathcal{T}_{ab}$ путем замены помеченного в листе «*ab*» на непомеченный узел, который имеет дочерние элементы, помеченные как «*a*» и «*b*». Обозначим это отображение $T' \to T$ через $\beta(T')$. И, наоборот, каждое дерево $T \in \mathcal{T}_{ab}$ может быть превращено в Σ'-дерево T', втянув листья, помеченные как a и b, в их (общего) родителя и пометив полученный «метаузел» как «*ab*». Обозначим это обратное отображение $T \to T'$ через $\alpha(T)$.

Частоты $\mathbf{p}' = \{p'_x\}_{x \in \Sigma'}$, которые мы присваиваем символам Σ', совпадают с Σ, за исключением частоты p'_{ab} нового символа ab, определенной как сумма $p_a + p_b$ частот двух символов, которые он представляет.

Первая итерация алгоритма `Huffman` объединяет листья, помеченные как a и b, после чего рассматривает их как неделимую единицу с суммарной частотой $p_a + p_b$. Это означает, что конечный результат алгоритма является таким же, как если бы он был перезапущен с нуля с входами Σ' и $\mathbf{p}'$, а результирующее Σ'-дерево транслируется отображением β обратно в Σ-дерево $\mathcal{T}_{ab}$.

Утверждение 14.3 (сохранение поведения алгоритма `Huffman`). *Выход алгоритма* `Huffman` *со входом* Σ *и* $\mathbf{p}$ *равен* $\beta(T')$, *где* T' *— это выход алгоритма* `Huffman` *со входом* Σ' *и* $\mathbf{p}'$.

Дополнительные свойства соответствия

Соответствие между Σ'-деревьями и Σ-деревьями в $\mathcal{T}_{ab}$, заданное отображениями α и β (рис. 14.2), также сохраняет среднюю глубину листа вплоть до константы, которая не зависит от выбора дерева.

Утверждение 14.4 (сохранение средней глубины листа). *Для каждого* Σ*-дерева* T *из* $\mathcal{T}_{ab}$ *с символьными частотами* $\mathbf{p}$ *и соответствующего* Σ'*-дерева* $T' = \alpha(T)$ *и символьных частот* $\mathbf{p}'$

$$L(T, \mathbf{p}) = L(T', \mathbf{p}') + \underbrace{p_a + p_b}_{\text{независимые от } T}.$$

Доказательство: листья T, не помеченные как a или b, занимают одну и ту же позицию в T'. Их символы имеют одинаковые частоты в $\mathbf{p}$ и $\mathbf{p}'$, поэтому эти листья вносят одинаковый вклад в среднюю глубину листьев обоих деревьев.

Суммарная частота a и b в $\mathbf{p}$ является такой же, что и у ab в $\mathbf{p}'$, но глубина соответствующих листьев на единицу больше. Таким образом, вклад a и b в среднюю глубину листа T больше $p_a + p_b$, чем вклад ab в среднюю глубину листа T'. *Ч. Т. Д.*

Поскольку соответствие между Σ'-деревьями и Σ-деревьями в $\mathcal{T}_{ab}$ сохраняет среднюю глубину листа (до независимой от дерева константы $p_a + p_b$), оно ассоциирует оптимальное Σ'-дерево с оптимальным Σ-деревом в $\mathcal{T}_{ab}$:

$$\text{наилучшее } \Sigma\text{-дерево в } T_{ab} \underset{\beta}{\overset{\alpha}{\rightleftharpoons}} \text{наилучшее } \Sigma'\text{-дерево}$$

$$\text{второе наилучшее } \Sigma\text{-дерево в } T_{ab} \underset{\beta}{\overset{\alpha}{\rightleftharpoons}} \text{второе наилучшее } \Sigma'\text{-дерево}$$

$$\vdots$$

$$\text{наихудшее } \Sigma\text{-дерево в } T_{ab} \underset{\beta}{\overset{\alpha}{\rightleftharpoons}} \text{наихудшее } \Sigma'\text{-дерево}$$

Следствие 14.5 (сохранение оптимальных решений). *Σ'-дерево T^* минимизирует $L(T', \mathbf{p}')$ над всеми Σ'-деревьями T' тогда и только тогда, когда соответствующее Σ-дерево $\beta(T^*)$ минимизирует $L(T, \mathbf{p})$ над всеми Σ-деревьями T в $\mathcal{T}_{ab}$.*

Реализация первой главной идеи

Теперь все подготовлено для доказательства утверждения (*), что среди всех деревьев $\mathcal{T}_{ab}$ алгоритм `Huffman` выводит дерево с минимально возможной средней глубиной листа:

1. Согласно утверждению 14.3, выход алгоритма `Huffman` со входом Σ и $\mathbf{p}$ равен $\beta(T')$, где T' является выходом алгоритма `Huffman` с входными Σ' и $\mathbf{p}'$.
2. Поскольку $|\Sigma'| < k$, из индукционной гипотезы вытекает, что выход T' алгоритма `Huffman` с входными Σ' и $\mathbf{p}'$ является оптимальным.
3. Исходя из следствия 14.5, Σ-дерево $\beta(T')$ является оптимальным для первоначальной задачи с входными Σ и $\mathbf{p}$.

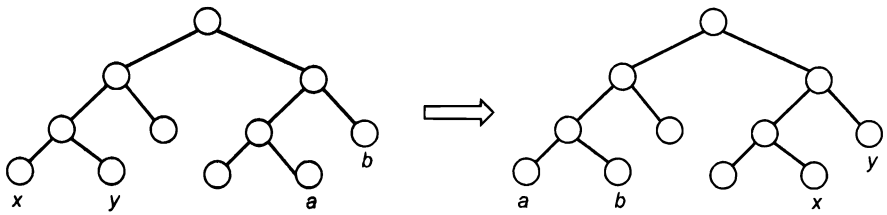
Реализация второй главной идеи

Вторая часть индукционного шага проще и основана на обмене аргументами. Здесь мы хотим доказать, что алгоритм `Huffman` не ошибся, взяв дерево,

в котором два символа наименьшей частоты являются элементами одного уровня:

> (†) существует дерево $\mathcal{T}_{ab}$, которое минимизирует среднюю глубину листа $L(T, \mathbf{p})$ по всем Σ-деревьям T.

Чтобы доказать (†), рассмотрим произвольное Σ-дерево T. Мы можем завершить доказательство, показав дерево $T^* \in \mathcal{T}_{ab}$, в котором a и b являются элементами одного уровня, причем $L(T^*, \mathbf{p}) \le L(T, \mathbf{p})$. Без потери общности каждый узел T либо является листом, либо имеет двух потомков[1]. Таким образом, существует два листа с общим родителем, которые населяют самый глубокий уровень T, скажем, с левым дочерним x и правым дочерним y[2]. Возьмем Σ-дерево $T^* \in \mathcal{T}_{ab}$ путем перестановки меток листьев, помеченных a и x, и меток листьев, помеченных b и y:



Как изменяется средняя глубина листа? Расширив определение (14.1) и аннулировав члены, которые соответствуют листьям, отличным от a, b, x, y, мы имеем

$$L(T) - L(T^*) = \sum_{z \in \{a,b,x,y\}} p_z \times (\textit{глубина}\, z \;\textit{в}\; T - \textit{глубина}\; z \;\textit{в}\; T^*).$$

Глубины в T^* могут быть переписаны с точки зрения глубин в T. Например, глубина a в T^* совпадает с глубиной x в T, глубина y в T^* совпадает

---

[1] Для формирования другого Σ-дерева с меньшей средней глубиной листа внутренний узел с единственным дочерним элементом может быть сращен.

[2] В упрощенном варианте x и y можно рассматривать как отличающиеся от a и b, но доказательство истинно и для случаев перекрывания $\{x, y\}$ и $\{a, b\}$.

с глубиной b в T и т. д. Поэтому мы можем незаметно переставить члены, получив

$$L(T) - L(T^*) = \underbrace{(p_x - p_a)}_{\geq 0} \times \underbrace{(\textit{глубина } x \textit{ в } T - \textit{глубина } a \textit{ в } T)}_{\geq 0} +$$
$$+ \underbrace{(p_y - p_b)}_{\geq 0} \times \underbrace{(\textit{глубина } y \textit{ в } T - \textit{глубина } b \textit{ в } T)}_{\geq 0} \geq 0.$$

Вследствие перестановки разница в левой стороне является неотрицательной: $p_x - p_a$ и $p_y - p_b$ являются неотрицательными, потому что a и b были выбраны как символы с наименьшими частотами, а два других условия в правой части являются неотрицательными, потому что x и y были выбраны из самого глубокого уровня дерева T. Значит, средняя глубина листьев $T^* \in \mathcal{T}_{ab}$ не больше глубины T. Поскольку дерево $\mathcal{T}_{ab}$ равняется каждому Σ-дереву либо их превосходит, $\mathcal{T}_{ab}$ содержит дерево, которое является оптимальным среди всех Σ-деревьев. Этим завершается доказательство (†).

Из утверждения (*) следует, что с входными Σ и **p** алгоритм `Huffman` на выходе выдает наилучшее дерево из ограниченного множества $\mathcal{T}_{ab}$. По (†) это дерево должно быть оптимальным для первоначальной задачи. Этим завершается доказательство индукционного шага и теоремы 14.2. *Ч. Т. Д.*

ВЫВОДЫ

- **Беспрефиксные двоичные коды переменной длины могут иметь меньшие средние длины кодирования, чем коды фиксированной длины, когда разные символы алфавита имеют разные частоты.**
- **Беспрефиксные коды могут быть визуализированы как бинарные деревья, в которых листья находятся во взаимно однозначном соответствии с символами алфавита. Кодировки соответствуют путям от корня к листу, где левое и правое дочерние ребра интерпретируются соответственно как нули и единицы, тогда как средняя длина кодирования соответствует средней глубине листа.**

- ✶ Жадный алгоритм Хаффмана поддерживает лес, где листья находятся в соответствии с символами алфавита, и на каждой итерации жадно выполняет слияние пары деревьев, вызывая минимально возможное увеличение средней глубины листа.
- ✶ Алгоритм Хаффмана гарантированно вычисляет беспрефиксный код с минимально возможной средней длиной кодирования.
- ✶ Алгоритм Хаффмана может быть реализован с работой за время $O(n \log n)$, где n — это число символов.
- ✶ Доказательство правильности использует обмен аргументами, показывая существование оптимального решения, в котором два символа наименьшей частоты являются сестрами, и индукцию, показывая, что указанный алгоритм вычисляет такое решение.

Задачи на закрепление материала

Задача 14.1. Рассмотрите следующие частоты символов для пятисимвольного алфавита:

Символ	Частота
A	0,32
B	0,25
C	0,2
D	0,18
E	0,05

Какова средняя длина кодирования оптимального беспрефиксного кода?

а) 2,23

б) 2,4

в) 3

г) 3,45

Задача 14.2. Рассмотрите следующие частоты символов для пятисимвольного алфавита:

Символ	Частота
A	0,16
B	0,08
C	0,35
D	0,07
E	0,34

Какова средняя длина кодирования оптимального беспрефиксного кода?

а) 2,11

б) 2,31

в) 2,49

г) 2,5

Задача 14.3. Какое максимальное число бит может использовать жадный алгоритм Хаффмана для кодирования одного символа? (Как обычно, $n = |\Sigma|$ обозначает размер алфавита.)

а) $\log_2 n$

б) $\ln n$

в) $n - 1$

г) n

Задача 14.4. Какие из следующих утверждений о жадном алгоритме Хаффмана являются истинными? Будем считать, что частота символов равна 1. (Выберите все, что применимо.)

а) Буква с частотой не менее 0,4 никогда не будет кодироваться двумя или более битами.

б) Буква с частотой не менее 0,5 никогда не будет кодироваться двумя или более битами.

в) Если все частоты символов меньше 0,33, то все символы будут кодироваться по меньшей мере двумя битами.

г) Если все частоты символов меньше 0,5, то все символы будут кодироваться по крайней мере двумя битами.

Сложные задачи

Задача 14.5. Приведите реализацию жадного алгоритма Хаффмана, который использует один вызов подпрограммы сортировки, за которым следует линейный объем дополнительной работы.

Задачи по программированию

Задача 14.6. Реализуйте на своем любимом языке программирования алгоритм Хаффмана из раздела 14.3 для задачи оптимального беспрефиксного кода. Насколько быстрее реализация на основе кучи (описанная в сноске на с. 66), чем простая квадратично-временная реализация?[1] Насколько реализация в задаче 14.5 быстрее, чем реализация на основе кучи? Тестовые случаи и наборы данных для сложных задач см. на веб-сайте www.algorithmsilluminated.org.

[1] Проверьте, встроена ли структура данных кучи в ваш любимый язык программирования (например, для Java — в класс `PriorityQueue`).

15 Минимальные остовные деревья

В этой главе парадигма проектирования жадного алгоритма применяется к знаменитой задаче графа — задаче о *минимальном остовном дереве* (minimum spanning tree, MST)[1]. Задача о минимальном остовном дереве — это уникальная площадка для изучения жадных алгоритмов, в которой почти любой жадный алгоритм, который вы можете придумать, оказывается правильным. После рассмотрения графов и формального определения задачи (раздел 15.1) мы обсудим два наиболее известных алгоритма MST — алгоритм Прима (раздел 15.2) и алгоритм Краскала (раздел 15.5). Оба алгоритма допускают невероятно быстрые реализации, используя такие структуры данных, как куча и система непересекающихся множеств (Union-Find). В разделе 15.8 описывается применение алгоритма Краскала в машинном обучении для кластеризации с одиночной связью.

15.1. Определение задачи

Задача о минимальном остовном дереве заключается в соединении группы объектов наиболее дешевым из возможных способом. Объекты и соединения могут представлять что-то физическое, как компьютерные серверы и каналы связи между ними. Либо, возможно, каждый объект является представлением документа (скажем, как вектор частот слов), где соединениям соответствуют пары «похожих» документов. Задача возникает естественным образом в нескольких прикладных областях, включая компьютерное сетевое взаимодействие (попробуйте поискать во Всемирной паутине «протокол покрывающего дерева», Spanning Tree Protocol (STP), где оно также именуется связующим) и машинное обучение (см. раздел 15.8).

15.1.1. Графы

Объекты и соединения между ними наиболее естественно моделируются графами. *Граф* $G = (V, E)$ имеет две составляющие: множество V *вершин* и множество E *ребер* (рис. 15.1). В этой главе рассматриваются только *неориентированные* графы, в которых каждое ребро E является неупорядоченной

[1] Существует аналог задачи MST для ориентированных графов (так называемая «задача минимальной стоимости древа» или «задача оптимального ветвления»). Быстрые алгоритмы для ее решения в рамках этой серии книг не рассматриваются.

парой $\{v, w\}$ вершин (записываемых как $e = (v, w)$ или $e = (w, v)$), которые называются *конечными точками* ребра. Числа $|V|$ и $|E|$ вершин и ребер обычно обозначаются соответственно через n и m.

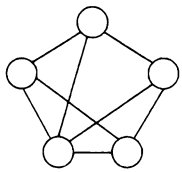


Рис. 15.1. Неориентированный граф с пятью вершинами и восемью ребрами

Графы могут кодироваться различными способами для использования в алгоритме. В этой главе мы исходим из того, что входной граф представлен списками смежности с массивом вершин, массивом ребер, указателями от каждого ребра к его двум конечным точкам и указателями от каждой вершины к ее инцидентным ребрам[1].

15.1.2. Остовные деревья

Входом в задачу о минимальном остовном дереве является неориентированный граф $G = \{V, E)$, в котором каждое ребро E имеет вещественную стоимость c_e. Например, c_e может указывать стоимость соединения двух компьютерных серверов. Цель в том, чтобы вычислить остовное дерево графа с минимально возможной суммой реберных стоимостей. Под *остовным (связующим) деревом* графа G мы имеем в виду подмножество $T \in E$ ребер, которое удовлетворяет двум свойствам. Во-первых, T не должно содержать цикл («древесная» часть)[2]. Во-вторых, для каждой пары $v, w \in V$ вершин T должно включать путь между v и w («связующая» часть)[3].

---

[1] Подробнее о графиках и их представлениях см. главу 7 *части 2*.

[2] Цикл в графе $G = (V, E)$ — путь, возвращающийся к месту своего начала, — представляет последовательность ребер $e_1 = (v_0, v_1)$, $e_2 = (v_1, v_2)$, ..., $e_k = (v_{k-1}, v_k)$ с $v_k = v_0$.

[3] Для удобства мы обычно разрешаем пути (v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k) в графе включать повторяющиеся вершины либо содержать один или несколько циклов. Подобный «составной» путь можно преобразовать в путь без циклов с одинаковыми конечными точками v_0 и v_k, последовательно выделяя субпути между повторными прохождениями одной и той же вершины (см. в этой связи рис. 15.2).

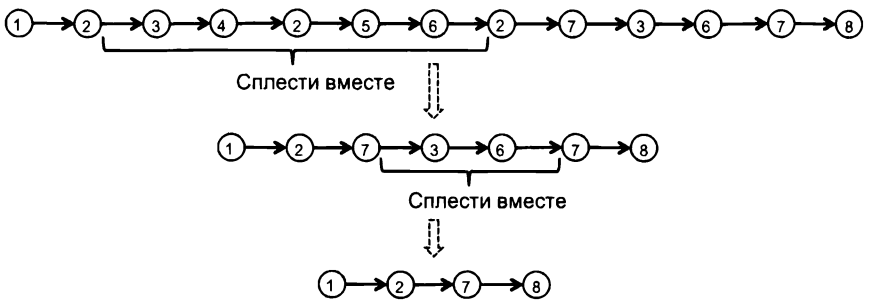
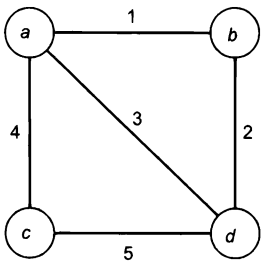


Рис. 15.2. Путь с повторяющимися вершинами может быть преобразован в путь без повторяющихся вершин и с теми же конечными точками

УПРАЖНЕНИЕ 15.1

Какова минимальная сумма реберных стоимостей остовного дерева следующего ниже графа? (Каждое ребро помечено его стоимостью.)



а) 6

б) 7

в) 8

г) 9

(Решение и пояснение см. в разделе 15.1.3.)

Имеет смысл говорить об остовных деревьях только *связных* графов $G = (V, E)$, в которых можно перемещаться из любой вершины $v \in V$ в любую другую вершину $w \in V$, используя путь ребер в E[1]. Если в E нет пути между вер-

[1] Например, график на рис. 15.1 является замкнутым, а график на рис. 15.3 — нет.

шинами v и w, то, разумеется, нет ни одного в любом подмножестве $T \subseteq E$ ребер. По этой причине в данной главе мы будем считать, что входной граф является связным графом.

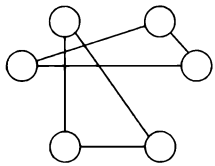


Рис. 15.3. Граф, который не является связным

ДОПУЩЕНИЕ ЗАДАЧИ ОБ ОСТОВНОМ ДЕРЕВЕ

Входной граф $G = (V, E)$ соединен, по крайней мере, одним путем между каждой парой вершин.

Вычислить минимальное остовное дерево четырехвершинного графа, как в упражнении 15.1, достаточно легко; а как насчет общего случая?

ЗАДАЧА О МИНИМАЛЬНОМ ОСТОВНОМ ДЕРЕВЕ (MST)

Вход: связный неориентированный граф $G = (V, E)$ и вещественная стоимость c_e для каждого ребра $e \in E$.

Выход: остовное дерево $T \subseteq E$ графа G с минимально возможной суммой $\sum_{e \in T} c_e$ реберных стоимостей[1].

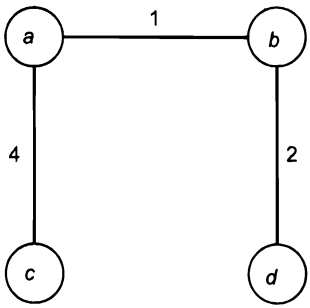
Мы можем допустить, что входной граф имеет не более одного ребра между каждой парой вершин; все, кроме самого дешевого из множества парал-

[1] Незамкнутые графы допустимо рассматривать в контексте минимальной проблемы покрывающего леса, сводящейся к отысканию максимального ациклического подграфа с минимально возможной суммой краевых затрат. Проблема решается последовательным вычислением замкнутых компонентов входного графа за линейное время (используя поиск по ширине или глубине, см. главу 8 *части 2*) и применением алгоритма для задачи MST к каждому компоненту в отдельности.

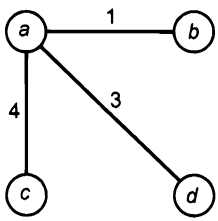
лельных ребер, могут быть отброшены без изменения задачи. Подобно минимизации суммы взвешенных сроков завершения (глава 13) или задачи оптимального беспрефиксного кода (глава 14), число возможных решений может быть экспоненциальным по размеру задачи[1]. Может ли существовать алгоритм, который волшебным образом наводит на иголку минимальной стоимости в стоге сена?

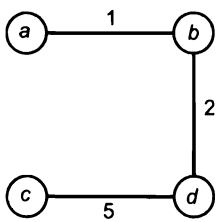
15.1.3. Решение упражнения 15.1

Правильный ответ: (б). Минимальное остовное дерево состоит из ребер (a, b), (b, d) и (a, c):



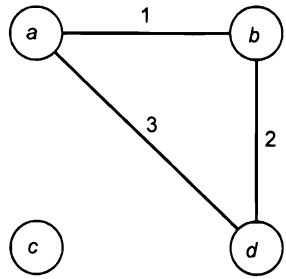
Сумма стоимостей ребер равна 7. Ребра не содержат цикла и могут использоваться для перемещения из любой вершины в любую другую вершину. Вот два остовных дерева с худшей суммарной стоимостью 8:





[1] Например, формула Кэли — результат комбинаторики, утверждающей, что полный граф с n-вершинами (в котором присутствуют $\binom{n}{2}$ возможных ребра) имеет n^{n-2} различных остовных деревьев. Но при $n \geq 50$ это число превысит предполагаемое количество атомов во Вселенной.

Три ребра — (*a*, *b*), (*b*, *d*) и (*a*, *d*) — имеют меньшую общую стоимость 6:



но эти ребра не образуют остовное дерево. Фактически они терпят неудачу по обоим пунктам, поскольку образуют цикл, при этом невозможно использовать их для перемещения из *c* в любую другую вершину.

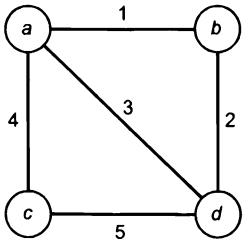
15.2. Алгоритм Прима

Нашим первым алгоритмом для задачи о минимальном остовном дереве будет *алгоритм Прима*, названный в честь Роберта С. Прима, который заявил о существовании алгоритма в 1957 году. Указанный алгоритм очень напоминает алгоритм отыскания кратчайшего пути Дейкстры (глава 9 *части 2*), поэтому не стоит удивляться, что двумя годами позже Эдсгер В. Дейкстра независимо пришел к тому же алгоритму. Только позже стало понятно, что указанный алгоритм был открыт еще раньше, в 1930 году, Войтехом Ярником. Соответственно алгоритм называется и *алгоритмом Ярника*, и *алгоритмом Прима—Ярника*[1].

15.2.1. Пример

Пройдемся по алгоритму Прима на конкретном примере (см. вновь упражнение 15.1):

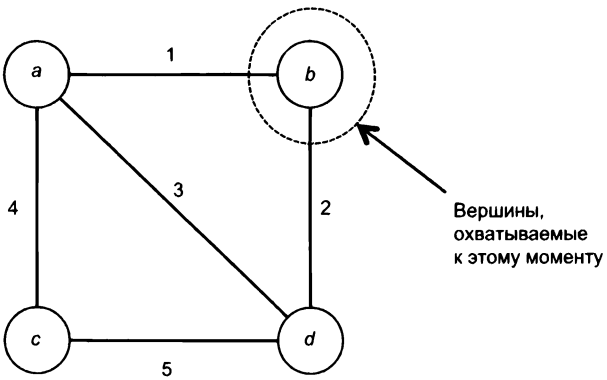
[1] В контексте сказанного следует критически переосмыслить статью «Об истории проблемы минимального остовного дерева» Рональда Л. Грэхема и Павола Ада («Летопись истории вычислений», 1985).



Может показаться странным, что мы будем пошагово анализировать пример алгоритма до того, как вы увидите его код, но поверьте: после того, как вы вникнете в пример, псевдокод напишется сам собой[1].

Алгоритм Прима начинается с произвольного выбора вершины, скажем, в нашем примере — вершины *b*. План состоит в том, чтобы строить дерево по одному ребру за раз, начиная с *b* и выращивая его, как плесневый гриб, до тех пор пока дерево не охватит все множество вершин. На каждой итерации мы будем жадно добавлять самое дешевое ребро, расширяющее охват текущего дерева.

Начальное (пустое) дерево алгоритма охватывает только стартовую вершину *b*. Существует два варианта расширения его охвата: ребро (*a*, *b*) и ребро (*b*, *d*).

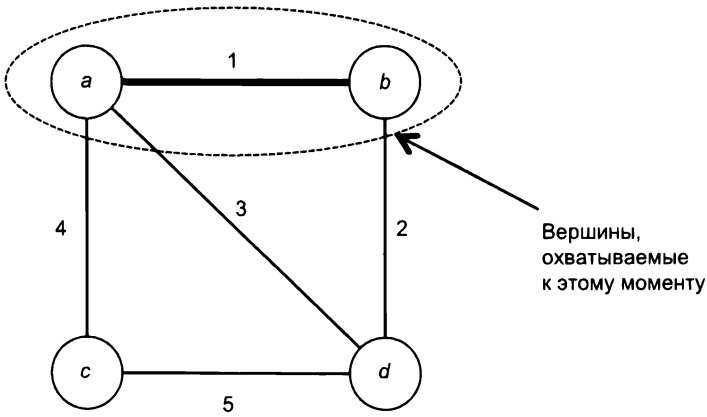


Первое дешевле, поэтому алгоритм выбирает его. Текущее дерево охватывает вершины *a* и *b*.

---

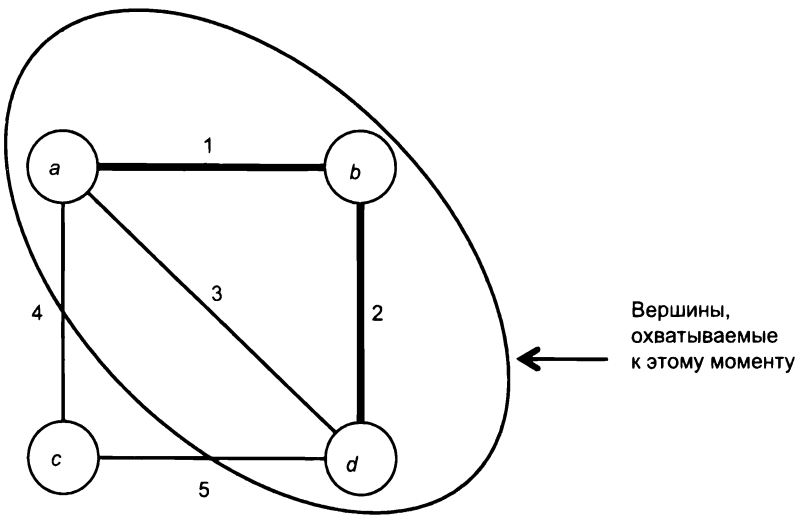
[1] Те, кто ранее прочел *часть 2*, по-видимому, должны признать разительное сходство с алгоритмом кратчайшего пути Дейкстры.

На второй итерации охват дерева выполнен тремя ребрами: (a, c), (a, d) и (b, d).

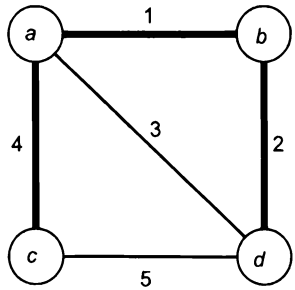


Самым дешевым из них является (b, d). После его добавления текущее дерево охватывает a, b и d. Обе конечные точки ребра (a, d) были втянуты в множество вершин, охватываемых к этому моменту; добавление этого ребра в будущем создало бы цикл, поэтому алгоритм не рассматривает его дальше.

На заключительной итерации есть два варианта расширения досягаемости дерева до c, ребра (a, c) и (c, d):



Алгоритм Прима выбирает более дешевое ребро (a, c), приходя к тому же минимальному остовному дереву, что и в упражнении 15.1:



15.2.2. Псевдокод

В общем случае алгоритм Прима выращивает остовное дерево из стартовой вершины по одному ребру за раз, причем каждая итерация расширяет охват текущего дерева на одну дополнительную вершину. Будучи жадным, этот алгоритм в процессе работы всегда выбирает самое дешевое ребро.

PRIM

Вход: связный неориентированный граф $G = (V, E)$, представленный в виде списков смежности, и стоимость c_e для каждого ребра $e \in E$.

Выход: ребра минимального остовного дерева графа G.

```
// Инициализация
X := {s} // s — это произвольно выбранная вершина
T := ∅ // инвариант: ребра в T охватывают X
// Главный цикл
while существует ребро (v, w), где v ∈ X, w ∉ X do
  (v*, w*) := минимально-стоимостное такое ребро
  добавить вершину w* в X
  добавить ребро (v*, w*) в T
return T
```

Множества T и X отслеживают выбранные ребра и вершины, охватываемые к этому моменту. Алгоритм инициализирует X произвольно выбранной стартовой вершиной s; как мы увидим, алгоритм является правильным независимо от того, какую именно вершину он выбирает[1]. Каждая итерация отвечает за добавление одного нового ребра в T. Во избежание избыточных ребер и с целью обеспечения того, чтобы добавление ребра расширяло охват T, алгоритм рассматривает только те ребра, которые «пересекают границу», с одной конечной точкой в каждом из X и $V - X$ (рис. 15.4). Если таких ребер много, то алгоритм жадно выбирает самое дешевое. После $n - 1$ итераций (где n — это число вершин) X содержит все вершины, и алгоритм останавливается. В рамках нашего допущения о том, что входной граф G является связным, алгоритм не может застрять; если бы когда-либо имелась итерация, в которой ни одно ребро не проходило G между X и $V - X$, то мы могли бы заключить, что G не является связным (потому что он не содержит пути из любой вершины в X до любой вершины в $V - X$).

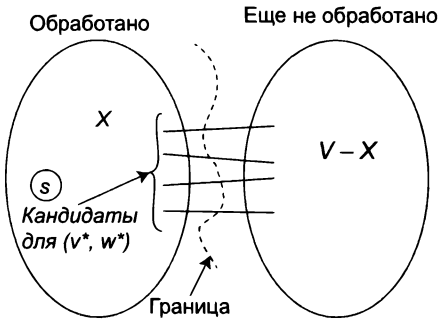


Рис. 15.4. Каждая итерация алгоритма Прима выбирает одно новое ребро, которое проходит из X в $V - X$

Алгоритм `Prim` вычисляет минимальное остовное дерево в четырехвершинном пятиреберном графе упражнения 15.1, что приближенно означает... ничего.

---

[1] В определении задачи MST нет ссылки на начальную вершину, в связи с чем ее искусственное введение может показаться странным. Однако начальная вершина позволяет весьма точно имитировать алгоритм Дейкстры по кратчайшему пути (обремененному начальной вершиной в рамках задачи о поиске кратчайшего пути из одного источника). Суть задачи от этого не меняется: соединение каждой пары вершин — то же, что и соединение некоторых вершин с любой другой вершиной (для получения пути v–w вставьте пути из v в s и из s в w).

Тот факт, что алгоритм правильно работает на конкретном примере, *не* означает, что он является правильным в общем случае![1] Вы должны изначально скептически относиться к алгоритму `Prim` и требовать доказательства его правильности.

Теорема 15.1 (правильность алгоритма `Prim`). *Для каждого связного графа $G = (V, E)$ и вещественных реберных стоимостей алгоритм* `Prim` *возвращает минимальное остовное дерево графа G.*

См. раздел 15.4 с доказательством теоремы 15.1.

15.2.3. Простая реализация

Анализ времени выполнения алгоритма `Prim` (исходя из простой реализации) намного проще, чем доказательство его правильности (что вполне типично для жадных алгоритмов).

УПРАЖНЕНИЕ 15.2

Какое из следующих ниже времен выполнения лучше всего описывает простую реализацию алгоритма `Prim` отыскания минимального остовного дерева для графов, представленных в виде списков смежности? Как обычно, n и m обозначают соответственно число вершин и ребер входного графа.

а) $O(m + n)$

б) $O(m \log n)$

в) $O(n^2)$

г) $O(mn)$

(Решение и пояснение см. ниже.)

Правильный ответ: (г). Простая реализация отслеживает то, какие вершины находятся в X, ассоциируя булеву переменную с каждой вершиной. На каж-

[1] Как известно, даже сломанные часы дважды в сутки показывают правильное время.

дой итерации выполняется исчерпывающий поиск по всем ребрам с целью выявления самого дешевого с одной конечной точкой в каждом из X и $V - X$. После $n - 1$ итераций у алгоритма иссякают новые вершины для добавления в свое множество X, и он останавливается. Поскольку число итераций равно $O(n)$ и каждая из них занимает $O(m)$ времени, суммарное время выполнения составляет $O(mn)$.

Утверждение 15.2 (время выполнения алгоритма `Prim` (в простой версии)). *Для каждого графа $G = (V, E)$ и вещественных реберных стоимостей простая реализация алгоритма* `Prim` *выполняется за время $O(mn)$, где $m = |E|$ и $n = |V|$.*

*15.3. Ускорение алгоритма Прима посредством куч

15.3.1. В поисках времени выполнения, близкого к линейному

Время выполнения простой реализации алгоритма `Prim` (утверждение 15.2) — не что иное, как полиномиальная функция размера задачи, тогда как исчерпывающий поиск по всем остовным деревьям графа может занять экспоненциальное количество времени (см. сноску на с. 84). Эта реализация является достаточно быстрой, для того чтобы справляться с обработкой графов среднего размера (с тысячами вершин и ребер) за разумное количество времени, но не графов с миллионами вершин и ребер. Помните мантру любого проектировщика алгоритмов: можем ли мы добиться лучшего? Святой Грааль в разработке алгоритмов — это линейно-временной алгоритм (или близкий к нему), и это именно то, чего мы хотим для задачи о минимальном остовном дереве (MST).

Для решения задачи за время, близкое к линейному, нам не нужен более хороший *алгоритм* — нужна более оптимальная *реализация* алгоритма Прима. Ключевое значение имеет то, что простая реализация снова и снова выполняет минимальные вычисления, используя исчерпывающий поиск. Любой метод вычисления повторных минимальных вычислений, который будет быстрее, чем исчерпывающий поиск, приведет к более быстрой реализации алгорит-

ма Прима. В разделе 14.3.6 мы кратко упомянули, что существует структура данных, смысл которой заключается в быстрых минимальных вычислениях: так называемая *кучевая* структура данных. Следовательно, и алгоритм Прима требует кучи!

15.3.2. Кучевая структура данных

Куча поддерживает эволюционирующее множество объектов с ключами, а также несколько быстрых операций, из которых нам потребуется три.

КУЧИ: ТРИ ПОДДЕРЖИВАЕМЫЕ ОПЕРАЦИИ

Вставить: с учетом кучи H и нового объекта x добавить x в H.

Извлечь минимум: с учетом кучи H удалить и вернуть из H объект с наименьшим ключом (либо указатель на него).

Удалить: с учетом кучи H и указателя на объект x в H удалить x из H.

Например, если вызвать операцию Вставить четыре раза, чтобы добавить объекты с ключами 12, 7, 29 и 15 в пустую кучу, то операция Извлечь минимум вернет объект с ключом 7.

Стандартные реализации кучи гарантированно обеспечивают следующее.

Теорема 15.3 (время выполнения трех кучевых операций). *В куче с n объектами операции* Вставить, Извлечь минимум *и* Удалить *выполняются за время $O(\log n)$.*

В качестве бонуса в типичных реализациях константа, скрытая обозначением O-большое, и объем пространственных накладных расходов относительно невелики[1].

[1] В контексте раздела не так важно знать, как именно реализованы кучи и как они выглядят изнутри. Обучение в данном случае строится на преимуществах операций с логарифмическим временем (о дополнительных операциях и подробностях реализации см. главу 10 *части 2*).

15.3.3. Как использовать кучи в алгоритме Прима

Кучи обеспечивают невероятно быструю, почти линейно-временную реализацию алгоритма Прима[1].

Теорема 15.4 (время выполнения алгоритма Прима (на основе кучи)). *Для каждого графа $G = (V, E)$ и вещественных реберных стоимостей реализация алгоритма Прима на основе кучи выполняется за время $O((m + n) \log n)$, где $m = |E|$ и $n = |V|$*[2].

Ограничение времени выполнения в теореме 15.4 — это только логарифмический фактор, превышающий время, необходимое для чтения входных данных. Таким образом, задача минимального остовного дерева квалифицируется как «свободный примитив», объединяющий такие элементы, как сортировка, вычисление связных компонентов графа и задача кратчайшего пути с единственным истоком.

БЕСПЛАТНЫЕ ПРИМИТИВЫ

Алгоритм с линейным или почти линейным временем выполнения можно рассматривать как «бесплатный» примитив, потому что объем используемых вычислений едва превышает объем, необходимый только для чтения входных данных. Когда у вас есть примитив, имеющий отношение к вашей задаче, к тому же необычайно быстрый, почему бы им не воспользоваться? Например, вы всегда можете вычислить минимальное остовное дерево для данных неориентированного графа на этапе предобработки, даже если не уверены, чем это обернется в дальнейшем. К слову, одна из целей этой книжной серии — снабдить ваш алгоритмический инструментарий как можно бóльшим числом бесплатных примитивов, готовых к применению в любой момент.

---

[1] Тем, кто уже прочел *часть 2*, вероятно, знакомы реализации алгоритма кратчайшего пути Дейкстры на основе кучи (см. раздел 10.4).

[2] Исходя из предположения о том, что входной граф связан, а m равно по крайней мере $n - 1$, можно упростить $O((m + n) \log n)$ до $O(m \log n)$ (в пределах времени выполнения).

В кучевой реализации алгоритма Прима объекты в куче соответствуют еще необработанным вершинам ($V - X$ в псевдокоде алгоритма `Prim`)[1, 2]. Ключ вершины $w \in V - X$ определяется как минимальная стоимость инцидентного пересекающего ребра (рис. 15.5).

ИНВАРИАНТ

Ключ вершины $w \in V - X$ есть минимальная стоимость ребра (v, w), где $v \in X$ (либо $+\infty$, если такого ребра не существует).

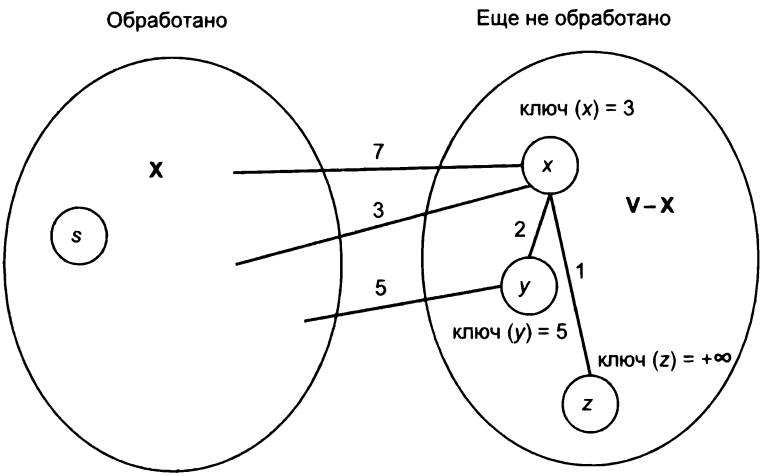


Рис. 15.5. Ключ вершины $w \in V - X$ определяется как минимальная стоимость ребра (v, w), где $v \in X$ (либо $+\infty$, если такого ребра не существует)

Интерпретируя эти ключи, представьте использование двухраундного турнира с выбыванием для выявления минимально-стоимостного ребра (v, w), где $v \in X$ и $w \notin X$. Первый раунд состоит из локального турнира для каждой

[1] Мы ссылаемся на вершины входного графа и соответствующие объекты в куче взаимозаменяемо.

[2] Да, можно хранить ребра входного графа в куче для последующей замены минимальных вычислений (по ребрам) в простой реализации с обращением к `ExtractMin`. Но более совершенная реализация хранит вершины в куче.

вершины $w \in V - X$, где участниками являются ребра (v, w) с $v \in X$, и победитель в первом туре является самым дешевым из участников (либо $+\infty$, если нет таких ребер). Победители первого раунда (не более одного на вершину $w \in V - X$) переходят во второй раунд, а окончательный чемпион является самым дешевым победителем первого раунда. Таким образом, ключом вершины $w \in V - X$ является именно стоимость выигрышного ребра в локальном турнире в w. Извлечение вершины с минимальным ключом затем реализует второй раунд турнира и возвращает на серебряном блюдечке с голубой каемочкой следующее дополнение в текущее решение. Поскольку мы платим за музыку и поддерживаем инвариант, сохраняя ключи объектов в актуальном состоянии, мы можем реализовать каждую итерацию алгоритма Прима с помощью одной кучевой операции.

15.3.4. Псевдокод

Тогда псевдокод будет выглядеть так:

PRIM (НА ОСНОВЕ КУЧИ)

Вход: связный неориентированный граф $G = (V, E)$, представленный в виде списков смежности, и стоимость c_e для каждого ребра $e \in E$.

Выход: ребра минимального остовного дерева графа G.

```
// Инициализация
 1 X := {s}, T = ∅, H := пустая куча
 2 for каждый v ≠ s do
 3     if существует ребро (s, v) ∈ E then
 4         key(v) := c_sv, winner(v) := (s, v)
 5     else    // v не имеет пересекающих инцидентных ребер
 6         key(v) := +∞, winner(v) := NULL
 7     Вставить v в H
   // Главный цикл
 8 while H не является пустым do
 9     w* := Извлечь_минимум(H)
```

```
10    добавить w* в X
11    добавить winner(w*) в T
      // обновить ключи для поддержки инварианта
12    for каждое ребро (w*, y) с y ∈ V – X do
13        if c_{w*y} < key(y) then
14            Удалить y из H
15            key(y) := c_{w*y}, winner(y) := (w*, y)
16            Вставить y в H
17  return T
```

Каждая еще не обработанная вершина w записывает в полях *победитель* `winner` и *ключ* `key` данные об идентичности и стоимости победителя своего локального турнира — самого дешевого ребра, инцидентного w, которое пересекает границу (то есть ребра (v, w), где $v \in X$). Строки 2–7 инициализируют эти значения для всех вершин, отличных от s, с целью удовлетворения инварианта, и вставляют эти вершины в кучу. Строки 9–11 реализуют одну итерацию главного цикла алгоритма Прима. Инвариант обеспечивает, чтобы локальный победитель извлеченной вершины был самым дешевым ребром, пересекающим границу, то есть являлся правильным ребром, добавляемым следующим в текущее дерево T. Упражнение 15.3 иллюстрирует, как извлечение может менять границу, требуя обновления ключей вершин, все еще находящихся в $V - X$ с целью обеспечения инварианта.

УПРАЖНЕНИЕ 15.3

На рис. 15.5 предположим, что вершина x извлечена и перемещена в множество X. Какими должны быть новые значения соответственно ключей y и z?

а) 1 и 2

б) 2 и 1

в) 5 и $+\infty$

г) $+\infty$ и $+\infty$

(Решение и пояснение см. в разделе 15.3.6.)

Строки 12–16 псевдокода выполняют необходимые обновления ключей вершин, оставшихся в $V - X$. Когда w^* перемещается из $V - X$ в X, ребра формы (w^*, y), где $y \in V - X$, пересекают границу в первый раз; в локальных турнирах в вершинах $V - X$ имеются новые участники. Мы можем игнорировать тот факт, что ребра формы (u, w^*), где $u \in X$ втягиваются в X и больше не пересекают границу, поскольку мы не несем ответственности за сохранение ключей для вершин в X. Для вершины $y \in V - X$ новым победителем ее локального турнира является либо старый *победитель* (сохраненный в `winner`(y)), либо новый участник (w^*, y). Строка 12 перебирает новых участников[1]. Строка 13 проверяет, является ли ребро (w^*, y) новым победителем локального турнира y; если да, то строки 14–16 обновляют соответственно поля *ключа* и *победителя* y и кучу H[2].

15.3.5. Анализ времени выполнения

Фаза инициализации (строки 1–7) выполняет $n - 1$ кучевых операций (по одной операции Вставить в вершину, отличную от s) и $O(m)$ дополнительной работы, где n и m обозначают соответственно число вершин и ребер. Существует $n - 1$ итераций главного цикла `while` (строки 8–16), поэтому строки 9–11 вносят в суммарное время выполнения $O(n)$ кучевых итераций и $O(n)$ дополнительной работы. Ограничение суммарного времени, проводимого в строках 12–16, является заковыристой частью; ключевое значение имеет то, что *каждое ребро графа G рассматривается в строке 12 только один раз*, в итерации, в которой первая из его конечных точек всасывается в X (то есть играет роль w^*). Когда ребро исследуется, алгоритм выполняет две кучевые операции (в строках 14 и 16) и $O(1)$ дополнительной работы, поэтому суммарный вклад строк 12–16 во время выполнения (по всем итерациям цикла) составляет $O(m)$ кучевых операций плюс $O(m)$ дополнительной работы. Окончательный перечень показателей выглядит так:

$$O(m + n) \text{ кучевых операций} + O(m + n) \text{ дополнительной работы.}$$

---

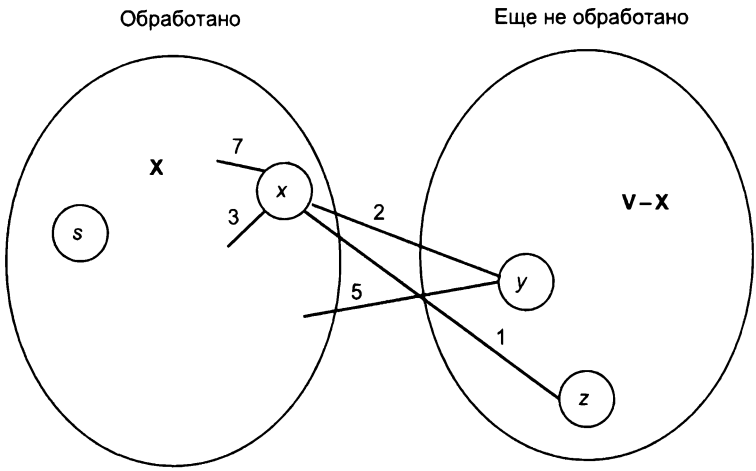
[1] На этом шаге весьма удобно представлять входной граф через списки смежности, поскольку к краям формы (w^*, y) можно обращаться напрямую через массив падающих ребер w^*.

[2] Некоторые реализации кучи экспортируют операцию `DecreaseKey`, что позволяет реализовать строки 14–16 с помощью всего одной кучевой операции, а не двух.

Куча никогда не хранит более $n - 1$ объектов, поэтому каждая кучевая операция выполняется за время $O(\log n)$ (теорема 15.3). Суммарное время выполнения составляет $O((m + n) \log n)$, согласно теореме 15.4. *Ч. Т. Д.*

15.3.6. Решение упражнения 15.3

Правильный ответ: (б). После перемещения вершины x из $V - X$ в X новое изображение выглядит так:



Ребра формы (v, x), где $v \in X$, всасываются в X и больше не пересекают границу (как и в случае с ребрами со стоимостями 3 и 7). Другие ребра, инцидентные x, (x, y) и (x, z), частично выдергиваются из $V - X$ и теперь пересекают границу. Как для y, так и для z эти новые инцидентные пересекающие ребра дешевле, чем все их старые. С целью поддержания инварианта оба их ключа должны быть обновлены: ключ y из 5 в 2, а ключ z из $+\infty$ в 1.

*15.4. Алгоритм Прима: доказательство правильности

Выполнить доказательство правильности алгоритма Прима (теорема 15.1) проще, когда все реберные стоимости различны (см. упражнение 15.5). До-

казательство выполним в два этапа. Сначала определим свойство, именуемое «свойством минимального узкого места», которым обладает выход алгоритма Прима. Второй этап позволяет установить, что в графе с различными реберными стоимостями остовное дерево с этим свойством должно быть минимальным остовным деревом[1].

15.4.1. Свойство минимального узкого места

Мы можем обусловить свойство минимального узкого места по аналогии с алгоритмом поиска кратчайшего пути Дейкстры. Единственное существенное различие между алгоритмами Прима и Дейкстры — это критерий, используемый для выбора пересекающего ребра на каждой итерации. Алгоритм Дейкстры жадно выбирает подходящее ребро, которое минимизирует расстояние (то есть *сумму* длин ребер) от стартовой вершины s и по этой причине вычисляет кратчайший путь от s до любой другой вершины (при условии, что длины ребер не являются отрицательными). Алгоритм Прима, всегда выбирая подходящее ребро с минимальной индивидуальной стоимостью, практически стремится минимизировать *максимальную* реберную стоимость вдоль каждого пути[2]. Свойство минимального узкого места (Minimum Bottleneck Property, MBP) отражает эту идею точнее: с учетом графа с вещественными реберными стоимостями определить узкое место пути P как максимальную стоимость $\max_{e \in P} c_e$ одного из его ребер.

---

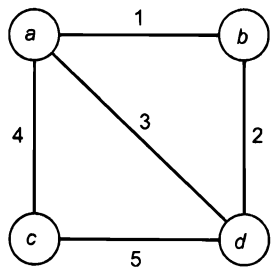
[1] Популярный, хотя и более абстрактный подход к доказательству правильности алгоритма Прима (и Краскала) сводится к использованию так называемого свойства обрезки MST (подробнее см. упражнение 15.7).

[2] Это наблюдение связано со следующей загадкой: почему алгоритм Дейкстры работает правильно только с неотрицательными длинами ребер, тогда как алгоритм Прима — с произвольными (положительными или отрицательными)? Ключевой компонент доказательства правильности алгоритма Дейкстры — так называемая монотонность пути, означающая, что добавление дополнительных ребер в конце пути может только ухудшить его. Прикрепление края отрицательной длины к пути уменьшит его общую длину, поэтому-то для сохранения монотонности пути и необходимы неотрицательные длины края. Следует, кроме того, помнить, что для алгоритма Прима мерой «измерения» служит максимальная стоимость ребра в пути и эта мера не может уменьшиться, так как дополнительные (положительные или отрицательные) ребра привязаны к пути.

СВОЙСТВО МИНИМАЛЬНОГО УЗКОГО МЕСТА (МВР)

Для графа $G = (V, E)$ с вещественными реберными стоимостями ребро $(v, w) \in E$ удовлетворяет *свойству минимального узкого места*, если оно является путем v–w с минимальным узким местом.

Другими словами, ребро (v, w) удовлетворяет свойству минимального узкого места тогда и только тогда, когда нет пути v–w, состоящего исключительно из ребер со стоимостью меньше c_{vw}. В нашем примере:



ребро (a, d) не удовлетворяет свойству минимального узкого места (каждое ребро на пути $a \rightarrow b \rightarrow d$ дешевле, чем (a, d)), как и ребро (c, d) (каждое ребро на пути $c \rightarrow a \rightarrow d$ дешевле, чем (c, d)). Остальные три ребра также удовлетворяют указанному свойству[1]. Следующая ниже лемма реализует первый этап нашего плана доказательства, связывая выход алгоритма Прима со свойством минимального узкого места.

Лемма 15.5 (алгоритм `Prim` достигает свойства МВР). *Для каждого связного графа $G = (V, E)$ и вещественных реберных стоимостей каждое ребро, выбранное алгоритмом* `Prim`, *удовлетворяет свойству минимального узкого места.*

Доказательство: рассмотрим ребро (v^*, w^*), выбранное на итерации алгоритма Прима, где $v^* \in X$ и $w^* \in V - X$. По жадному правилу алгоритма,

$$c_{v^*w^*} \leq c_{xy} \tag{15.1}$$

для каждого ребра $(x, y) \in E$, где $x \in X$ и $y \in V - X$.

[1] Отметим, вовсе не случайно ребра, удовлетворяющие МВР в данном примере, являются и ребрами в минимальном остовном дереве.

Чтобы доказать, что (v^*, w^*) удовлетворяет условию MBP, рассмотрим произвольный v^*–w^*-путь P. Поскольку $v^* \in X$ и $w^* \notin X$, путь P проходит в некоторой точке из X в $V - X$, скажем, через ребро (x, y), где $x \in X$ и $y \in V - X$ (рис. 15.6). Узким местом P является c_{xy}, которая по неравенству (15.1) равна, по крайней мере, $c_{v^*w^*}$. Поскольку P был произвольным, ребро (v^*, w^*) является минимальным узким местом пути v^*–w^*. *Ч. Т. Д.*

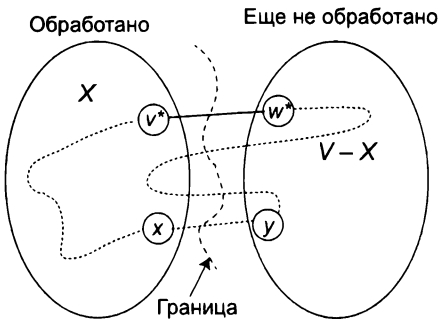


Рис. 15.6. Каждый путь v^*–w^* проходит, по крайней мере, один раз из X в $V - X$. Пунктирные линии представляют один такой путь

Мы вызвались решить задачу минимального остовного дерева, а не достичь свойства минимального узкого места. Но я бы не стал отнимать ваше время; в графах с разными реберными стоимостями из последнего автоматически вытекает первое[1].

Теорема 15.6 (из свойства минимального узкого места вытекает минимальное остовное дерево). *Пусть $G = (V, E)$ есть граф с различными вещественными реберными стоимостями, а T — остовное дерево графа G. Если каждое ребро дерева T удовлетворяет свойству* минимального узкого места, *то T является минимальным остовным деревом.*

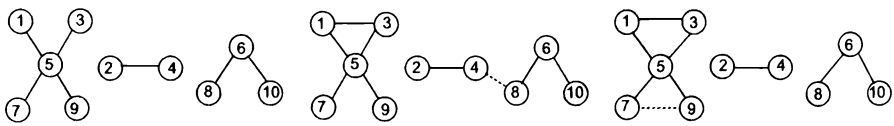
Плохая новость — доказательство теоремы 15.6 включает несколько этапов. Хорошая новость — мы можем повторно использовать их все для того, чтобы установить и правильность еще одного важного алгоритма отыскания

[1] Верно и обратное утверждение теоремы 15.6 (даже с неопределенными затратами на ребра): каждое ребро MST удовлетворяет MBP (см. задачу 15.4).

минимального остовного дерева, алгоритма Краскала (теорема 15.11 в разделе 15.5)[1].

15.4.2. Интересные факты об остовных деревьях

Предварительно докажем несколько простых и полезных фактов о неориентированных графах и их остовных деревьях. Несколько слов о терминологии. Граф $G = (V, E)$ — не обязательно связный — естественным образом распадается на «части», именуемые *связными компонентами* графа. Более формально связная компонента представляет собой максимальное подмножество $S \subseteq V$ вершин, в котором существует путь в G из любой вершины в S до любой другой вершины в S. Например, связными компонентами графа на рис. 15.7, *а* являются $\{1, 3, 5, 7, 9\}$, $\{2, 4\}$ и $\{6, 8, 10\}$. Граф связан с путем между каждой парой вершин тогда и только тогда, когда он имеет одну связную компоненту[2].



а) Три компоненты *б)* Слияние компонент *в)* Создание цикла

Рис. 15.7. В (*а*) граф с множеством вершин {1, 2, 3,..., 10} и тремя связными компонентами. В (*б*) добавление ребра (4, 8) объединяет две компоненты. В (*в*) добавление ребра (7, 9) создает новый цикл

Теперь представьте, что вы начинаете с пустого графа (с вершинами, но без ребер) и добавляете к нему ребра по одному. Что меняется при добавлении нового ребра? Одна из возможностей заключается в том, что новое ребро объединяет две связные компоненты (рис. 15.7, *б*). Мы называем это *добавлением ребра с типом F* («*F*» от *англ.* fusion). Еще одна возможность заключается

---

[1] Теорема 15.6 не выполняется в графах с неопределенными затратами на ребра. В качестве контрпримера рассмотрим треугольник с одним ребром со стоимостью 1 и двумя ребрами со стоимостью 2 каждый. Тем не менее, алгоритмы Прима и Краскала остаются верными с произвольными действительными стоимостными ребрами (см. задачу 15.5).

[2] Подробнее о замкнутых компонентах, алгоритме их вычисления за линейное время см. главу 8 *части 2*.

в том, что новое ребро закрывает уже существующий путь, создавая цикл (рис. 15.7, *в*). Мы называем это *добавлением ребра с типом C* («*C*» от *англ.* cycle). Наша первая лемма утверждает, что каждое добавление ребра (v, w) имеет либо тип C, либо тип F (но не оба), в зависимости от того, имеет ли уже граф путь v–w. Если это утверждение кажется вам очевидным, то можно, минуя доказательство, двигаться дальше.

Лемма 15.7 (создание цикла/слияние компонент). *Пусть $G = (V, E)$ есть неориентированный граф, и $v, w \in V$ — две разные вершины, такие что $(v, w) \notin E$.*

а) (тип C) Если v и w находятся в одной и той же связной компоненте графа G, то добавление ребра (v, w) в G создает по крайней мере один новый цикл и не изменяет число связных компонент.

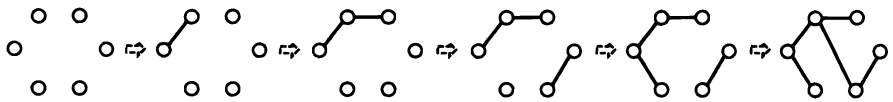
б) (тип F) Если v и w находятся в разных связных компонентах графа G, то добавление ребра (v, w) в G не создает никаких новых циклов и уменьшает число связных компонент на 1.

Доказательство: для части (а) если v и w находятся в одной и той же связной компоненте G, то существует v–w-путь P в G. После добавления ребра $P \cup \{(v, w)\}$ образует новый цикл. Связные компоненты остаются прежними, при этом новое ребро (v, w) поглощается связной компонентой, которая уже содержит обе его конечные точки. Для части (б) обозначим через S_1 и S_2 (разные) связные компоненты графа G, содержащие соответственно v и w. Во-первых, после добавления ребра связные компоненты S_1 и S_2 сливаются в единую компоненту $S_1 \cup S_2$, уменьшая число компонент на 1. Для вершин $x \in S_1$ и $y \in S_2$ вы можете создать путь x–y в новом графе, склеив путь x–v в G, ребро (v, w) и путь w–y в G. Во-вторых, предположим от противного, что добавление ребра создало новый цикл C. Этот цикл должен включать новое ребро (v, w). Но тогда $C - \{(v, w)\}$ будет путем v–w в G, что противоречит нашему допущению, что v и w находятся в разных связных компонентах. *Ч. Т. Д.*

Имея в своем распоряжении лемму 15.7, мы можем быстро вывести некоторые интересные факты об остовных деревьях.

Следствие 15.8 (остовные деревья имеют ровно $n - 1$ ребер). *Каждое остовное дерево n-вершинного графа имеет $n - 1$ ребер.*

Доказательство: пусть T равно остовному дереву графа $G = (V, E)$ с n вершинами. Начнем с пустого графа с множеством вершин V и будем добавлять ребра T по одному. Поскольку T не имеет циклов, каждое добавление ребра имеет тип F и уменьшает число связных компонент на 1 (лемма 15.7):

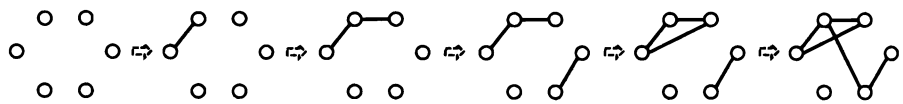


Процесс начинается с n связных компонент (где каждая вершина находится в своей компоненте) и заканчивается единицей (потому что T — это остовное дерево), поэтому число реберных добавлений должно составлять $n - 1$. *Ч. Т. Д.*

Есть два случая, когда подграф может не быть остовным деревом: он содержит цикл; он не является связным. Подграф с $n - 1$ ребрами — кандидат на остовное дерево по следствию 15.8 — не соблюдает одно из условий, только если он не соблюдает ни одно из них.

Следствие 15.9 (связность и ацикличность сопровождают друг друга). *Пусть $G = (V, E)$ есть граф и $T \subseteq E$ — подмножество $n - 1$ ребер, где $n = |V|$. Граф (V, T) является связным тогда и только тогда, когда он не содержит циклов.*

Доказательство: повторим процесс добавления ребер из следствия 15.8. Если каждое из $n - 1$ реберных добавлений имеет тип F, то лемма 15.7 (б) приводит к тому, что процесс завершается одной связной компонентой и без циклов (то есть к остовному дереву). В противном случае имеется добавление ребра с типом C, которое по лемме 15.7 (а) создает цикл, а также не уменьшает число связных компонент:



В этом случае процесс начинается с n связных компонент, и $n - 1$ реберных добавлений уменьшают число связных компонент не более чем в $n - 2$ раз, оставляя конечный граф (V, T), по крайней мере, с двумя связными компонентами. Следовательно, (V, T) не является ни связным, ни ациклическим. *Ч. Т. Д.*

Мы можем проаргументировать схожим образом, что выход алгоритма Прима является остовным деревом, не утверждая при этом, что остовное дерево является *минимальным*.

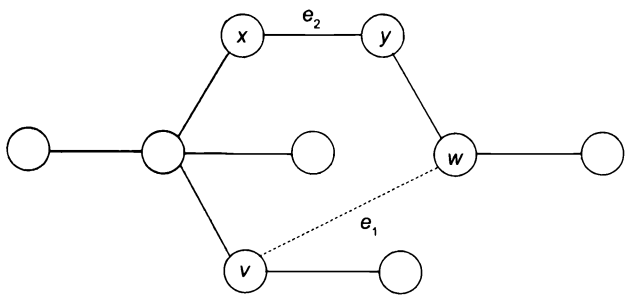
Следствие 15.10 (Алгоритм `Prim` выводит остовное дерево). *Для каждого связного входного графа алгоритм* `Prim` *выводит остовное дерево.*

Доказательство: на протяжении всего алгоритма вершины X образуют одну связную компоненту (V, T), и каждая вершина $V - X$ изолирована в своей собственной связной компоненте. Каждое из $n - 1$ реберных добавлений включает вершину w^* $V - X$ и, следовательно, имеет тип F, поэтому конечным результатом является остовное дерево. *Ч. Т. Д.*

15.4.3. Доказательство теоремы 15.6 (из свойства минимального узкого места следует минимальное остовное дерево)

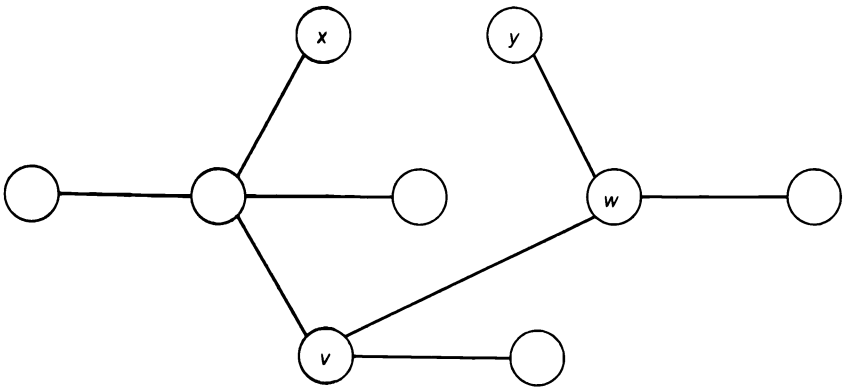
Доказательство теоремы 15.6 — это как раз то место, где мы используем наше постоянное допущение о том, что стоимости ребер являются разными.

Доказательство теоремы 15.6 (от противного). Пусть T равно связующему дереву, в котором каждое ребро удовлетворяет свойству минимального узкого места. Предположим, минимальное остовное дерево T^* имеет строго меньшую сумму реберных стоимостей. В контексте доказательства теоремы 13.1 следует обменять одно ребро на другое, создав остовное дерево T' с общей стоимостью меньше, чем T^*, тем самым противореча предполагаемой оптимальности T^*. Деревья T и T^* должны быть разными и иметь $n - 1$ ребер, где $n = |V|$ (следствие 15.8). Таким образом, T содержит по крайней мере одно ребро $e_1 = (v, w)$, которое не находится в T^*. Добавление e_1 в T^* создает цикл C, который содержит e_1 (лемма 15.7 (а)):



Являясь ребром T, e_1 удовлетворяет свойству минимального узкого места, поэтому существует, по крайней мере, одно ребро $e_2 = (x, y)$ в v–w-пути $C - \{e_1\}$ со стоимостью не менее c_{vw}. В рамках нашего допущения, что реберные стоимости являются разными, стоимость e_2 должна быть строго больше: $c_{xy} > c_{vw}$.

Теперь выведем T' из $T^* \cup \{e_1\}$, удалив ребро e_2:



Поскольку T^* имеет $n - 1$ ребер, то же справедливо и для T'. Поскольку T^* является связным, то же справедливо и для T'. Удаление ребра из цикла отменяет добавление ребра типа C, которое по лемме 15.7 (а) не влияет на число связных компонент. Из следствия 15.9 вытекает, что T' также является ациклическим и, следовательно, остовным деревом. Поскольку стоимость e_2 больше, чем стоимость e_1, T' имеет более низкую суммарную стоимость, чем T^*, что противоречит предполагаемой оптимальности T^* и завершает доказательство. *Ч. Т. Д.*

15.4.4. Сводя все воедино

Теперь у нас есть все для доказательства правильности алгоритма Прима в графах с разными реберными стоимостями.

Доказательство теоремы 15.1: следствие 15.10 доказывает, что выходом алгоритма Прима является остовное дерево. Из леммы 15.5 вытекает, что каждое ребро этого остовного дерева удовлетворяет свойству минимального узкого места. Теорема 15.6 гарантирует, что это остовное дерево является минимальным остовным деревом. *Ч. Т. Д.*

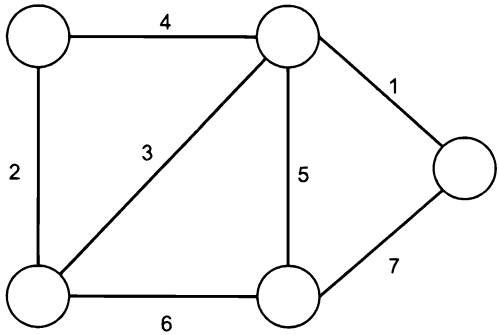
15.5. Алгоритм Краскала

В этом разделе описывается второй алгоритм для задачи о минимальном остовном дереве, *алгоритм Краскала*[1]. Но почему нас должен заботить алгоритм Краскала при наличии в нашем распоряжении алгоритма Прима на основе кучи? Во-первых, этот алгоритм находится в зале славы одним из первых, поэтому каждый опытный программист и информатик должен знать о нем. Надлежаще реализованный, он соперничает с алгоритмом Прима как в теории, так и на практике. Во-вторых, это дает возможность изучить новую и полезную структуру данных — *Union-Find*, или *непересекающихся (дизъюнктивных) множеств*. В-третьих, существуют очень интересные связи между алгоритмом Краскала и широко используемыми алгоритмами кластеризации (см. раздел 15.8).

15.5.1. Пример

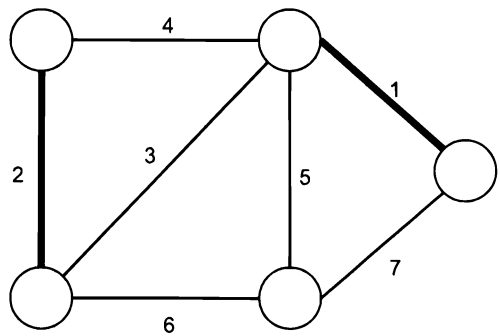
Как и в случае с алгоритмом Прима, полезно увидеть пример алгоритма Краскала в действии, и только потом перейти к его псевдокоду. Вот входной граф:

[1] Выявлен Джозефом Б. Краскалом в середине 1950-х, то есть примерно тогда, когда Прим и Дейкстра заново открывали то, что сейчас называется алгоритмом Прима.

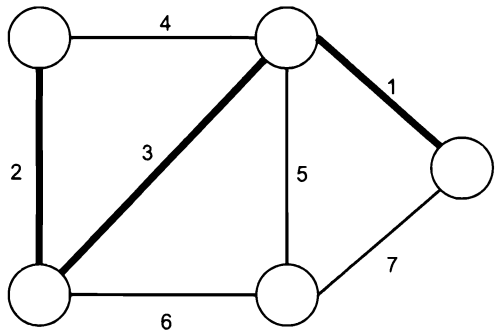


Алгоритм Краскала, как и алгоритм Прима, жадно строит остовное дерево по одному ребру за раз. Но вместо того, чтобы выращивать одно дерево из начальной вершины, алгоритм Краскала может выращивать многочисленные деревья параллельно и в конкурентных условиях, сливая результаты в единое дерево только в конце алгоритма. Таким образом, в отличие от алгоритма Прима, который был ограничен выбором самого дешевого ребра, пересекающего текущую границу, алгоритм Краскала свободен в выборе самого дешевого оставшегося ребра во всем графе, отдавая предпочтение самому дешевому ребру, не создающему цикл.

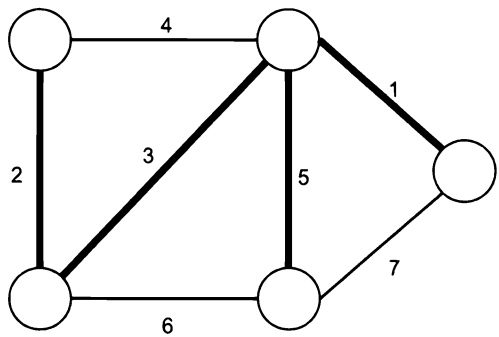
В нашем примере алгоритм Краскала начинает с пустого множества ребер T и на своей первой итерации жадно рассматривает самое дешевое ребро (ребро со стоимостью 1) и добавляет его в T. Вторая итерация следует той же процедуре, выбирая следующее самое дешевое ребро (ребро со стоимостью 2). В этом месте множество T текущего решения выглядит так:



Два ребра, выбранные к настоящему моменту, не пересекаются, поэтому алгоритм практически выращивает два дерева параллельно. На следующей итерации рассматривается ребро со стоимостью 3. Его включение не создает цикла, случайным образом соединяя два текущих дерева:



Далее алгоритм рассматривает ребро со стоимостью 4. Добавление этого ребра в T создаст цикл (с участием ребер со стоимостями 2 и 3), поэтому алгоритм вынужден его пропустить. Следующим оптимальным вариантом является ребро со стоимостью 5; его включение не создает цикла и, по сути дела, приводит к остовному дереву:



Алгоритм пропускает ребро со стоимостью 6 (которое создало бы треугольник с участием ребер со стоимостями 3 и 5), а также конечное ребро со стоимостью 7 (которое создало бы треугольник с участием ребер со стоимостями 1 и 5). Приведенный выше конечный выход представляет собой минимальное остовное дерево графа.

15.5.2. Псевдокод

KRUSKAL

Вход: связный неориентированный граф $G = (V, E)$, представленный в виде списков смежности, и стоимость c_e для каждого ребра $e \in E$.

Выход: ребра минимального остовного дерева графа G.

```
// Предобработка
T := ∅
отсортировать ребра E по стоимости // напр., сортировкой
                                    // слиянием MergeSort
// Главный цикл
for каждый e ∈ E, в неуменьшающемся порядке стоимости do
    if T ∪ {e} является ациклическим then
        T := T ∪ {e}
return T
```

Алгоритм Краскала последовательно рассматривает ребра входного графа (от самого дешевого до самого дорогого), поэтому на этапе предобработки имеет смысл отсортировать их в неуменьшающемся порядке стоимости (используя предпочтительный алгоритм сортировки; см. сноску 1 на с. 24 в главе 13). Связи между ребрами можно разрывать произвольно. Основной цикл проходит через ребра в этом порядке, добавляя ребро в текущее решение, если оно не создает цикл[1]. В итоге алгоритм Краскала возвращает остовное дерево.

Теорема 15.11 (правильность алгоритма `Kruskal`). *Для каждого связного графа $G = (V, E)$ и вещественно-значных реберных стоимостей алгоритм Краскала возвращает минимальное остовное дерево графа G.*

[1] Одна несложная оптимизация: алгоритм может быть остановлен раньше, в момент добавления $|V| - 1$ ребер в T, поскольку T уже будет являться остовным деревом (согласно следствию 15.9).

Наиболее трудоемкая работа уже проделана в процессе доказательства правильности алгоритма Прима (теорема 15.1). В разделе 15.7 приводятся остальные детали доказательства теоремы 15.11.

15.5.3. Простая реализация

УПРАЖНЕНИЕ 15.4

Какое из следующих ниже времен выполнения лучше всего описывает простую реализацию алгоритма Краскала отыскания минимального остовного дерева для графов, представленных в виде списков смежности? Как обычно, n и m обозначают соответственно число вершин и ребер входного графа.

а) $O(m \log n)$

б) $O(n^2)$

в) $O(mn)$

г) $O(m^2)$

(Решение и пояснение см. ниже.)

Правильный ответ: (в). На этапе предобработки алгоритм сортирует массив ребер входного графа, который содержит m элементов. При хорошем алгоритме сортировки (например, сортировке слиянием `MergeSort`) этот шаг вносит вклад в $O(m \log n)$-работу в совокупное время выполнения[1]. Эта работа всецело зависит от того, что делается главным циклом алгоритма, который мы проанализируем далее.

Главный цикл имеет m итераций. Каждая итерация отвечает за проверку того, можно или нельзя добавить рассматриваемое ребро $e = (v, w)$ в текущее реше-

[1] Между $O(m \log n)$ и $O(m \log m)$ в действительности нет никакой разницы. Число ребер связного графа с n вершинами без параллельных ребер не меньше $n - 1$ (достигается с помощью дерева) и не более $\binom{n}{2} = n(n-1)/2$ (достигается с помощью полного графа). Таким образом, $\log m$ лежит между $\log (n-1)$ и $2 \log n$ для каждого связного графа без параллельных ребер, что оправдывает взаимозаменяемость использования $\log m$ и $\log n$ в составе O-большого.

ние T без создания цикла. По лемме 15.7 добавление e в T создает цикл тогда и только тогда, когда T уже содержит путь v–w. Последнее условие может быть проверено за линейное время с использованием любого разумного алгоритма поиска в графе, такого как поиск в ширину или в глубину, начиная с v (см. главу 8 *части 2*). Под «линейным временем» подразумевается линейный размер графа (V, T), который, как ациклический граф с n вершинами, имеет не более $n - 1$ ребер. Таким образом, время выполнения за итерацию равно $O(n)$, давая совокупное время выполнения $O(mn)$.

Утверждение 15.12 (время выполнения алгоритма `Kruskal` (в простой реализации)). *Для каждого графа $G = (V, E)$ и вещественных реберных стоимостей простая реализация алгоритма* `Kruskal` *выполняется за время $O(mn)$, где $m = |E|$ и $n = |V|$.*

*15.6. Ускорение алгоритма Краскала с помощью структуры данных Union-Find

Как и в случае с алгоритмом Прима, время выполнения алгоритма Краскала можно уменьшить с разумной полиномиальной границы $O(mn)$ (утверждение 15.12) до невероятно быстрой почти линейной границы $O(m \log n)$ за счет оптимизации использования структуры данных. Ни одна из структур данных, рассмотренных ранее в этой серии книг, не подходит для этой работы; нам понадобится новая, оптимизированная структура, именуемая структурой данных *Union-Find*, или непересекающимися множествами[1].

Теорема 15.13 (время выполнения алгоритма `Kruskal` (в реализации на основе структуры данных Union-Find)). *Для каждого графа $G = (V, E)$ и вещественных реберных стоимостей реализация алгоритма* `Kruskal` *на основе структуры данных Union-Find выполняется за время $O((m + n) \log n)$, где $m = |E|$ и $n = |V|$*[2].

---

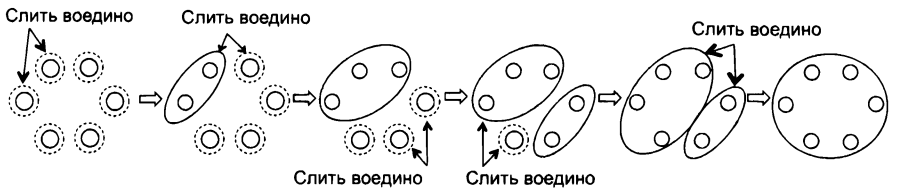
[1] Известна также как структура данных с *несвязным* множеством.

[2] По-прежнему полагая, что входной граф связан, упростим $O((m + n) \log n)$ до $O(m \log n)$.

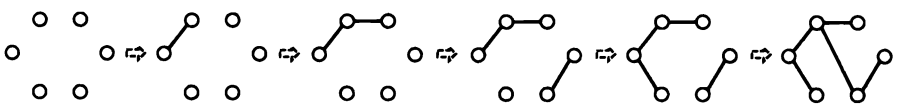
15.6.1. Структура данных Union-Find

При выполнении сложных вычислений всякий раз возникает вопрос об оптимизации структуры данных с целью ускорения этих вычислений. На каждой итерации главного цикла алгоритм Прима выполняет минимальные вычисления, поэтому структура данных кучи является очевидным совпадением. Каждая итерация алгоритма Краскала выполняет проверку цикла или, что эквивалентно, проверку пути. Добавление ребра (v, w) в текущее решение T создает цикл, лишь если T уже содержит путь v–w.

Какая структура данных позволит нам быстро выявлять наличие пути между данной парой вершин в текущем решении? Смысл структуры данных Union-Find заключается в поддержании подразделения статического множества объектов[1]. В начальном подразделении каждый объект находится в своем собственном множестве. Эти множества с течением времени могут объединяться, но они никогда не могут разделяться:



В нашем приложении ускорения алгоритма Краскала объекты будут соответствовать вершинам входного графа, а множества в подразделении — связным компонентам T текущего решения:



[1] *Разделение* множества X объектов предполагает их распределение на подмножества (частный случай — включение всех объектов в одно подмноджество), то есть в наборе S_1, S_2, ..., S_p непустых подмножеств X каждый объект $x \in X$ принадлежит только одному из подмножеств.

Проверка того, содержит ли T уже путь v–w, сводится к проверке того, принадлежат ли v и w одному и тому же множеству подразделения (одной и той же связной компоненте). Для доступа к подразделению и его изменения структура данных Union-Find поддерживает операции Объединить и Найти, вынесенные в ее название.

СТРУКТУРА ДАННЫХ UNION-FIND: ПОДДЕРЖИВАЕМЫЕ ОПЕРАЦИИ

Инициализировать: с учетом массива X объектов создать структуру данных непересекающихся множеств, в которой каждый объект $x \in X$ находится в своем собственном множестве.

Найти: с учетом структуры данных непересекающихся множеств и объекта x в ней вернуть имя множества, которое содержит x.

Объединить: с учетом структуры данных непересекающихся множеств и двух объектов $x, y \in X$ в ней объединить множества, которые содержат x и y, в единое множество[1].

При хорошей реализации операции Объединить и Найти занимают время, логарифмическое по числу объектов[2].

Теорема 15.14 (время выполнения операций Объединить и Найти). *В структуре данных Union-Find с n объектами операции Инициализировать, Найти и Объединить выполняются соответственно за время $O(n)$, $O(\log n)$ и $O(\log n)$.*

Подводя итог, вот перечень показателей.

[1] При нахождении x и y в одной и той же области раздела операция безрезультатна.

[2] Границы для черновой реализации, представленной в разделе 15.6.4. Более совершенные варианты в данном случае являются избыточными (см. видео на сайте www.algorithmsilluminated.org для более подробного ознакомления с современными структурами данных, сформированными по схемам «объединение по рангу», «сжатие пути» и «обратная функция Аккермана»).

Таблица 15.1. Структура данных Union-Find: поддерживаемые операции и время их выполнения, где n обозначает число объектов

Операция	Время выполнения
Инициализировать	$O(n)$
Найти	$O(\log n)$
Объединить	$O(\log n)$

Сначала мы покажем, как реализовать алгоритм Краскала с учетом структуры данных Union-Find с логарифмически-временными операциями, а затем обрисуем реализацию такой структуры данных.

15.6.2. Псевдокод

Главная идея ускорения алгоритма Краскала — в использовании структуры данных Union-Find для отслеживания связных компонент текущего решения. В начале алгоритма каждая вершина находится в своей собственной связной компоненте, и соответственно структура данных Union-Find рождается с каждым объектом в другом множестве. Всякий раз, когда новое ребро (v, w) добавляется в текущее решение, связные компоненты v и w сливаются в одну, и соответственно для обновления структуры данных Union-Find достаточно одной операции Объединить. Проверка того, создаст ли добавление ребра (v, w) цикл, эквивалентна проверке того, находятся ли v и w уже в одной связной компоненте, то есть сводится к двум операциям Найти.

АЛГОРИТМ KRUSKAL (НА ОСНОВЕ СТРУКТУРЫ ДАННЫХ UNION-FIND)

Вход: связный неориентированный граф $G = (V, E)$, представленный в виде списков смежности, и стоимость c_e для каждого ребра $e \in E$.

Выход: ребра минимального остовного дерева G.

```
// Инициализация
T := ∅
```

```
U := Инициализировать(V) // структура данных Union-Find
отсортировать ребра E по стоимости // например, с помощью
// сортировки слиянием MergeSort
// Главный цикл
for каждый (v, w) ∈ E, в неубывающем порядке стоимости do
    if Найти(U, v) ≠ Найти(U, w) then
        // в T нет v-w-пути, поэтому можно добавить (v, w)
        T := T ∪ {(v, w)}
        // обновить из-за слияния компонент
Объединить(U, v, w)
return T
```

Алгоритм поддерживает инвариант, при котором множества структуры данных Union-Find U в начале итерации цикла соответствуют связным компонентам (V, T). Таким образом, условие Найти $(U, v) \neq$ Найти (U, w) удовлетворяется тогда и только тогда, когда v и w находятся в разных связных компонентах (V, T), или, что эквивалентно, тогда и только тогда, когда добавление (v, w) в T не создает цикла. Таким образом, реализация алгоритма `Kruskal` на основе структуры данных Union-Find верна своей первоначальной реализации, причем обе версии дают одинаковый результат.

15.6.3. Анализ времени выполнения

Анализ времени выполнения реализации алгоритма Краскала на основе структуры данных Union-Find прост. Инициализация структуры данных Union-Find занимает время $O(n)$. Как и в первоначальной реализации, этап сортировки требует времени $O(m \log n)$ (см. упражнение 15.4). Главный цикл имеет m итераций, и каждая использует две операции Найти (при суммарном значении $2m$). Существует одна операция Объединить для каждого ребра, добавленного в выход, который, как ациклический граф, имеет не более $n - 1$ ребер (следствие 15.8). При условии, что операции Найти и Объединить выполняются за время $O(\log n)$, как это гарантируется теоремой 15.14, суммарное время выполнения составляет:

предобработка	$O(n)+O(m\log n)$
$2n$ операции НАЙТИ	$O(m\log n)$
$n-1$ операции ОБЪЕДИНИТЬ	$O(m\log n)$
+ оставшаяся служебная работа	$O(m)$
итого	$O((m+n)\log n)$.

Это соответствует границе времени выполнения, обещанной в теореме 15.13. *Ч. Т. Д.*

15.6.4. Быстрая и приближенная реализация структуры данных Union-Find

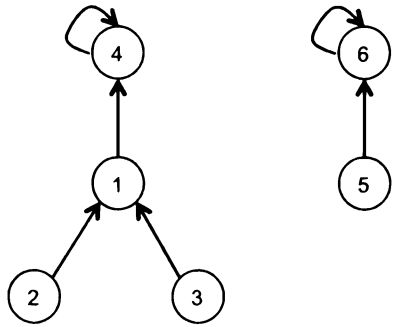
Родительский граф

Под капотом структура данных Union-Find реализована в виде массива и может быть визуализирована как множество ориентированных деревьев. Массив имеет одну позицию для каждого объекта $x \in X$. Каждый элемент массива имеет *родительское* поле, в котором хранится индекс массива некоторого объекта $y \in X$ (где разрешено $y = x$). Затем мы можем представить текущее состояние структуры данных как ориентированный граф — *родительский граф*, — в котором вершины соответствуют (индексам) объектов $x \in X$, и ориентированное ребро (x, y) называется *родительским ребром* всякий раз, когда *родитель*$(x) = y$[1]. Например, если X имеет шесть объектов и текущее состояние структуры данных:

Индекс объекта x	Родитель(x)
1	4
2	1
3	1
4	4
5	6
6	6

[1] Родительский граф — плод нашего воображения. Не путайте его с реальным (неориентированным) входным графом в алгоритме Краскала.

то родительский граф представляет собой пару непересекающихся деревьев, корни которых указывают на себя:



В общем случае множества в подразделении, поддерживаемые структурой данных, будут соответствовать деревьям в родительском графе, причем каждое множество наследует имя своего корневого объекта. Деревья не обязательно являются бинарными, так как нет ограничений на число объектов, которые могут иметь одного и того же родителя. В приведенном выше примере первые четыре объекта принадлежат множеству с именем «4», а последние два — множеству с именем «6».

«Инициализировать» и «Найти»

Вмененная семантика родительского графа предопределяет, как должны быть реализованы операции Инициализировать и Найти.

ИНИЦИАЛИЗИРОВАТЬ

Для каждого $i = 1, 2, ..., n$ инициализировать *родитель*(i) в i.

Операция Инициализировать выполняется за время $O(n)$. Начальный родительский граф состоит из изолированных вершин с петлями:

Для операции Найти мы перепрыгиваем от родителя к родителю до тех пор, пока не достигнем корневого объекта, который можно идентифицировать по его петле.

НАЙТИ

1. Начиная с позиции x в массиве, многократно пройти по родительским ребрам до достижения позиции j, где *родитель*(j) = j.
2. Вернуть j.

Если в нашем текущем примере операция Найти вызывается для третьего объекта, то указанная операция проверяет позицию 3 (где *родитель*(3) = 1), затем позицию 1 (где *родитель*(1) = 4) и, наконец, возвращает позицию 4 (корень, как *родитель*(4) = 4). Определим *глубину* объекта x как число обходов родительских ребер, выполненных операцией Найти из x. В нашем примере четвертый и шестой объекты имеют глубину 0, первый и пятый объекты — глубину 1, а второй и третий объекты — глубину 2. Операция Найти выполняет $O(1)$-работу за обход родительского ребра, поэтому ее наихудшее время выполнения пропорционально наибольшей глубине любого объекта, что эквивалентно наибольшей *высоте* одного из деревьев в родительском графе.

УПРАЖНЕНИЕ 15.5

Каково время выполнения операции Найти в зависимости от числа n объектов?

а) $O(1)$

б) $O(\log n)$

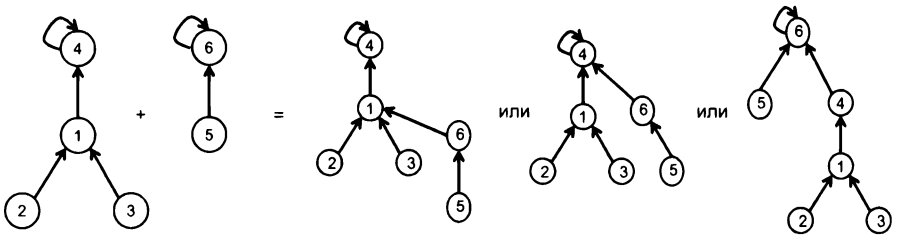
в) $O(n)$

г) для ответа недостаточно информации

(Решение и пояснение см. в разделе 15.6.5.)

Объединить

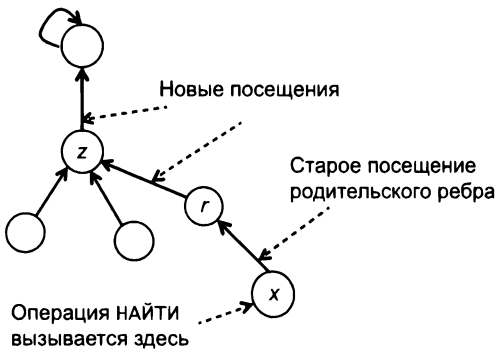
Когда операция Объединить вызывается с объектами x и y, содержащие их два дерева, T_1 и T_2, родительского графа должны быть объединены в одно дерево. Самое простое решение — «понизить» корень одного из деревьев и «повысить» корень другого. Например, если мы решили понизить корень T_1, то он устанавливается как дочерний элемент объекта в другом дереве T_2, то есть его *родительское* поле переназначается из индекса собственного массива в индекс объекта в T_2. Повышенный корень (из T_2) продолжает служить корнем объединенного дерева. Объединить два дерева таким образом можно несколькими способами, например:



Для завершения реализации необходимо принять два решения:

1. Какой из двух корней мы повышаем?
2. Под каким объектом мы устанавливаем пониженный корень?

Предположим, мы устанавливаем корень дерева T_1 под объектом z другого дерева T_2. Каковы последствия для времени выполнения операции Найти? Для объекта в T_2 их нет: операция проходит точно такое же множество родительских ребер, как и раньше. Для объекта x, который ранее был расположен в T_1, операция Найти проходит тот же путь, что и раньше (от x до старого корня r дерева T_1), плюс новое родительское ребро из r в z, плюс родительские ребра из z в корень T_2:



То есть глубина каждого объекта в T_1 увеличивается на 1 (для нового родительского ребра) плюс глубина z.

Ответ на второй вопрос теперь ясен: установить пониженный корень непосредственно под повышенным корнем (с нулевой глубиной), так чтобы обитатели дерева T_1 претерпели увеличение глубины только на 1.

УПРАЖНЕНИЕ 15.6

Предположим, что мы произвольно выбираем корень, который мы будем повышать. Каково время выполнения операции Найти в зависимости от числа n объектов?

а) $O(1)$

б) $O(\log n)$

в) $O(n)$

г) для ответа недостаточно информации

(Решение и пояснение см. в разделе 15.6.5.)

Решение упражнения 15.6 демонстрирует, что для достижения желаемого логарифмического времени выполнения нам нужна другая идея. Если мы понизим корень T_1, то обитатели T_1 будут отодвинуты на один шаг дальше от нового корня; в противном случае обитателей T_2 постигнет та же участь. Кажется, будет справедливым минимизировать число объектов, претерпевающих увеличение глубины, то есть мы должны понизить корень меньшего дерева (произвольно разрывая совпадения значений)[1]. Для этого нам нужен легкий доступ к популяциям двух деревьев. Поэтому наряду с полем «*родитель*» структура данных хранит с каждой записью массива поле «*размер*», инициализированное в операции Инициализировать значением 1. При слиянии

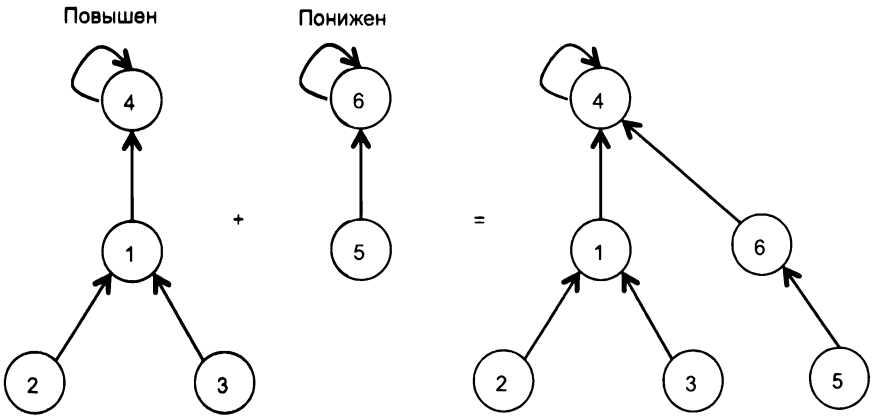
[1] Вариант реализации, известный как «объединение по размеру». Другой, сходный по результативности вариант — объединение по рангу, понижающее корень дерева с меньшей высотой (произвольно разрывая связи). Объединение по рангу подробно обсуждается в видео на www.algorithmsilluminated.org.

двух деревьев поле «*размер*» повышаемого корня соответствующим образом обновляется значением объединенного размера двух деревьев[1].

ОБЪЕДИНИТЬ

1. Вызвать операцию Найти дважды, локализовав позиции i и j корней родительских графовых деревьев, содержащих соответственно x и y. Если $i = j$, то вернуть.
2. Если *размер*(i) $\geq$ *размер*(j), то установить *родитель*(j) := i и *размер*(i) := *размер*(i) + *размер*(j).
3. Если *размер*(i) < *размер*(j), то установить *родитель*(i) := j и *размер*(j) := *размер*(i) + *размер*(j).

В нашем текущем примере (с. 118) эта реализация операции Объединить повышает корень 4 и понижает корень 6, в результате чего:



Операция Объединить выполняет две операции, Найти и $O(1)$, — дополнительную работу, поэтому ее время выполнения совпадает со временем операции Найти.

[1] Сохранять поле размера точным, после того как корень был «преобразован» в НЕ-корень, не требуется.

УПРАЖНЕНИЕ 15.7

При приведенной выше реализации операции Объединить каково время выполнения операции Найти (и следовательно операции Объединить) как функции от числа n объектов?

а) $O(1)$

б) $O(\log n)$

в) $O(n)$

г) для ответа недостаточно информации

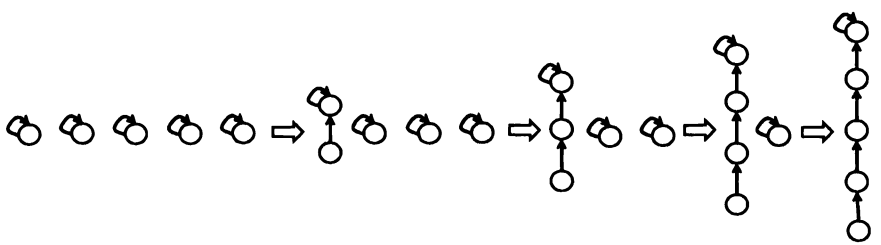
(Решение и пояснение см. в разделе 15.6.5.)

С решением упражнения 15.7 мы приходим к выводу, что наша быстрая и приближенная реализация структуры данных Union-Find соблюдает гарантии времени выполнения, обещанные теоремой 15.14 и табл. 15.1.

15.6.5. Решения упражнений 15.5–15.7

Решение упражнения 15.5

Правильный ответ: (в) или (г). Наихудшее время выполнения операции Найти пропорционально наибольшей высоте дерева родительского графа. Насколько оно может быть большим? Ответ зависит от того, как мы реализуем операцию Объединить; в этом смысле ответ (г) является правильным. Плохая реализация может привести к дереву высотой $n - 1$:



В этом смысле ответ (в) тоже является правильным.

Решение упражнения 15.6

Правильный ответ: (в). При произвольных решениях о повышении и понижении последовательность $n - 1$ операций Объединить может производить дерево высотой $n - 1$, показанное в решении упражнения 15.5, при этом каждая операция устанавливает текущее дерево под ранее изолированным объектом.

Решение упражнения 15.7

Правильный ответ: (б). Каждый объект x начинается с глубины 0. Только один тип события может увеличить глубину x: операция Объединить, в которой корень дерева x в родительском графе понижается. По нашему критерию повышения это происходит, когда наше дерево сливается с другим деревом сопоставимых размеров. Иначе говоря:

> *всякий раз, когда глубина x увеличивается, популяция дерева x по крайней мере удваивается.*

Поскольку популяция не может превышать суммарное число n объектов, глубина x не может быть увеличена более чем в $\log_2 n$ раз. Поскольку время выполнения операции Найти пропорционально глубине объекта, его наихудшее время выполнения равно $O(\log n)$.

*15.7. Алгоритм Краскала: доказательство правильности

Докажем правильность алгоритма Краскала (теорема 15.11) в условиях разной стоимости. Теорема 15.11 может быть доказана в общем виде при несколько бóльших усилиях (см. упражнение 15.5). Сначала покажем, что выход алгоритма является связным (и поскольку он явно ациклический, является остовным деревом). Следующая лемма показывает, что как только ребро (v, w) обрабатывается алгоритмом `Kruskal`, текущее решение (и, следовательно, конечный выход) с необходимостью содержит путь v–w.

Лемма 15.15 (соединение смежных вершин). *Пусть T — множество ребер, выбираемых алгоритмом* `Kruskal` *с точностью до итерации, которая ис-*

следует ребро $e = (v, w)$. Тогда v и w находятся в одной связной компоненте графа (V, T).

Доказательство: в терминологии леммы 15.7 добавление e в текущее решение является либо добавлением с типом C, либо добавлением с типом F. В первом случае v и w уже принадлежат одной и той же связной компоненте до рассмотрения e. Во втором случае алгоритм добавит ребро (v, w) в текущее решение (по лемме 15.7 (б), это не создает цикла), непосредственно соединяя v и w и сливая их связные компоненты воедино. *Ч. Т. Д.*

Другое следствие расширяет лемму 15.15 от индивидуальных ребер до многопереходных путей.

Следствие 15.16 (от ребер к путям). *Пусть P равно пути v–w в G, а T — множеству ребер, выбранных алгоритмом* `Kruskal` *с точностью до последней итерации, которая исследует ребро P. Тогда v и w находятся в одной и той же связной компоненте графа (V, T).*

Доказательство: обозначим ребра P через $(x_0, x_1), (x_1, x_2), ..., (x_{p-1}, x_p)$, где x_0 — это v, а x_p — это w. По лемме 15.15 сразу после итерации, обрабатывающей ребро (x_{i-1}, x_i), x_{i-1} и x_i лежат в одной и той же связной компоненте текущего решения (в последующие итерации включается больше ребер). После того как все ребра P обработаны, все его вершины — и, в частности, его конечные точки v и w — принадлежат одной и той же связной компоненте (V, T) текущего решения. *Ч. Т. Д.*

Теперь докажем, что алгоритм `Kruskal` выводит остовное дерево.

Лемма 15.17 (алгоритм `Kruskal` выводит остовное дерево). *Для каждого связного входного графа алгоритм* `Kruskal` *выводит остовное дерево.*

Доказательство: указанный алгоритм явно гарантирует, что его конечный выход T является ациклическим. Чтобы доказать, что его выход является связным, предположим, что все его вершины принадлежат одной и той же связной компоненте (V, T). Зададим пару v, w вершин; поскольку входной граф является связным, он содержит v–w-путь P. По следствию 15.16, как только алгоритм `Kruskal` обработал каждое ребро P, его конечные точки v и w принадлежат одной и той же связной компоненте текущего решения (и, следовательно, конечного выхода (V, T)). *Ч. Т. Д.*

Чтобы применить теорему 15.6, нужно доказать, что каждое ребро, выбираемое алгоритмом `Kruskal`, удовлетворяет свойству минимального узкого места (MBP)[1].

Лемма 15.18 (алгоритм `Kruskal` достигает свойства минимального узкого места). *Для каждого связного графа $G = (V, E)$ и вещественных реберных стоимостей каждое ребро, выбираемое алгоритмом* `Kruskal`, *удовлетворяет свойству минимального узкого места.*

Доказательство: предположим, что выход алгоритма `Kruskal` никогда не включает ребро, которое не удовлетворяет свойству минимального узкого места. Пусть $e = (v, w)$ равно такому ребру, а P — v–w-пути в G, в котором каждое ребро стоит меньше c_e. Поскольку алгоритм `Kruskal` сканирует ребра в порядке неубывания стоимости, указанный алгоритм обрабатывает каждое ребро P перед e. Из следствия 15.16 вытекает, что к тому времени, когда алгоритм `Kruskal` достигает ребра e, его конечные точки v и w уже принадлежат одной и той же связной компоненте T текущего решения. Добавление e в T создаст цикл (лемма 15.7 (а)), поэтому алгоритм `Kruskal` исключает это ребро из своего выхода. *Ч. Т. Д.*

Сведя все воедино, мы доказываем теорему 15.11 для графов с разными реберными стоимостями.

Доказательство теоремы 15.11: лемма 15.17 доказывает, что выход алгоритма `Kruskal` является остовным деревом. Из леммы 15.18 следует, что каждое ребро этого остовного дерева удовлетворяет свойство минимального узкого места. Теорема 15.6 гарантирует, что это остовное дерево является минимальным остовным деревом. *Ч. Т. Д.*

15.8. Применение: кластеризация с одиночной связью

Обучение без учителя — это отрасль машинного обучения и статистики, призванная понимать крупные коллекции точек данных, находя в них скрытые

[1] Ребро $e = (v, w)$ в графе G удовлетворяет MBP тогда и только тогда, когда каждый путь v–w в G имеет ребро со стоимостью по крайней мере c_e (см. вновь раздел 15.4).

паттерны. Каждая точка данных может представлять человека, снимок, документ, геномную последовательность и т. д. Например, точка данных, соответствующая цветному снимку размером 100 на 100 пикселов, может быть трехмерным вектором с тремя координатами на пиксел, хранящим интенсивности красного, зеленого и синего в этом пикселе[1]. В данном разделе подчеркивается связь между одним из самых элементарных алгоритмов в обучении без учителя и алгоритмом Краскала отыскания минимального остовного дерева.

15.8.1. Кластеризация

Одним из широко используемых подходов к обучению без учителя является *кластеризация*, цель которой состоит в разбиении точек данных на «когерентные группы» (именуемые *кластерами*) «похожих точек» (рис. 15.8). Предположим, что у нас есть *функция подобия* f, которая присваивает неотрицательное вещественное число каждой паре точек данных. Мы исходим из того, что f является *симметричной*, то есть $f(x, y) = f(y, x)$ для каждой пары x, y точек данных. Тогда мы можем интерпретировать точки x, y с малым значением $f(x, y)$ как «похожие», а точки с большим значением как «непохожие»[2]. Например, если точки данных являются векторами с общей размерностью, как в приведенном выше примере со снимком, то $f(x, y)$ можно определить как евклидово (то есть по прямой линии) расстояние между x и y[3]. В качестве еще одного примера раздел 17.1 определяет *расстояние Нидлмана—Вунша* (Needleman—Wunsch), которое является симметричной функцией подобия, предназначенной для геномных последовательностей. В идеальном кластере точки данных в одном кластере относительно похожи, в то время как точки в разных кластерах относительно различны. Пусть k обозначает число требуемых кластеров. Разумные значения для k варьируются от 2 до крупного числа, в зависимости от применения.

---

[1] *Обучение с учителем* фокусируется на прогнозировании, а не на поиске шаблонов как таковых. Подразумевается, что каждая точка данных имеет метку (например, 1, если на изображении, допустим, кошка, и 0 в противном случае), а цель — в том, чтобы научиться безошибочно прогнозировать метки отсутствующих точек данных.

[2] В контексте содержания правильнее вести речь о функции несхожести.

[3] Для $x = (x_1, x_2, ..., x_d)$ и $y = (y_1, y_2, ..., y_d)$, являющихся d-мерными векторами, точная формула будет иметь вид: $f(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$.

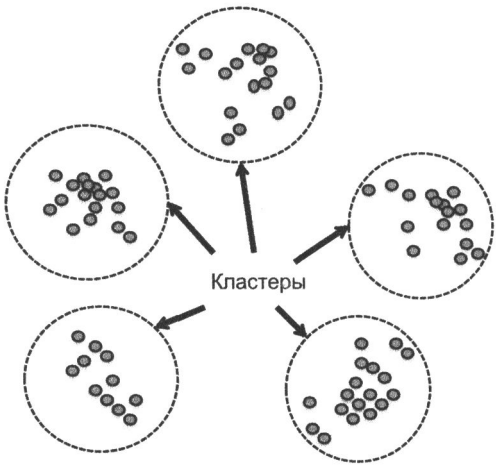


Рис. 15.8. В идеальной кластеризации точки данных в одном кластере относительно похожи, в то время как точки данных в разных кластерах относительно различны

Например, чтобы кластеризовать сообщения в блогах о политике США в группы «левых» и «правых» сообщений, имеет смысл выбрать $k = 2$. Чтобы кластеризовать коллекцию разнообразных снимков в соответствии с их предметом, следует использовать более крупное значение k. Если же вы не уверены в лучшем значении для k, то можете попробовать несколько разных вариантов и выбрать свой любимый среди полученных подразделений.

15.8.2. Восходящая кластеризация

Главная идея *восходящей* (снизу вверх), или *агломеративной*, кластеризации заключается в том, чтобы начать с каждой точки данных в своем собственном кластере, а затем последовательно объединять пары кластеров, до тех пор пока не останется ровно k.

ВОСХОДЯЩАЯ КЛАСТЕРИЗАЦИЯ (ОБОБЩЕННАЯ)

Вход: множество X точек данных, симметричная функция подобия f и положительное целое число $k \in \{1, 2, 3, ..., |X|\}$.

Выход: подразделение X на k непустых множеств.

```
C := ∅                  // отслеживает текущие кластеры
for каждый x ∈ X do
    добавить {x} в C    // каждая точка является своим
                        // собственным кластером
// Главный цикл
while C содержит более чем k кластеров do
    удалить кластеры S1, S2 из C    // детализирует ТВА
    добавить S1 ∪ S2 в C            // слить кластеры воедино
return C
```

Каждая итерация главного цикла уменьшает число кластеров в C на 1, поэтому суммарное число итераций равняется $|X| - k$ (рис. 15.9)[1].

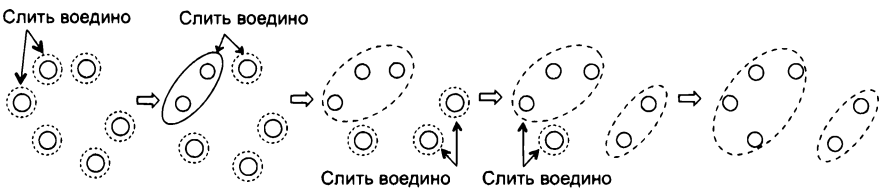


Рис. 15.9. В восходящей кластеризации каждая точка начинается в собственном кластере, и пары кластеров последовательно объединяются, до тех пор пока не останется только k кластеров

Обобщенный алгоритм восходящей кластеризации не определяет, какую пару кластеров объединять на каждой итерации. Можно ли применить жадный подход, и если да, то по какому критерию? Следующий шаг — получение функции подобия F для пар кластеров из заданной функции f для пар точек данных. Одним из простейших вариантов F является сходство по наилучшему случаю между точками в разных кластерах:

$$F(S_1, S_2) = \min_{x \in S_1, y \in S_2} f(x, y). \tag{15.2}$$

[1] Восходящая кластеризация — один из нескольких наиболее распространенных подходов. Так, нисходящие алгоритмы начинаются со всех точек данных в одном кластере, последовательно разделяя кластеры на два до получения требуемого значения k кластеров. Другие алгоритмы — к примеру, кластеризация k средних — поддерживают k кластеров от начала до конца.

Другие разумные варианты F включают сходство по наихудшему либо среднему случаю между точками в разных кластерах. Как только функция F выбрана, обобщенный алгоритм восходящей кластеризации может быть конкретизирован в жадном слиянии «наиболее похожих» пар кластеров на каждой итерации:

ВОСХОДЯЩАЯ КЛАСТЕРИЗАЦИЯ (ЖАДНАЯ)

```
// Главный цикл
```

while C содержит более чем k кластеров **do**

  удалить из C кластеры (S_1, (S_2, которые минимизируют

    $F(S_1, S_2)$ `// например, с помощью` F `как в (15.2)`

  добавить $S_1 \cup S_2$ в C

return C

Кластеризация с одиночной связью относится к жадной восходящей кластеризации с функцией подобия по наилучшему случаю (15.2). Видите ли вы какие-либо связи между кластеризацией по наилучшему случаю и алгоритмом отыскания минимального остовного дерева Краскала (раздел 15.5)? Подумайте над этим вопросом.

* * *

Алгоритм Краскала начинается с пустого множества ребер и с того, что каждая вершина изолирована в своей собственной связной компоненте, так же как кластеризация с одиночной связью начинается с того, что каждая точка данных находится в своем собственном кластере. Каждая итерация алгоритма Краскала, добавляющая новое ребро, сливает две связные компоненты в одну, так же как каждая итерация кластеризации с одиночной связью сливает два кластера в один. Алгоритм Краскала многократно добавляет самое дешевое новое ребро, которое не создает цикл, объединяя компоненты, содержащие его конечные точки, так же как кластеризация с одиночной связью многократно объединяет пару кластеров, содержащих наиболее подобную пару точек данных из разных кластеров. Таким образом, алгоритм Краскала соответствует типу кластеризации, в которой вместо вершин выступают точки данных, а вместо связных компонент — кластеры.

Единственное отличие в том, что кластеризация с одиночной связью останавливается, как только имеется k кластеров, тогда как алгоритм Краскала продолжает работу, пока не останется только одна связанная компонента. Мы заключаем, что *кластеризация с одиночной связью аналогична алгоритму Краскала, останавливаемому досрочно.*

КЛАСТЕРИЗАЦИЯ С ОДИНОЧНОЙ СВЯЗЬЮ ПОСРЕДСТВОМ АЛГОРИТМА КРАСКАЛА

1. Определить полный неориентированный граф $G = (X, E)$ из набора данных X и функции подобия f с множеством вершин X и одним ребром $(x, y) \in E$ со стоимостью $c_{xy} = f(x, y)$ для каждой пары вершин $x, y \in X$.
2. Выполнять алгоритм Краскала с входным графом G до тех пор, пока T текущего решения не будет содержать $|X| - k$ ребер или, что эквивалентно, пока граф (X, T) не будет иметь k связных компонент.
3. Вычислить связные компоненты (X, T) и вернуть соответствующее подразделение X.

ВЫВОДЫ

- Остовное дерево графа — это ациклический подграф, который содержит путь между каждой парой вершин.
- В задаче о минимальном остовном дереве (minimum spanning tree, MST) входом является связный неориентированный граф с вещественными реберными стоимостями, и цель состоит в том, чтобы вычислить остовное дерево с минимально возможной суммой реберных стоимостей.
- Алгоритм Прима строит минимальное остовное дерево по одному ребру за раз, начиная с произвольной вершины, и выращивает его, как плесень, до тех пор пока не будет охвачен весь набор вершин. На каждой итерации он жадно выбирает самое дешевое ребро, которое расширяет охват текущего решения.

- При реализации с кучевой структурой данных алгоритм Прима выполняется за время $O(m \log n)$, где m и n обозначают соответственно число ребер и вершин входного графа.
- Алгоритм Краскала также строит минимальное остовное дерево по одному ребру за раз, жадно выбирая самое дешевое ребро, добавление которого в текущем решении не создает цикла.
- При реализации со структурой данных непересекающихся множеств Union-Find алгоритм Краскала работает за время $O(m \log n)$.
- Первый шаг в доказательствах правильности алгоритмов Прима и Краскала состоит в том, чтобы показать, что каждый алгоритм выбирает только те ребра, которые удовлетворяют свойству минимального узкого места (minimum bottleneck property, MBP).
- Второй шаг — использовать обмен аргументами с целью доказать, что остовное дерево, в котором каждое ребро удовлетворяет свойству минимального узкого места, должно быть минимальным остовным деревом.
- Кластеризация с одиночной связью является жадным методом восходящей кластеризации в неконтролируемом обучении и соответствует алгоритму Краскала, останавливаемому досрочно.

Задачи на закрепление материала

Задача 15.1. Рассмотрим неориентированный граф $G = (V, E)$, в котором каждое ребро $e \in E$ имеет разную и неотрицательную стоимость. Пусть T равно минимальному остовному дереву, а P — кратчайшему пути из некоторой вершины s в некоторую вершину t. Предположим, стоимость каждого ребра e графа G увеличивается на 1 и становится $c_e + 1$. Назовем новый граф G'. Что из нижеследующего является истинным в отношении G'?

а) T должно быть минимальным остовным деревом, и P должен быть кратчайшим путем s–t;

б) T должно быть минимальным остовным деревом, но P не может быть кратчайшим путем s–t;

в) T не может быть минимальным остовным деревом, но P должен быть кратчайшим путем s–t;

г) T не может быть минимальным остовным деревом, и P не может быть кратчайшим путем s–t.

Задача 15.2. Рассмотрим следующий ниже алгоритм, который пытается вычислить минимальное остовное дерево связного неориентированного графа $G = (V, E)$ с разными реберными стоимостями, выполнив алгоритм Краскала «в обратном порядке»:

```
KRUSKAL (ОБРАТНАЯ ВЕРСИЯ)
T := E
отсортировать ребра E в убывающем порядке стоимости
for каждое e ∈ E, по порядку do
    if T − {e} является связным then
        T := T − {e}
return T
```

Какое из следующих ниже утверждений является истинным?

а) Выход алгоритма никогда не будет иметь цикла, но он может быть несвязным;

б) Выход алгоритма всегда будет связным, но он может иметь циклы;

в) Алгоритм всегда выводит остовное дерево, но оно может не быть минимальным остовным деревом;

г) Алгоритм всегда выводит минимальное остовное дерево.

Задача 15.3. Какая из следующих ниже задач простым способом сводится к задаче о минимальном остовном дереве? Выберите все, что применимо.

а) Задача об остовном дереве с максимальной стоимостью. То есть среди всех остовных деревьев T связного графа с реберными стоимостями

вычислить одно такое дерево с максимально возможной суммой $\Sigma_{e \in T} c_e$ реберных стоимостей;

б) Задача об остовном дереве с минимальным произведением. То есть среди всех остовных деревьев T связного графа со строго положительными реберными стоимостями вычислить одно такое дерево с минимально возможным произведением $\prod_{e \in T} c_e$ реберных стоимостей;

в) Задача о кратчайшем пути с единственным истоком. В этой задаче вход содержит связный неориентированный граф $G = (V, E)$, неотрицательную длину ℓ_e для каждого ребра $e \in E$ и назначенную начальную вершину $s \in V$. Требуемым выходом является минимальная суммарная длина пути из s в v для каждого возможного стока $v \in V$;

г) С учетом связного неориентированного графа $G = (V, E)$ с положительными реберными стоимостями вычислить множество минимальных стоимостей $F \subseteq E$ ребер, такое что граф $(V, E - F)$ будет ациклическим.

О РЕДУКЦИЯХ

Задача *A сводится* (или редуцируется) к задаче *B*, если алгоритм, решающий задачу *B*, можно легко перевести в алгоритм, решающий задачу *A*. Например, задача вычисления медианного (срединного) элемента массива сводится к задаче сортировки массива. Редукции являются одним из наиболее важных понятий в изучении алгоритмов и их ограничений и могут иметь большую практическую пользу.

Вы всегда должны быть в поиске редукций. Всякий раз, сталкиваясь с, казалось бы, новой задачей, всегда спрашивайте себя: является ли эта задача замаскированной версией той, решение которой вы уже знаете? Можно ли свести общую версию задачи к частному случаю?

Задачи повышенной сложности

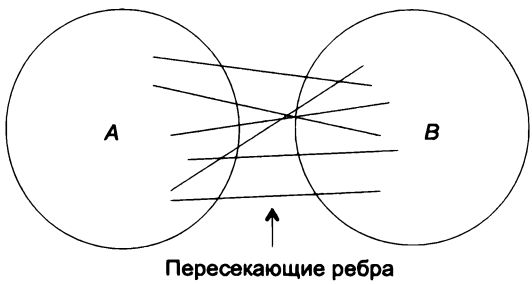
Задача 15.4. Докажите обратное теореме 15.6: если T является минимальным остовным деревом графа с вещественными реберными стоимостями, то каждое ребро T удовлетворяет свойству минимального узкого места.

Задача 15.5. Докажите правильность алгоритмов Прима и Краскала (теоремы 15.1 и 15.11) в полной общности для графов, в которых реберные стоимости не обязательно должны быть разными.

Задача 15.6. Докажите, что в связном неориентированном графе с разными реберными стоимостями существует уникальное минимальное остовное дерево.

Задача 15.7. Альтернативный подход к доказательству правильности алгоритмов Прима и Краскала заключается в использовании так называемого свойства разреза минимальных остовных деревьев (Cut MST). В процессе решения этой задачи будем считать, что реберные стоимости являются разными.

Разрезом (cut) неориентированного графа $G = (V, E)$ является подразделение его множества вершин V на два непустых множества A и B.



Ребро графа G пересекает разрез (A, B), если оно имеет одну конечную точку в каждом из множеств A и B.

СВОЙСТВО РАЗРЕЗА

Пусть $G = (V, E)$ равно связному неориентированному графу с разными вещественными реберными стоимостями. Если ребро $e \in E$ является самым дешевым ребром, пересекающим разрез (A, B), то e принадлежит каждому минимальному остовному дереву графа G[1].

[1] Читатели, решившие задачу 15.6, могут сформулировать вывод несколько иначе: «...тогда e принадлежит MST от G».

Другими словами, одним из способов обоснования для включения алгоритмом ребра e в его решение является получение разреза графа G, для которого e является самым дешевым пересекающим ребром[1].

а) Докажите свойство разреза.

б) Примените свойство разреза, для того чтобы доказать правильность алгоритма Прима.

в) Повторите (б) для алгоритма Краскала.

Задача 15.8. Рассмотрим связный неориентированный граф с разными вещественно-значными реберными стоимостями. *Минимальное по узкому месту остовное дерево* (minimum bottleneck spanning tree, MBST) — это остовное дерево T с минимально возможным узким местом (то есть минимальной максимальной реберной стоимостью $\max_{e \in T} c_e$).

а) (Сложно) Дайте линейно-временной алгоритм вычисления минимального по узкому месту остовного дерева (MBST).

б) Следует ли из этого линейно-временной алгоритм вычисления суммарной стоимости минимального остовного дерева?

Задачи по программированию

Задача 15.9. Реализуйте на любимом языке программирования алгоритмы `Prim` и `Kruskal`. Для получения бонусных баллов реализуйте версию алгоритма `Prim` на основе кучевой структуры данных (раздел 15.3) и версию алгоритма `Kruskal` на основе структуры данных Union-Find (раздел 15.6). Кажется ли вам, что один из алгоритмов явно быстрее другого? Тестовые случаи и наборы данных для сложных задач см. на веб-сайте www.algorithmsilluminated.org.

[1] Существует также свойство Cycle, согласно которому в случае, если ребро e является самым дорогим в некотором цикле C, то каждое MST исключает e. Проверьте эквивалентность названного свойства обратному (см. доказательство теоремы 15.6 в задаче 15.4).

16 Введение в динамическое программирование

Рассмотренные нами парадигмы проектирования алгоритмов («разделяй и властвуй» и «жадные алгоритмы») не охватывают все вычислительные задачи, с которыми вы столкнетесь. Вторая половина этой книги посвящена третьей парадигме проектирования: парадигме *динамического программирования*. Динамическое программирование — высокоэффективный метод, обычно приводящий к наиболее эффективным решениям.

Большинство людей изначально считают динамическое программирование сложным и нелогичным. Для овладения приемами динамического программирования (которое, заметим, является относительно формализованным) требуется практика. В этой и двух следующих главах представлено до полудюжины подробных тематических исследований, включая несколько выдающихся алгоритмов. Вы узнаете, как они работают, добавив в свой программистский арсенал общую и гибкую технику разработки алгоритмов, которую вы можете применить к задачам, возникающим в ваших собственных проектах.

ОБОДРЯЮЩИЕ СЛОВА

Совершенно нормально чувствовать себя в замешательстве, впервые столкнувшись с динамическим программированием. Совет: не тушуйтесь. Не упускайте возможность стать еще умнее.

16.1. Задача о взвешенном независимом множестве

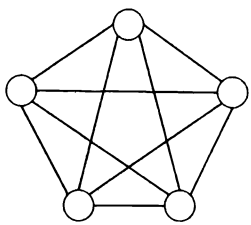
Я пока не собираюсь рассказывать, что такое динамическое программирование. До этого мы разработаем с нуля алгоритм сложной и конкретной вычислительной задачи, требующий новых идей. Решив эту задачу, мы уменьшим масштаб и определим ингредиенты решения, иллюстрирующие общие принципы динамического программирования. Затем, вооружившись шаблоном для разработки алгоритмов динамического программирования и примером создания экземпляров, займемся более сложными приложениями указанной парадигмы.

16.1.1. Определение задачи

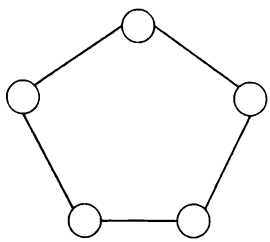
Пусть $G = (V, E)$ равно неориентированному графу. *Независимое множество* графа G есть подмножество $S \subseteq V$ взаимно несмежных вершин: для каждого $v, w \in S$, $(v, w) \notin E$. По аналогии, независимое множество не содержит обеих конечных точек любого ребра G. Например, если вершины представляют людей и ребра — пары людей, которые не нравятся друг другу, то независимые множества соответствуют группам всех людей, которые ладят. Или же, если вершины представляют учебные занятия, о которых вы думаете, и между каждой парой конфликтующих учебных занятий есть ребро, то независимые множества соответствуют возможным расписаниям курсов (при условии, что вы не можете быть в двух местах одновременно).

УПРАЖНЕНИЕ 16.1

Сколько различных независимых множеств имеет полный граф с пятью вершинами?



Как насчет цикла с пятью вершинами?



а) 1 и 2 (соответственно)

б) 5 и 10

в) 6 и 11

г) 6 и 16

(Решение и пояснение см. в разделе 16.1.4).

Теперь сформулируем задачу о *взвешенном независимом множестве* (Weighted Independent Set, WIS):

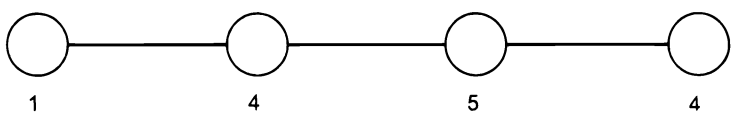
ЗАДАЧА: ВЗВЕШЕННОЕ НЕЗАВИСИМОЕ МНОЖЕСТВО (WIS)

Вход: неориентированный граф $G = (V, E)$ и неотрицательный вес w_v для каждой вершины $v \in V$.

Выход: независимое множество $S \subseteq V$ графа G с максимально возможной суммой $\Sigma_{v \in S} w_v$ вершинных весов.

Оптимальное решение задачи о взвешенном независимом множестве называется *независимым множеством с максимальным весом* (maximum-weight independent set, MWIS). Например, если вершины представляют курсы, веса вершин — единицы, а ребра — конфликты между курсами, то независимое множество с максимальным весом соответствует допустимому расписанию курса с самой тяжелой нагрузкой (в единицах).

Задача о взвешенном независимом множестве является сложной даже в простейшем случае *путевых графов*. Например, вход в задачу может выглядеть следующим образом (с вершинами, помеченными их весами):



Этот граф имеет 8 независимых множеств: пустое множество, четыре одноэлементных множества, первую и третью вершины, первую и четвертую вершины, а также вторую и четвертую вершины. Последнее из них имеет наибольший суммарный вес 8. Число независимых множеств путевого графа

растет экспоненциально вместе с числом вершин (поясните почему), поэтому нет никакой надежды решить эту задачу путем исчерпывающего поиска, за исключением самых крошечных случаев.

16.1.2. Естественный жадный алгоритм оказывается безуспешным

Для многих вычислительных задач жадные алгоритмы — отличный повод для мозгового штурма. Такие алгоритмы, как правило, легко придумать, и даже если с его помощью не удается решить задачу (что бывает), неудача позволяет лучше понять тонкости задачи. Для задачи о взвешенном независимом множестве, пожалуй, наиболее естественным жадным алгоритмом является аналог алгоритма Краскала: выполнить единственный проход по вершинам, от лучшей (с высоким весом) до худшей (с низким весом), добавляя вершину в текущее решение, при условии, что она не конфликтует с ранее выбранной вершиной. С учетом входного графа $G = (V, E)$ с вершинными весами псевдокод выглядит так:

ВЗВЕШЕННОЕ НЕЗАВИСИМОЕ МНОЖЕСТВО: ЖАДНЫЙ ПОДХОД

$S := \emptyset$

отсортировать вершины V по весу

for каждый $v \in V$, в не увеличивающемся порядке веса **do**

  if $S \cup \{v\}$ является независимым множеством графа G **then**

    $S := S \cup \{v\}$

return S

Выглядит все достаточно просто. Но работает ли это?

УПРАЖНЕНИЕ 16.2

Каков суммарный вес на выходе жадного алгоритма, когда входной граф имеет четырехвершинный путь, показанный на с. 140? Является ли он максимально возможным?

а) 6; нет

б) 6; да

в) 8; нет

г) 8; да

(Решение и пояснение см. в разделе 16.1.4.)

Главы 13–15 избаловали нас изобилием правильных жадных алгоритмов, но не забывайте: жадные алгоритмы обычно *не* являются правильными.

16.1.3. Подход «разделяй и властвуй»?

Парадигму проектирования алгоритмов «разделяй и властвуй» (раздел 13.1.1) всегда стоит опробовать в случае задач, в которых есть естественный способ разбить входные данные на более мелкие подзадачи. Для задачи о взвешенном независимом множестве с входным путевым графом $G = (V, E)$ естественным высокоуровневым подходом является (игнорируя базовый случай) следующий:

ВЗВЕШЕННОЕ НЕЗАВИСИМОЕ МНОЖЕСТВО: ПОДХОД «РАЗДЕЛЯЙ И ВЛАСТВУЙ»

G_1 := первая половина графа G

G_2 := вторая половина графа G

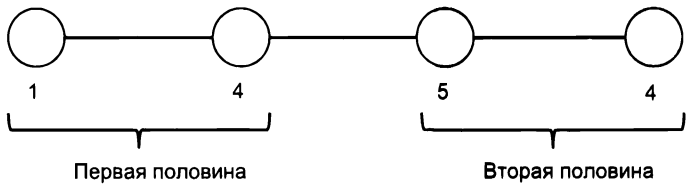
S_1 := рекурсивно решить задачу о взвешенном независимом множестве на G_1

S_2 := рекурсивно решить задачу о взвешенном независимом множестве на G_2

объединить S_1, S_2 в решение S для G

вернуть S

Дьявол кроется в деталях шага совмещения. Возвращаясь к нашему текущему примеру



заметим, что первый и второй рекурсивные вызовы возвращают вторую и третью вершины как оптимальные решения их соответствующих подзадач. Объединение решений *не* является независимым множеством из-за конфликта на границе между двумя решениями. Легко увидеть, как разрядить пограничный конфликт, когда входной граф имеет четыре вершины; когда же он имеет сотни или тысячи вершин, все обстоит сложнее[1].

Можем ли мы добиться лучшего по сравнению с жадным алгоритмом или алгоритмом «разделяй и властвуй»?

16.1.4. Решения упражнений 16.1–16.2

Решение упражнения 16.1

Правильный ответ: (в). Полный граф не имеет несмежных вершин, поэтому каждое независимое множество имеет не более одной вершины. Таким образом, существует шесть независимых множеств: пустое множество и пять одноэлементных множеств. Цикл имеет те же шесть независимых множеств, что и полный граф, плюс несколько независимых множеств размера 2. Каждое подмножество из трех или более вершин имеет пару смежных вершин. Цикл имеет пять независимых множеств размера 2 (как вы можете убедиться сами) при суммарном значении 11.

Решение упражнения 16.2

Правильный ответ: (а). Первая итерация жадного алгоритма берет вершину с максимальным весом, то есть третью вершину (с весом 5). Это исключает

[1] С помощью алгоритма «разделяй и властвуй», выполняющего четыре рекурсивных вызова, а не два, задача может быть решена за время $O(n_2)$, где n — количество вершин. С помощью алгоритма динамического программирования задача будет решена за время $O(n)$.

смежные вершины (вторую и четвертую, обе с весом 4) из дальнейшего рассмотрения. Затем алгоритм застревает, выбирая первую вершину, и выводит независимое множество с общим весом 6, что не является оптимальным, так как вторая и четвертая вершины составляют независимое множество с общим весом 8.

16.2. Линейно-временной алгоритм для взвешенного независимого множества на путях

16.2.1. Оптимальная подструктура и рекуррентное соотношение

Чтобы быстро решить задачу о взвешенном независимом множестве на путевых графах, поразмышляем об идеальном решении. Как оно должно выглядеть? Подобное решение должно быть построено предписанным способом из оптимальных решений меньших подзадач, сужая поле кандидатов до приемлемого числа[1].

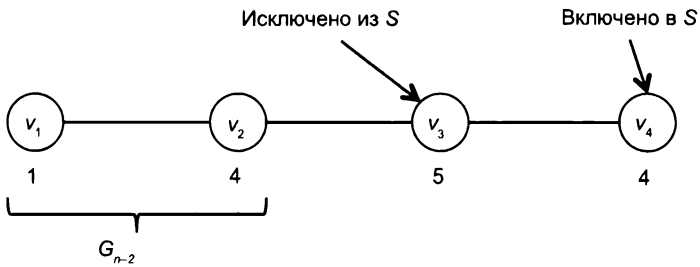
Пусть $G = (V, E)$ обозначает n-вершинный путевой граф с ребрами (v_1, v_2), (v_2, v_3), ..., (v_{n-2}, v_{n-1}), (v_{n-1}, v_n) и неотрицательный вес w_i для каждой вершины $v_i \in V$. Предположим, что $n \geq 2$; в противном случае ответ является очевидным. Предположим, что мы знали независимое множество с максимальным весом (MWIS) $S \subseteq V$ с суммарным весом W. Что можно сказать об этом? Вот тавтология: S либо содержит конечную вершину v_n, либо нет. Рассмотрим эти случаи в обратном порядке.

Случай 1: $v_n \notin S$. Предположим, оптимальному решению S случилось исключить v_n. Возьмем $(n - 1)$-вершинный путевой граф G_{n-1} из G, отщипнув последнюю вершину v_n и последнее ребро (v_{n-1}, v_n). Поскольку S не включает последнюю вершину графа G, оно содержит только вершины G_{n-1} и может рассматриваться как независимое множество G_{n-1} (по-прежнему с суммарным весом W) — и не только любое старое независимое множество G_{n-1}, но

[1] Размышлять следует не над вычисляемым объектом «вообще», а над путями поиска эффективного алгоритма вычислений.

и множество с *максимальным весом*. Ибо если бы S^* было независимым множеством G_{n-1} с суммарным весом $W^* > W$, то S^* также составляло бы независимое множество суммарного веса W^* в большем графе G. Это противоречит предполагаемой оптимальности S. Другими словами, независимое множество с максимальным весом (MWIS), исключающее последнюю вершину, имеет вид MWIS меньшего графа G_{n-1}.

Случай 2: $v_n \in S$. Предположим, S включает последнюю вершину v_n. Как независимое множество, S не может включать две подряд вершины из пути, поэтому оно исключает предпоследнюю вершину: $v_{n-1} \notin S$. Возьмем $(n-2)$-вершинный путевой граф G_{n-2} из G, отбросив последние две вершины и ребра[1]:



Поскольку S содержит v_n, а G_{n-2} нет, мы не можем рассматривать S как независимое множество G_{n-2}. После удаления последней вершины из S, мы получим, что $S - \{v_n\}$ не содержит ни v_{n-1}, ни v_n и, следовательно, может рассматриваться как независимое множество меньшего графа G_{n-2} (с суммарным весом $W - w_n$). Кроме того, $S - \{v_n\}$ должно быть независимым множеством с максимальным весом (MWIS) G_{n-2}. Предположим, S^* было независимым множеством G_{n-2} с суммарным весом $W^* > W - w_n$. Поскольку G_{n-2} (и, следовательно, S^*) исключает предпоследнюю вершину v_{n-1}, беспечное добавление последней вершины v_n в S^* не создаст никаких конфликтов, и поэтому $S^* \cup \{v_n\}$ будет независимым множеством графа G с суммарным весом $W^* + w_n > (W - w_n) + w_n = W$. Это противоречит предполагаемой оптимальности S.

Другими словами, если независимое множество с максимальным весом (MWIS) включает последнюю вершину, то оно имеет вид MWIS меньшего

[1] Если $n = 2$, то G_0 интерпретируется как пустой граф (без вершин и ребер). Единственный независимый набор G_0 — это пустой набор, общий вес которого равен нулю.

графа G_{n-2}, дополненного конечной вершиной v_n. Вывод: два и только два кандидата соперничают за то, чтобы быть множеством с максимальным весом (MWIS).

Лемма 16.1 (оптимальная подструктура взвешенного независимого множества). *Пусть S равно независимому множеству с максимальным весом (MWIS) путевого графа G с не менее чем 2 вершинами. Пусть G_i обозначает подграф графа G, содержащий его первые i вершин и i − 1 ребер. Тогда S является либо*:

(*i*) независимым множеством с максимальным весом (MWIS) G_{n-1},

либо

(*ii*) независимым множеством с максимальным весом (MWIS) G_{n-2}, дополненным конечной вершиной v_n графа G.

Лемма 16.1 выделяет только две возможности для независимого множества с максимальным весом, поэтому любой вариант с бóльшим суммарным весом является оптимальным решением. Отсюда — рекурсивная формула (рекуррентное соотношение) для суммарного веса независимого множества с максимальным весом (MWIS):

Следствие 16.2 (рекуррентное соотношение взвешенного независимого множества). *С допущениями и обозначением леммы 16.1 пусть W_i обозначает суммарный вес независимого множества с максимальным весом (MWIS) графа G_i. При i = 0 интерпретируем W_i как 0. Тогда*

$$W_n = \max\{\underbrace{W_{n-1}}_{\text{Случай 1}}, \underbrace{W_{n-2} + w_n}_{\text{Случай 2}}\}.$$

В более общем случае для каждого $i = 2, 3, ..., n$

$$W_i = \max\{W_{i-1}, W_{i-2} + w_i\}.$$

Более общее утверждение в следствии 16.2 формулируется путем вызова первого утверждения для каждого $i = 2, 3, ..., n$, где G_i играет роль входного графа G.

16.2.2. Наивный рекурсивный подход

Применение леммы 16.1 позволяет сузить область поиска до двух кандидатов на оптимальное решение. Это приводит к следующему ниже псевдокоду, в котором графы G_{n-1} и G_{n-2} определяются, как и раньше.

РЕКУРСИВНЫЙ АЛГОРИТМ ДЛЯ ВЗВЕШЕННОГО НЕЗАВИСИМОГО МНОЖЕСТВА

Вход: путевой граф G с множеством вершин $[v_1, v_2, ..., v_n]$ и неотрицательным весом w_i для каждой вершины v_i.

Выход: независимое множество с максимальным весом графа G.

```
1 if n = 0 then                          // базовый случай #1
2     return пустое множество
3 if n = 1 then                          // базовый случай #2
4     return {v1}
  // рекурсия при n ≥ 2
5 S1 := рекурсивно вычислить MWIS графа G(n-1)
6 S2 := рекурсивно вычислить MWIS графа G(n-2)
7 return S1 или S2 ∪ {vn}, в зависимости от того, что имеет
  больший вес
```

Простое доказательство по индукции показывает, что алгоритм гарантированно вычисляет независимое множество с максимальным весом[1]. Но за какое время?

УПРАЖНЕНИЕ 16.3

Что такое асимптотическое время выполнения рекурсивного алгоритма отыскания взвешенного независимого множества как

[1] Доказательство методом индукции проводится по числу вершин n. Базовые случаи ($n = 0, 1$) не вызывают сомнений. Для индуктивного шага ($n \geq 2$) S_1 и S_2 гарантированно являются MWIS G_{n-1} и G_{n-2} соответственно. Лемма 16.1 подразумевает, что лучшим из S_1 и $S_2 \cup \{v_n\}$ является MWIS из G.

функция числа n вершин? Выберите самое приближенное правильное утверждение:

а) $O(n)$

б) $O(n \log n)$

в) $O(n^2)$

г) ничего из вышеперечисленного

(Решение и пояснение см. в разделе 16.2.5.)

16.2.3. Рекурсия с кэшем

Упражнение 16.3 показывает, что рекурсивный алгоритм отыскания взвешенного независимого множества не лучше, чем исчерпывающий поиск. Упражнение 16.4 содержит ключ к радикальному сокращению времени выполнения.

УПРАЖНЕНИЕ 16.4

Каждый из (экспоненциально многих) рекурсивных вызовов рекурсивного алгоритма отыскания взвешенного независимого множества отвечает за вычисление независимого множества с максимальным весом (MWIS) заданного входного графа. Сколько разных входных графов когда-либо рассматривалось в пределах всех вызовов?

а) $\Theta(1)$ [1]

б) $\Theta(n)$

в) $\Theta(n^2)$

г) $2^{\Theta(n)}$

(Решение и пояснение см. в разделе 16.2.5.)

Из упражнения 16.4 следует, что экспоненциальное время работы нашего рекурсивного алгоритма отыскания взвешенного независимого множества

[1] Обозначение O-большое аналогично «меньше или равно», а Θ-большое аналогично «равно». Формально функция $f(n)$ равна $\Theta(g(n))$, если существуют такие константы c_1 и c_2, что $f(n)$ относятся к диапазону $c_1 \times g(n)$ и $c_2 \times g(n)$ для всех достаточно больших n.

проистекает исключительно из его абсурдной избыточности, многократно решая одни и те же подзадачи с нуля. Но почему бы, решая подзадачу впервые, не сохранить результат в кэше? Если мы столкнемся с той же подзадачей позже, то сможем быстро найти готовое решение[1].

Примешать кэширование в псевдокод со с. 147 легко. Результаты прошлых вычислений хранятся в глобально видимом массиве длины $(n + 1)$ A, при этом $A[i]$ хранит MWIS G_i, где G_i содержит первые i вершин и первые $i - 1$ ребер первоначального входного графа (а G_0 является пустым графом). В строке 6 алгоритм сначала проверяет, содержит ли массив A уже соответствующее решение S_1; если нет, то он вычисляет S_1 рекурсивно, как и раньше, и кэширует результат в A. Схожим образом новая версия строки 7 либо просматривает, либо рекурсивно вычисляет и кэширует S_2 по мере необходимости.

Каждая из $n + 1$ подзадач решается с нуля только один раз. Кэширование, конечно, ускоряет алгоритм, но насколько? При правильной реализации время выполнения падает с экспоненциального до *линейного*. Это резкое ускорение легче увидеть, переформулировав наш нисходящий рекурсивный алгоритм как восходящий итеративный, и последний обычно является как раз тем, который вы хотите реализовать на практике.

16.2.4. Восходящая итеративная реализация

Разобравшись, как встраивать кэширование в наш рекурсивный алгоритм отыскания взвешенного независимого множества, мы выяснили, что существует $n + 1$ релевантных подзадач, соответствующих всем возможным префиксам входного графа (упражнение 16.4).

ВЗВЕШЕННОЕ НЕЗАВИСИМОЕ МНОЖЕСТВО В ПУТЕВЫХ ГРАФАХ: ПОДЗАДАЧИ

Вычислить W_i, суммарный вес независимого множества с максимальным весом (MWIS) префиксного графа G_i.

(Для каждого $i = 0, 1, 2, ..., n$).

[1] Технику кеширования результата вычислений, позволяющую избежать повторов, иногда называют запоминанием.

Сосредоточимся на вычислении суммарного веса независимого множества с максимальным весом (MWIS) для подзадачи. Раздел 16.3 показывает, как выявлять вершины независимого множества с максимальным весом. Теперь, когда мы знаем, какие подзадачи являются важными, почему бы не решить их одну за другой? Решение подзадачи зависит от решения двух меньших подзадач. Чтобы обеспечить доступность этих двух решений, следует начинать с простых случаев и заканчивать все более крупными подзадачами.

ВЗВЕШЕННОЕ НЕЗАВИСИМОЕ МНОЖЕСТВО (WIS)

Вход: путевой граф G с множеством вершин $\{v_1, v_2, ..., v_n\}$ и неотрицательным весом для каждой вершины v_i.

Выход: суммарный вес независимого множества с максимальным весом графа G.

A := массив длиной $(n + 1)$ `// решения подзадач`

$A[0] := 0$ `// базовый случай #1`

$A[1] := w_1$ `// базовый случай #2`

for $i = 2$ to n **do**

`// использовать рекуррентное соотношение`

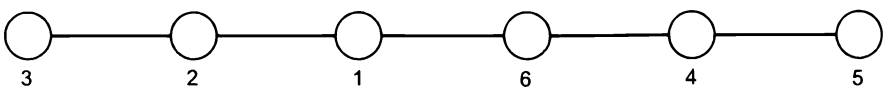
`// из следствия 16.2`

$$A[i] := \max\{\underbrace{A[i-1]}_{\text{Случай 1}}, \underbrace{A[i-2]+w_i}_{\text{Случай 2}}\}$$

return $A[n]$ `// решение наибольшей подзадачи`

Массив A длиной $(n + 1)$ индексируется от 0 до n. К тому времени, когда итерация главного цикла должна вычислить решение подзадачи $A[i]$, значения $A[i - 1]$ и $A[i - 2]$ двух соответствующих меньших подзадач уже были вычислены в предыдущих итерациях (или в базовых случаях). Таким образом, каждая итерация цикла занимает $O(1)$ времени, давая невероятно быстрое время выполнения $O(n)$.

Например, для входного графа



вы должны убедиться сами, что конечными значениями массива являются:

Длина i-го префикса

0	1	2	3	4	5	6
0	3	3	4	9	9	14

По завершении алгоритма `WIS` каждый элемент массива $A[i]$ хранит суммарный вес независимого множества с максимальным весом (MWIS) графа G_i, который содержит первые i вершин и $i-1$ ребер входного графа. Это следует из индукционного аргумента, аналогичного приведенному в сноске на с. 147. Базовые случаи $A[0]$ и $A[1]$ явно верны. При вычислении $A[i]$ с $i \geq 2$ по индукции значения $A[i-1]$ и $A[i-2]$ действительно являются суммарными весами независимых множеств с максимальным весом (MWIS) соответственно графов G_{i-1} и G_{i-2}. Из следствия 16.2 вытекает, что $A[i]$ также вычисляется правильно. В приведенном выше примере суммарный вес MWIS в первоначальном входном графе является значением в конечном элементе массива (14), соответствующим независимому множеству, состоящему из первой, четвертой и шестой вершин.

Теорема 16.3 (свойства алгоритма `WIS`). *Для каждого путевого графа и неотрицательных вершинных весов алгоритм `WIS` выполняется за линейное время и возвращает суммарный вес независимого множества с максимальным весом.*

16.2.5. Решения упражнений 16.3–16.4

Решение упражнения 16.3

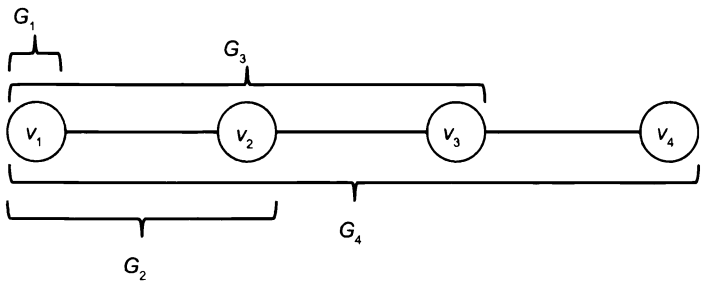
Правильный ответ: (г). Внешне шаблон рекурсии похож на $O(n \log n)$-временной алгоритм «разделяй и властвуй», такой как сортировка слиянием `MergeSort` с двумя рекурсивными вызовами, за которыми следует простой шаг совмещения. Но есть большая разница: алгоритм `MergeSort` выбрасывает половину входов перед рекурсией, в то время как наш рекурсивный алгоритм `WIS` выбрасывает только одну или две вершины (возможно, из тысяч или миллионов). Оба алгоритма имеют деревья рекурсии с коэффициентом ветвления 2[1].

[1] Каждый рекурсивный алгоритм может быть связан с деревом рекурсии, узлы которого соответствуют рекурсивным вызовам алгоритма. Корень дерева соответствует

Первый имеет примерно $\log_2 n$ уровней и, следовательно, только линейное число листьев. Последний не имеет листьев вплоть до уровней $n/2$ и позже, что означает, что он имеет по крайней мере $2^{n/2}$ листьев. Мы заключаем, что время выполнения рекурсивного алгоритма растет экспоненциально вместе с n.

Решение упражнения 16.4

Правильный ответ: (б). Как изменяется входной граф при переходе к рекурсивному вызову? Одна либо две вершины и ребра отщипываются от конца графа. Таким образом, инвариант по всей рекурсии заключается в том, что каждому рекурсивному вызову дается некоторый *префикс* G_i в качестве его входного графа, где G_i обозначает первые i вершин и $i - 1$ ребер первоначального входного графа (а G_0 обозначает пустой граф):



Существует только $n + 1$ таких графов (G_0, G_1, G_2, ..., G_n), где n — это число вершин во входном графе. Поэтому только $n + 1$ разных подзадач имеют решение через экспоненциальное число разных рекурсивных вызовов.

16.3. Алгоритм реконструкции

Алгоритм `WIS` в разделе 16.2.4 вычисляет только *вес*, которым обладает независимое множество с максимальным весом (MWIS) путевого графа, а не

начальному вызову алгоритма (с исходным вводом) с одним дочерним элементом на следующем уровне для каждого из рекурсивных вызовов. Листья в нижней части дерева соответствуют рекурсивным вызовам, запускающим базовый случай, не генерируя при этом дополнительных рекурсивных вызовов.

само множество MWIS. Хитрость состоит в том, чтобы модифицировать алгоритм `WIS` так, чтобы каждый элемент массива $A[i]$ хранил u суммарный вес MWIS i-й подзадачи графа G_i, u вершины множества MWIS графа G_i, который реализует это значение.

Более совершенный подход, который экономит время и пространство, заключается в использовании шага постобработки для реконструкции множества MWIS из дорожек в грязи, оставленных алгоритмом `WIS` в его массиве подзадач A. Для начала, как узнать, принадлежит ли последняя вершина v_n входного графа G множеству MWIS? Ключом снова является лемма 16.1, в которой говорится, что два и только два кандидата соперничают за множества MWIS из G: множество MWIS графа G_{n-1} и множество MWIS графа G_{n-2}, дополненного вершиной v_n. Который из них? Тот, который имеет более крупный суммарный вес. Как узнать, какой это? Просто взгляните на подсказки, оставленные в массиве A! Конечные значения $A[n-1]$ и $A[n-2]$ хранят суммарные веса множеств MWIS соответственно графов G_{n-1} и G_{n-2}. Поэтому:

1. Если $A[n-1] \geq A[n-2] + w_n$, то множество MWIS графа G_{n-1} также является множеством MWIS графа G_n.
2. Если $A[n-2] + w_n \geq A[n-1]$, то дополнение множества MWIS графа G_{n-2} вершиной v_n дает множество MWIS графа G_n.

В первом случае мы знаем, что нужно исключить v_n из нашего решения, и можем продолжить процесс реконструкции из v_{n-1}. Во втором случае мы знаем, что нужно включить v_n в наше решение, что заставляет нас исключить v_{n-1}. Затем процесс реконструкции возобновляется с v_{n-2}.[1]

WIS_RECONSTRUCTION

Вход: массив A, вычисленный алгоритмом `WIS` для путевого графа G с множеством вершин $\{v_1, v_2, ..., v_n\}$, и неотрицательный вес w_i для каждой вершины v_i.

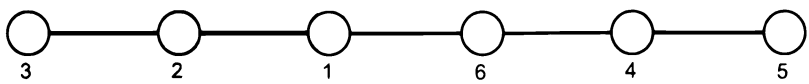
Выход: независимое множество с максимальным весом графа G.

[1] При наличии связи $A[n-2] + w_n = A[n-1]$ оба варианта приводят к оптимальному решению.

```
S := ∅                                  // вершины в множестве MWIS
i := n
while i ≥ 2 do
  if A[i − 1] ≥ A[i − 2] + w_i then     // Случай 1 побеждает
    i := i − 1                          // исключить v_i
  else                                  // Случай 2 побеждает
    S := S ∪ {v_i}                      // включить v_i
    i := i − 2                          // исключить v_{i−1}
if i = 1 then                           // базовый случай #2
  S := S ∪ {v_1}
return S
```

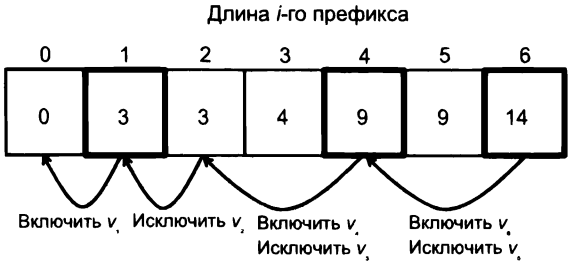
Постобработка `WIS_Reconstruction` выполняет единственный обратный проход по массиву A и тратит время $O(1)$ на итерацию цикла, поэтому он выполняется за время $O(n)$. Индукционное доказательство правильности выполняется аналогично доказательству правильности алгоритма `WIS` (теорема 16.3)[1].

Например, для входного графа



реконструкция `WIS_Reconstruction` включает v_6 (вынуждая исключить v_5) и включает v_4 (вынуждая исключить v_3), исключает v_2 и включает v_1:

[1] Можно сетовать на необходимость пересчета сравнения формы $A[i-1]$ с $A[i-2] + w_i$, ранее выполненного алгоритмом `WIS`. Если этот алгоритм модифицирован для кэширования результатов сравнения (что позволяет запомнить, какой случай повторения использовался для заполнения каждой записи массива), то потребуется получить результаты заново — уже при помощи реконструированного алгоритма `WIS`. Это особенно важно для понимания некоторых наиболее сложных задач, рассматриваемых в главах 17 и 18.



16.4. Принципы динамического программирования

16.4.1. Трехшаговый рецепт

Итак, с помощью алгоритма `WIS` мы только что разработали наш первый алгоритм динамического программирования. Общая парадигма динамического программирования может быть резюмирована трехшаговым рецептом. Пока у нас есть только один наглядный пример, поэтому вернемся к этому разделу после того, как закончим еще несколько тематических исследований.

ПАРАДИГМА ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ

1. Определить относительно малую коллекцию подзадач.
2. Показать, как быстро и правильно решать «более крупные» подзадачи с учетом решений «более мелких».
3. Показать, как быстро и правильно выводить окончательное решение из решений всех подзадач.

После выполнения этих трех шагов соответствующий алгоритм динамического программирования сформируется в такой последовательности: систематически решить все подзадачи одну за другой, работая от «наименьшей» до «наибольшей», и извлечь окончательное решение из этих подзадач. В решении задачи о взвешенном независимом множестве в n-вершинных путевых графах мы реализовали первый шаг, выявив коллекцию из $n + 1$ подзадач. Для $i = 0, 1,$

2, ..., n i-я подзадача заключается в вычислении суммарного веса множества MWIS графа G_i, состоящего из первых i вершин и $i - 1$ ребер входного графа (где G_0 обозначает пустой граф). Существует очевидный способ упорядочить подзадачи от «наименьших» до «наибольших», а именно G_0, G_1, G_2, ..., G_n. Рекурренция из следствия 16.2 является формулой, которая реализует второй шаг, показывая, как вычислить решение i-й подзадачи за время $O(1)$ из решений $(i - 2)$-й и $(i - 1)$-й подзадач. Третий шаг: вернуть решение наибольшей подзадачи, которая совпадает с первоначальной задачей.

16.4.2. Желаемые свойства подзадач

Ключом, который открывает потенциал динамического программирования для решения задачи, является выявление правильной коллекции подзадач. Какие свойства мы хотим удовлетворить? Исходя из того, что мы выполняем, по крайней мере, постоянный объем работы, решая каждую подзадачу, число подзадач является нижней границей времени выполнения нашего алгоритма. Таким образом, желательно, чтобы число подзадач было минимальным — наше решение задачи о взвешенном независимом множестве (`WIS`) использовало только линейное число подзадач, что обычно является наилучшим сценарием. Точно так же время, необходимое для решения подзадачи (с учетом решений меньших подзадач) и для выведения окончательного решения, будет учитываться в общем времени работы алгоритма. Например, предположим, что алгоритм решает не более $f(n)$ разных подзадач (работает систематически от «наименьшей» к «наибольшей»), используя не более $g(n)$ времени для каждой, и выполняет не более $h(n)$ постобработки для извлечения конечного решения (где n обозначает размер входа). Тогда время работы алгоритма составит не более

$$\underbrace{f(n)}_{\text{\# подзадачи}} \times \underbrace{g(n)}_{\substack{\text{время на подзадачу} \\ \text{(с учетом предыдущих решений)}}} + \underbrace{h(n)}_{\text{постобработка}}. \tag{16.1}$$

Три шага рецепта требуют, чтобы соответственно $f(n)$, $g(n)$ и $h(n)$ были как можно меньше. В базовом алгоритме `WIS` без этапа постобработки `WIS Reconstruction` мы имеем $f(n) = O(n)$, $g(n) = O(1)$ и $h(n) = O(1)$, в общей сложности получая время выполнения $O(n)$. Если мы включим шаг реконструкции, то член $h(n)$ подпрыгивает до $O(n)$, но совокупное время работы $O(n) \times O(1) + O(n) = O(n)$ остается линейным.

16.4.3. Повторяемый мыслительный процесс

При разработке собственных алгоритмов динамического программирования суть дела заключается в выяснении коллекции подзадач. После этого все остальное становится на свои места как бы само собой. Но как их получить? Будучи профессионалом в динамическом программировании, вы, вероятно, смогли бы интуитивно нащупать эти подзадачи. В наших тематических исследованиях мы не будем брать подзадачи с потолка, а проведем мыслительный процесс, результатом которого станет понимание коллекции подзадач (как в случае с задачей о взвешенном независимом множестве `WIS`). Этот процесс повторяем, и вы можете сымитировать его, когда применяете парадигму динамического программирования к задачам, возникающим в ваших собственных проектах. Необходимо рассуждать о структуре оптимального решения, выявляя разные способы его конструирования из оптимальных решений более мелких подзадач. В результате можно выявить и соответствующие подзадачи, и рекурренцию (аналогично следствию 16.2), которая выражает решение подзадачи как функцию решений меньших подзадач. Тогда алгоритм динамического программирования может заполнить массив решениями подзадач, продвигаясь от меньших к бóльшим подзадачам и используя рекурренцию для вычисления каждого элемента массива.

16.4.4. Динамическое программирование против «разделяй и властвуй»

Читатели, знакомые с парадигмой проектирования алгоритмов «разделяй и властвуй» (раздел 13.1.1), могут заметить некоторые сходства с динамическим программированием, в особенности с его рекурсивной формулировкой «сверху вниз» (разделы 16.2.2, 16.2.3). Обе парадигмы рекурсивно решают малые подзадачи и комбинируют результаты в решение первоначальной задачи[1]. Назовем шесть различий между типичным использованием этих двух парадигм:

---

[1] Так, в алгоритме `MergeSort` каждый рекурсивный вызов делит входной массив на левую и правую части. Алгоритм быстрой сортировки вызывает подпрограмму разделения для выбора способа разделения входного массива, после чего фиксирует это деление в процессе выполнения.

1. Каждый рекурсивный вызов типичного алгоритма «разделяй и властвуй» придерживается единственного способа разделения входа на более мелкие подзадачи. Каждый рекурсивный вызов алгоритма динамического программирования оставляет свои варианты открытыми, рассматривая несколько способов выявления меньших подзадач и выбора наилучшей из них[1].

2. Поскольку каждый рекурсивный вызов алгоритма динамического программирования пробует несколько вариантов меньших подзадач, подзадачи обычно повторяются в различных рекурсивных вызовах; кэширование решений подзадач тогда становится «оптимизацией на дважды два». В большинстве алгоритмов «разделяй и властвуй» все подзадачи различны и нет смысла кэшировать их решения[2].

3. Большинство канонических приложений парадигмы «разделяй и властвуй» заменяют простой полиномиально-временной алгоритм задачи более быстрой версией «разделяй и властвуй»[3]. Высокоэффективные приложения динамического программирования являются полиномиально-временными алгоритмами оптимизационных задач, для которых простые решения (например, исчерпывающий поиск) требуют экспоненциального времени.

4. В алгоритме «разделяй и властвуй» подзадачи выбираются в первую очередь для оптимизации времени выполнения; правильность часто заботится о себе сама. В динамическом программировании подзадачи обычно выбираются с учетом правильности, каким бы ни было время выполнения[4].

---

[1] К примеру, в алгоритме `WIS` каждый рекурсивный вызов выбирает между подзадачей с одним наименьшим количеством вершин и подзадачей с двумя наименьшими значениями количества вершин.

[2] В частности, в алгоритмах `MergeSort` и `QuickSort` каждая подзадача соответствует отдельному подмассиву входного массива.

[3] Так, алгоритм `MergeSort` переносит время выполнения сортировки массива длиной n с прямой границы $O(n^2)$ до $O(n \log n)$. Можно упомянуть также алгоритм Карацубы (увеличивающий время выполнения умножения двух n-значных чисел с $O(n^2)$ на $O(n^{1.59})$) и алгоритм Штрассена (для умножения двух матриц $n \times n$ в $O(n^{2.81})$ скорее, чем $O(n^3)$ время).

[4] Например, алгоритм быстрой сортировки всегда правильно сортирует входной массив (независимо от того, насколько хороши или плохи выбранные им элементы сводки). Хорошим примером следует считать и динамический алгоритм программирования (применительно к решению задачи из раздела 16.5).

5. Алгоритм «разделяй и властвуй», как правило, рекурсивно работает на подзадачах с размером не более постоянной доли (например, 50 %) от входа. Динамическое программирование лояльно к рекурсивной работе на подзадачах, которые едва меньше входа (как в алгоритме `WIS`), если это необходимо для правильности.

6. Парадигму «разделяй и властвуй» можно рассматривать как частный случай динамического программирования, в котором каждый рекурсивный вызов выбирает фиксированную коллекцию подзадач для рекурсивного решения. Как более сложная парадигма, динамическое программирование применяется к более широкому кругу задач, чем «разделяй и властвуй», но оно также более технически требовательно в применении (по крайней мере, до тех пор пока у вас не будет достаточного практического опыта).

Какую парадигму следует предпочесть, столкнувшись с новой задачей? Если вы видите решение «разделяй и властвуй», непременно используйте его. Если все ваши попытки «разделяй и властвуй» оказываются безуспешными — и в особенности если они оказываются безуспешными, потому что шаг совмещения всегда требует повторного выполнения большого объема вычислений с нуля, — то пришло время попробовать динамическое программирование.

16.4.5. Почему «динамическое программирование»?

В происхождении термина «динамическое программирование» много неясных моментов. Отец динамического программирования Ричард Э. Беллман описывал свою работу в корпорации RAND так:

> 1950-е годы были не лучшими для математических исследований. В Вашингтоне в то время был очень интересный джентльмен по имени Уилсон — министр обороны, испытывавший патологический страх и ненависть к слову «исследования». Лицо Уилсона наливалось кровью, если люди использовали термин «исследование» в его присутствии. Можете себе представить, как он относился к термину «математический». Корпорация RAND работала на ВВС, и, по сути, Уилсон был ее боссом. Я чувствовал, что должен сделать что-то, чтобы скрыть от Уилсона и ВВС, что в действительности занимаюсь математикой в корпорации RAND. В первую очередь меня интересовало

планирование, принятие решений, мышление. Но планирование — это не вполне подходящее в данном случае слово. Поэтому я решил использовать слово «программирование»... Слово [«динамический»] обладает очень интересным свойством как прилагательное, а именно слово «динамический» невозможно употребить в уничижительном смысле. Попробуйте придумать какое-нибудь сочетание, которое, возможно, придаст ему уничижительный смысл. Это невозможно. Таким образом, я подумал, что динамическое программирование было бы хорошим названием. Это было то, против чего не мог возражать даже конгрессмен. Поэтому я и использовал его в качестве зонтика для своей деятельности[1].

16.5. Задача о ранце

Наш второй пример касается хорошо известной *задачи о ранце*. Следуя подходу, примененному для разработки алгоритма `WIS` в разделе 16.2, приходим к известному решению задачи динамического программирования.

16.5.1. Определение задачи

Экземпляр задачи о ранце задается $2n + 1$ целыми положительными числами, где n — это число «предметов» (которые помечены произвольно от 1 до n): значение v_i и размер s_i каждого предмета i и емкость ранца C[2]. Ответственность алгоритма заключается в выборе подмножества предметов. Общая стоимость предметов должна быть как можно больше, а их общий объем — не более C.

ЗАДАЧА: РАНЕЦ

Вход: значения $v_1, v_2, ..., v_n$ предметов, размеры $s_1, s_2, ..., s_n$ предметов и емкость ранца C. (Все положительные целые числа.)

---

[1] *Ричард Э. Беллман.* «Эпицентр урагана: автобиография», World Scientic, 1984, с. 159.

[2] На самом деле неважно, что значения элементов являются целыми числами (в отличие от произвольных положительных действительных чисел). Важно, как мы вскоре убедимся, чтобы размеры элементов были целыми числами.

Выход: подмножество $S \subseteq \{1, 2, ..., n\}$ предметов с максимально возможной суммой $\Sigma_{i \in S} v_i$ значений, при условии что суммарный размер $\Sigma_{i \in S} s_i$ не превышает C.

УПРАЖНЕНИЕ 16.5

Рассмотрим пример задачи о ранце с емкостью ранца $C = 6$ и четырьмя элементами:

Предмет	Значение	Размер
1	3	4
2	2	3
3	4	2
4	4	3

Каково суммарное значение оптимального решения?

а) 6

б) 7

в) 8

г) 10

(Решение и пояснение см. в разделе 16.5.7.)

Поиск ответов на вопросы вроде «На какие товары и услуги вы должны потратить свою зарплату, чтобы получить максимальную отдачу?» или «С учетом оперативного бюджета и множества кандидатов на работу с различной производительностью и требуемой заработной платой, кого вы должны нанять?» подразумевает решение все той же задачи о ранце.

16.5.2. Оптимальная подструктура и рекурренция

Чтобы применить парадигму динамического программирования к задаче о ранце, необходимо выявить правильную коллекцию подзадач. Как и в слу-

чае с задачей о взвешенном независимом множестве (WIS), следует обратиться к структуре оптимальных решений и поиску различных способов их построения из оптимальных решений более мелких подзадач. Еще одним практическим результатом этого упражнения будет рекурренция для быстрого вычисления решения подзадачи из двух меньших подзадач.

Рассмотрим пример задачи о ранце со значениями предметов $v_1, v_2, ..., v_n$, размерами предметов $s_1, s_2, ..., s_n$ и емкостью ранца C и предположим, что имеется оптимальное решение $S \subseteq \{1, 2, ..., n\}$ с суммарным значением $V = \Sigma_{i \in S} v_i$. Как оно должно выглядеть? Как и в случае с задачей о взвешенном независимом множестве, мы начинаем с дилеммы: S либо содержит последний предмет (предмет n), либо нет[1].

Случай 1: $n \notin S$. Поскольку оптимальное решение S исключает последний предмет, его можно рассматривать как допустимое решение (по-прежнему с суммарным значением V и суммарным размером не более C) меньшей задачи, состоящей только из первых $n - 1$ предметов (и вместимости ранца C). Более того, S должно быть оптимальным решением меньшей подзадачи: решение $S^* \subseteq \{1, 2, ..., n - 1\}$ с суммарным размером не более C и суммарным значением больше V аналогично первоначальному варианту решения, что противоречит предполагаемой оптимальности S.

Случай 2: $n \in S$. Сложнее, когда оптимальное решение задействует последний предмет n. Этот случай может произойти только при $s_n \leq C$. Мы не можем рассматривать S как допустимое решение меньшей задачи только с первыми $n - 1$ предметами (такая возможность возникает после удаления элемента n). Является ли $S - \{n\}$ оптимальным решением меньшей подзадачи?

---

[1] Задача ИСВ на графах путей, по сути, является последовательной, причем вершины упорядочены вдоль пути. Это по понятным причинам привело к подзадачам, соответствующим префиксам ввода. Элементы в задаче о ранце не упорядочены по своей сути, но для определения правильного набора подзадач полезно придерживаться подхода, описанного ранее, делая вид, что они упорядочены произвольным образом. «Префикс» элементов соответствует первым i элементам в произвольном порядке (для некоторого $i \in \{0, 1, 2, ..., n\}$). Заметим, что и многие другие алгоритмы динамического программирования используют этот же прием.

УПРАЖНЕНИЕ 16.6

Какое из следующих ниже утверждений соблюдается для множества $S - \{n\}$? Выберите все, что применимо.

а) Это решение является оптимальным для подзадачи, состоящей из первых $n - 1$ предметов и ранца вместимостью C;

б) Это решение является оптимальным для подзадачи, состоящей из первых $n - 1$ предметов и ранца вместимостью $C - v_n$;

в) Это решение является оптимальным для подзадачи, состоящей из первых $n - 1$ предметов и ранца вместимостью $C - s_n$;

г) Решение, возможно, не является допустимым, если вместимость ранца составляет только $C - s_n$.

(Решение и пояснение см. в разделе 16.5.7.)

Этот анализ случая показывает, что два и только два кандидата соперничают за то, чтобы быть оптимальным решением задачи о ранце.

Лемма 16.4 (оптимальная подструктура ранца). *Пусть S равно оптимальному решению задачи о ранце с $n \geq 1$ предметами, значениями предметов $v_1, v_2, \ldots, v_n$, размерами предметов $s_1, s_2, \ldots, s_n$ и емкостью ранца C. Тогда S равно либо*:

(i) оптимальному решению для первых $n - 1$ предметов с вместимостью ранца C, либо

(ii) оптимальному решению для первых $n - 1$ предметов с вместимостью ранца $C - s_n$, дополненному последним предметом n.

Решение в (i) всегда является вариантом оптимального решения. Решение в (ii) является вариантом тогда и только тогда, когда $s_n \leq C$; в этом случае единицы мощности s_n фактически резервируются заранее для предмета n[1].

[1] Это аналогично ситуации, рассмотренной в рамках решения задачи ИСВ на графах путей с исключением предпоследней вершины графа (чтобы зарезервировать пространство для конечной вершины).

Вариант с бóльшим суммарным значением является оптимальным решением, приводящим к следующему рекуррентному соотношению.

Следствие 16.5 (рекуррентное соотношение ранца). *С допущениями и обозначениями леммы 16.4 пусть $V_{i,c}$ обозначает максимальное суммарное значение подмножества первых i предметов с суммарным размером не более c. При $i = 0$ интерпретируйте $V_{i,c}$ как 0. Для каждого $i = 1, 2, ..., n$ и $c = 0, 1, 2, ..., C$*

$$V_{i,c} = \begin{cases} \underbrace{V_{i-1,\,c}}_{\text{Случай 1}} & \text{если } s_i > c \\ \max\{\underbrace{V_{i-1,\,c}}_{\text{Случай 1}}, \underbrace{V_{i-1,\,c-s_i} + v_i}_{\text{Случай 2}}\} & \text{если } s_i \le c. \end{cases}$$

Поскольку и C, и размеры предметов являются целыми числами, остаточная емкость $c - s_i$ во втором случае также является целым числом.

16.5.3. Подзадачи

Следующим шагом является определение коллекции релевантных подзадач и их систематическое решение с использованием рекуррентного соотношения, указанного в следствии 16.5. Сейчас мы сосредоточимся на вычислении суммарного значения оптимального решения для каждой подзадачи. Как и в задаче о взвешенном независимом множестве на путевых графах, из этой информации мы сможем реконструировать предметы в оптимальном решении первоначальной задачи. В задаче о взвешенном независимом множестве на путевых графах использован только один параметр i для индексирования подзадач, где i — длина префикса входного графа. Для задачи о ранце из леммы 16.4 и следствия 16.5 видно, что подзадачи должны быть параметризованы двумя индексами: длиной i префикса имеющихся предметов и имеющейся емкостью C ранца[1]. Варьируя в диапазоне всех соответствующих значений двух параметров, сформулируем подзадачи.

---

[1] В задаче `WIS` для графов путей соответствующая подзадача может рассматриваться в качестве второстепенной при наличии меньшего количества вершин. В задаче с ранцем таких подзадач две (для случаев с меньшим числом предметов либо с меньшим свободным объемом ранца).

РАНЕЦ: ПОДЗАДАЧИ

Вычислить $V_{i,c}$, суммарное значение оптимального решения задачи о ранце с первыми i элементами и вместимостью c ранца.

(Для каждого $i = 0, 1, 2, ..., n$ и $c = 0, 1, 2, ...C$.)

Самая большая подзадача (при $i = n$ и $c = C$) в точности совпадает с первоначальной задачей. Поскольку все размеры предметов и емкость ранца C являются положительными целыми числами и поскольку емкость всегда уменьшается на размер некоторого предмета (резервируя для него место), единственными остаточными емкостями, которые могут когда-либо возникнуть, являются целые числа от 0 до C[1].

16.5.4. Алгоритм динамического программирования

С учетом подзадачи и рекуррентного соотношения составим алгоритм динамического программирования для задачи о ранце.

KNAPSACK

Вход: значения предметов $v_1, v_2, ..., v_n$, размеры предметов $s_1, s_2, ..., s_n$ и емкость ранца C (все положительные целые числа).

Выход: максимальное суммарное значение подмножества $S \subseteq \{1, 2, ..., n\}$, где $\Sigma_{i \in S} s_i \leq C$.

```
// решения подзадач (проиндексированы от 0)
A := (n + 1) × (C + 1) двумерный массив
// базовый случай (i = 0)
for c = 0 to C do
    A[0][c] = 0
```

[1] Каждый рекурсивный вызов удаляет последний элемент и целое число единиц емкости. Возникающие вследствие этого подзадачи включают в себя некоторый префикс элементов и некоторую целочисленную остаточную емкость.

```
// систематически решить все подзадачи
for i = 1 to n do
    for c = 0 to C do
        // использовать рекуррентное соотношение
        // из следствия 16.5
        if s_i > c then
            A[i][c] := A[i − 1][c]
        else
            A[i][c] :=
```

$$\max\{\underbrace{A[i-1][c]}_{\text{Случай 1}},\ \underbrace{A[i-1][c-s_i]+v_i}_{\text{Случай 2}}\}$$

```
return A[n][C] // решение наибольшей подзадачи
```

Массив A теперь является двумерным, отражая два индекса, i и c, используемые для параметризации подзадач. К тому времени, когда итерация сдвоенного цикла `for` должна вычислить решение подзадачи $A[i][c]$, значения $A[i-1][c]$ и $A[i-1][c-s_i]$ двух релевантных меньших подзадач уже были вычислены на предыдущей итерации внешнего цикла (или в базовом случае). Следовательно, алгоритм тратит время $O(1)$ на решение каждой из подзадач $(n+1)(C+1) = O(nC)$ с совокупным временем выполнения $O(nC)$.[1,2]

Наконец, как и в случае с алгоритмом `WIS`, правильность алгоритма `Knapsack` следует по индукции по количеству предметов, используя рекуррентное соотношение из следствия 16.5 для обоснования индукционного шага.

Теорема 16.6 (свойства ранца). *Для каждого экземпляра задачи о ранце алгоритм* `Knapsack` *возвращает суммарное значение оптимального решения и выполняется за время $O(nC)$, где n — это число предметов, а C — емкость ранца.*

---

[1] В выражении (16.1) $f(n) = O(nC)$, $g(n) = O(1)$ и $h(n) = O(1)$.

[2] Показатель времени работы $O(nC)$ особенно впечатляет при малых значениях C, например, если $C = O(n)$ или (в идеале) еще меньше. В следующей книге (*часть 4* серии) мы отыщем причину, указывающую на известную сложность задачи с ранцем.

16.5.5. Пример

Вспомните пример с четырьмя предметами из упражнения 16.5, где $C = 6$:

Предмет	Значение	Размер
1	3	4
2	2	3
3	4	2
4	4	3

Поскольку $n = 4$ и $C = 6$, массив A в алгоритме `Knapsack` можно представить в виде таблицы с пятью столбцами (соответствующими $i = 0, 1, ..., 4$) и семью строками (соответствующими $c = 0, 1, ..., 6$). Конечными значениями массива будут:

Остаточная емкость c	0	1	2	3	4
6	0	3	3	7	8
5	0	3	3	6	8
4	0	3	3	4	4
3	0	0	2	4	4
2	0	0	0	4	4
1	0	0	0	0	0
0	0	0	0	0	0
	Префиксная длина i				

Алгоритм `Knapsack` вычисляет эти элементы столбец за столбцом (работает слева направо) и внутри столбца снизу вверх. Чтобы заполнить элемент

i-го столбца, алгоритм сравнивает элемент сразу слева (соответствующий случаю 1) с v_i плюс элемент один столбец слева и s_i строк вниз (случай 2). Например, для $A[2][5]$ наиболее оптимально пропустить второй предмет и наследовать «3» сразу слева, тогда как для $A[3][5]$ наиболее оптимально включение третьего предмета и достижение 4 (для v_3) плюс 2 в элементе $A[2][3]$.

16.5.6. Реконструкция

Алгоритм `Knapsack` вычисляет только суммарное значение оптимального решения, а не само оптимальное решение. Как и в случае с алгоритмом `WIS`, возможно реконструировать оптимальное решение, проследив заполненный массив A. Начиная с самой большой подзадачи в правом верхнем углу, алгоритм реконструкции проверяет, какой случай рекуррентного отношения был использован для вычисления $A[n][C]$. Если это случай 1, то алгоритм опускает предмет n и возобновляет реконструкцию с элемента $A[n-1][C]$. Если это случай 2, то алгоритм включает предмет n в свое решение и возобновляет реконструкцию с элемента $A[n-1][C-s_n]$.

KNAPSACK_RECONSTRUCTION

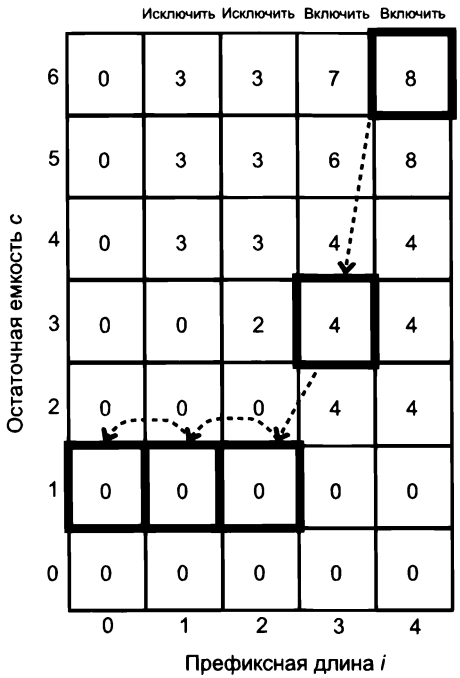
Вход: массив A, вычисленный алгоритмом `Knapsack`, со значениями предметов $v_1, v_2, \ldots, v_n$, размерами предметов $s_1, s_2, \ldots, s_n$ и емкостью ранца C.

Выход: оптимальное решение задачи о ранце.

```
S := ∅                          // предметы в оптимальном решении
c := C                                  // остаточная емкость
for i = n downto 1 do
    if s_i ≤ c and A[i − 1][c − s_i] + v_i ≥ A[i − 1][c] then
        S := S ∪ {i}        // случай 2 побеждает, включить i
        c := c − s_i        // резервировать для него место
    // в противном случае пропустить i, емкость остается
    // прежней
return S
```

Шаг постобработки `Knapsack_Reconstruction` выполняется за время $O(n)$ (с $O(1)$-й работой в расчете на итерацию главного цикла), что намного быстрее, чем время $O(nC)$, используемое для заполнения массива в алгоритме `Knapsack`[1].

Например, обратная трассировка массива из примера на с. 168 дает оптимальное решение {3, 4}:



16.5.7. Решения упражнений 16.5–16.6

Решение упражнения 16.5

Правильный ответ: (в). Так как емкость ранца равна 6, нет места для выбора более чем двух предметов. Самая ценная пара предметов — это третий

[1] В выражении (16.1) последующая обработка с помощью алгоритма реконструкции ранца увеличивает значение $h(n)$ до $O(n)$. Общее время работы $O(nC) \times O(1) + O(n) = O(nC)$ при этом остается неизменным.

и четвертый (с суммарным значением 8), причем они помещаются в ранце (с суммарным размером 5).

Решение упражнения 16.6

Правильный ответ: (в). Наиболее очевидно ложным является утверждение (б), которое даже не проверяет тип (C в единицах размера, v_n в единицах значения). Например, v_n может быть больше C, в каковом случае $C - v_n$ является отрицательным и бессмысленным. В случае (г), так как S является допустимым для первоначальной задачи, его суммарный размер составляет не более C; после того как n удалено из S, суммарный размер падает до не более $C - s_n$ и, следовательно, $S - \{n\}$ является допустимым для пониженной емкости. Ответ (а) является естественной догадкой, но также не является правильным[1].

В случае (в) мы фактически резервируем s_n единиц емкости для включения предмета n, что оставляет остаточную емкость $C - s_n$. $S - \{n\}$ является допустимым решением меньшей подзадачи (с емкостью ранца $C - s_n$) с суммарным значением $V - v_n$. Если бы имелось оптимальное решение $S^* \subseteq \{1, 2, ..., n - 1\}$, с суммарным значением $V^* > V - v_n$ и суммарным размером не более $C - s_n$, то $S^* \cup \{n\}$ будет иметь суммарный размер не более C и суммарное значение $V^* + v_n > (V - v_n) + v_n = V$. Это противоречило бы предполагаемой оптимальности S для первоначальной задачи.

ВЫВОДЫ

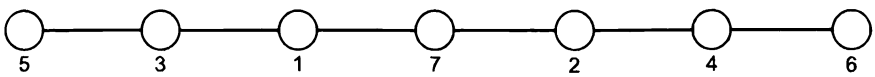
- Динамическое программирование следует трехшаговому рецепту: (i) определить относительно малую коллекцию подзадач; (ii) показать, как быстро и правильно решать «более крупные» подзадачи с учетом решений «более мелких»; (iii) показать, как быстро и правильно выводить окончательное решение из решений всех подзадач.

[1] Предположим, к примеру, что $C = 2$. Рассмотрим два элемента с $v_1 = s_1 = 1$ и $v_2 = s_2 = 2$. Оптимальным решением S является $\{2\}$. $S - \{2\}$ — пустое множество, но единственное оптимальное решение подзадачи, состоящей из первого предмета и емкости ранца 2, — это $\{1\}$.

- Алгоритм динамического программирования, который решает не более $f(n)$ разных подзадач, используя не более $g(n)$ времени для каждой, и выполняет не более $h(n)$ постобработки по извлечению окончательного решения, выполняется за время $O(f(n) \times g(n) + h(n))$, где n обозначает размер входа.
- Правильная коллекция подзадач и рекуррентное соотношение для систематического их решения могут быть выявлены рассуждением о структуре оптимального решения и различных способах, которыми оно могло бы быть сконструировано из оптимальных решений меньших подзадач.
- Типичные алгоритмы динамического программирования заполняют массив значениями решений подзадач, а затем выполняют обратную трассировку заполненного массива, реконструируя само решение.
- Независимое множество неориентированного графа является подмножеством взаимно несмежных вершин.
- В n-вершинных путевых графах независимое множество с максимальным весом может быть вычислено с использованием динамического программирования за время $O(n)$.
- В задаче о ранце с учетом n предметов со значениями и размерами и емкостью ранца C (все из них положительные целые числа) цель состоит в том, чтобы выбрать подмножество предметов с максимальным значением, имеющих суммарный размер не более C.
- Задача о ранце может быть решена с помощью динамического программирования за время $O(nC)$.

Задачи на закрепление материала

Задача 16.1. Рассмотрим входной граф



В нем вершины помечены их весами. Каковы конечные элементы массива в алгоритме `WIS` из раздела 16.2 и какие вершины принадлежат множеству MWIS?

Задача 16.2. Какое из следующих ниже утверждений соблюдается? Выберите все, что применимо.

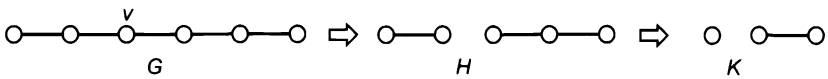
а) Алгоритмы `WIS` и `WIS_Reconstruction` разделов 16.2 и 16.3 всегда возвращают решение, включающее вершину с максимальным весом;

б) Когда веса вершин являются разными, алгоритмы `WIS` и `WIS_Reconstruction` никогда не возвращают решение, которое включает вершину с минимальным весом;

в) Если вершина v не принадлежит множеству MWIS префикса G_i, содержащего первые i вершин и $i - 1$ ребер входного графа, то она не принадлежит любому множеству MWIS G_{i+1}, G_{i+2}, ..., G_n;

г) Если вершина v не принадлежит множеству MWIS графа G_{i-1} либо G_i, то она не принадлежит любому множеству MWIS G_{i+1}, G_{i+2}, ..., G_n.

Задача 16.3. Эта задача обрисовывает подход к решению задачи о взвешенном независимом множестве в графах, более сложных, чем путевые. Рассмотрим произвольный неориентированный граф $G = (V, E)$ с неотрицательными весами вершин и произвольную вершину $v \in V$ с весом w_v. Возьмем H из G, удалив v и ее инцидентные ребра из G. Возьмем K из H, удалив соседей вершины v и их инцидентные ребра:



Пусть W_G, W_H и W_K обозначают суммарный вес соответственно множества MWIS в G, H и K. Рассмотрим формулу:

$$W_G = \max\{W_H, W_K + w_v\}.$$

Какое из следующих ниже утверждений является истинным? Выберите все, что применимо.

а) Формула не всегда является правильной в путевых графах;

б) Формула всегда является правильной в путевых графах, но не всегда правильной в деревьях;

в) Формула всегда является правильной в деревьях, но не всегда правильной в произвольных графах;

г) Формула всегда является правильной в произвольных графах;

д) Формула приводит к линейно-временному алгоритму для задачи о взвешенном независимом множестве в деревьях;

е) Формула приводит к линейно-временному алгоритму для задачи о взвешенном независимом множестве в произвольных графах.

Задача 16.4. Рассмотрим пример задачи о ранце с пятью предметами:

Предмет	Значение	Размер
1	1	1
2	2	3
3	3	2
4	4	5
5	5	4

и ранец емкостью $C = 9$. Какими являются окончательные элементы массива алгоритма `Knapsack` из раздела 16.5? Какие предметы принадлежат оптимальному решению?

Задачи повышенной сложности

Задача 16.5. Эта задача описывает четыре обобщения задачи о ранце. В каждом из них входные данные состоят из значений предметов v_1, v_2, ..., v_n, размеров предметов s_1, s_2, ..., s_n и дополнительных, специфичных для задачи данных (все — положительные целые числа). Какое из этих обобщений может быть решено динамическим программированием во времени, полиномом по числу n предметов и наибольшему числу M, которое появляется на входе? Выберите все, что применимо.

а) С учетом положительной целочисленной емкости C вычислить подмножество предметов с максимально возможным суммарным значением, при условии что суммарный размер *точно* равен C. Если такого множества не существует, то алгоритм должен правильно обнаружить этот факт;

б) С учетом положительной целочисленной емкости C и бюджета предметов $k \in \{1, 2, ..., n\}$ вычислить подмножество предметов с максимально возможным суммарным значением при условии наличия суммарного размера не более C и *не более k предметов*;

в) С учетом емкостей C_1 и C_2 двух ранцев вычислить непересекающиеся подмножества S_1, S_2 предметов с максимально возможным суммарным значением $\sum_{i \in S_1} v_i + \sum_{i \in S_2} v_i$ в зависимости от емкостей ранца: $\sum_{i \in S_1} s_i \leq C_1$ и $\sum_{i \in S_2} s_i \leq C_2$;

г) С учетом емкостей C_1, C_2, ..., C_m m ранцев, где m может быть таким же, как n, вычислить непересекающиеся подмножества S_1, S_2, ..., S_m предметов с максимально возможным суммарным значением $\sum_{i \in S_1} v_i + \sum_{i \in S_2} v_i + \cdots + \sum_{i \in S_m} v_i$ в зависимости от емкостей рюкзака: $\sum_{i \in S_1} s_i \leq C_1$, $\sum_{i \in S_2} s_i \leq C_2$, ... и $\sum_{i \in S_m} s_i \leq C_m$.

Задачи по программированию

Задача 16.6. Реализуйте на своем любимом языке программирования алгоритмы `WIS` и `WIS_Reconstruction`. Тестовые случаи и наборы данных для сложных задач см. на веб-сайте www.algorithmsilluminated.org.

Задача 16.7. Реализуйте на любимом языке программирования алгоритмы `Knapsack` и `Knapsack_Reconstruction`. Тестовые случаи и наборы данных для сложных задач см. на веб-сайте www.algorithmsilluminated.org.

17 Расширенное динамическое программирование

Эта глава продолжает курс молодого бойца по динамическому программированию еще двумя примерами: задачей о выравнивании последовательностей (раздел 17.1) и задачей вычисления бинарного дерева поиска с минимально возможным средним временем поиска (раздел 17.2). В обоих случаях структура оптимальных решений сложнее, чем в примерах из предыдущей главы, причем решение подзадачи зависит от решения более чем двух меньших подзадач. Закончив эту главу, задайтесь вопросом: могли бы вы решить любую из этих задач, не изучив предварительно динамическое программирование?

17.1. Выравнивание последовательностей

17.1.1. Актуальность

Если вы проходите курс по вычислительной геномике, то первые несколько лекций, скорее всего, будут посвящены задаче о *выравнивании последовательностей*[1]. В этой задаче вход состоит из двух символьных последовательностей, представляющих части одного или нескольких геномов, над алфавитом $\{A, C, G, T\}$. Символьные последовательности не обязательно должны иметь одинаковую длину. Например, входами могут быть символьные последовательности *AGGGCT* и *AGGCA*. Формально задача заключается в том, чтобы определить, насколько похожи две символьные последовательности; мы уточним это в следующем разделе.

Потребность в решении этой задачи обусловлена, по меньшей мере, двумя причинами. Во-первых, предположим, что вы пытаетесь выяснить функцию области сложного генома, такого как геном человека. Один подход состоит в том, чтобы отыскать подобную область в более хорошо понимаемом геноме, таком как геном мыши, и предположить, что эти области играют те же самые или схожие роли. Совершенно другое применение задачи состоит в том, чтобы делать выводы о филогенетическом дереве — какие виды от каких видов произошли и когда. Например, вам может быть интересно, произошел ли вид B от вида A, а затем вид C от B, или же B и C эволюционировали независимо от A. Сходство генома можно использовать в качестве индикатора близости в филогенетическом дереве.

[1] Идеи рисунков в примерах этого раздела заимствованы из раздела 6.6 книги «Разработка алгоритмов» Джона Кляйнберга и Эвы Тардос (Pearson, 2005).

17.1.2. Определение задачи

Наши примеры символьных последовательностей *AGGGCT* и *AGGCA*, очевидно, не идентичны, но они все же интуитивно ощущаются скорее похожими, чем нет. Как формализовать эту интуицию? Одна идея состоит в том, чтобы отметить, что эти две символьные последовательности могут быть «безупречно выровнены»:

$$\begin{array}{cccccc} A & G & G & G & C & T \\ A & G & G & - & C & A \end{array}$$

где черта «–» указывает на зазор, вставленный между двумя буквами второй цепочки символов, в которой, похоже, отсутствует «*G*». Две символьные последовательности совпадают в четырех из шести столбцах; единственными недостатками в выравнивании являются зазор и несовпадение между *A* и *T* в последнем столбце. В общем случае выравнивание — это способ вставки зазоров в одну или обе входные символьные последовательности, с тем чтобы они имели одинаковую длину:

$$\begin{array}{l} \text{———} X + \textit{зазоры} \text{———} \\ \underbrace{\text{———} Y + \textit{зазоры} \text{———}}_{\text{общая длина } \ell} \end{array}$$

Затем мы можем определить сходство двух символьных цепочек в соответствии с качеством их самого безупречного выравнивания. Но что делает выравнивание «безупречным»? Что лучше иметь: один зазор и одно несовпадение или же три зазора и никаких несовпадений?

Предположим, что на такие вопросы уже даны ответы в виде известных *штрафов* за зазоры и несовпадения, предусмотренных в составе входа (наряду с двумя символьными последовательностями). Они конкретизируют штрафы, налагаемые выравниванием в каждом из его столбцов; суммарный штраф выравнивания является суммой штрафов его столбцов. Например, приведенное выше выравнивание *AGGGCT* и *AGGCA* понесет штраф α_{gap} (предоставленная стоимость зазора), а также штраф α_{AT} (представленная стоимость несовпадения между *A* и *T*). Задача о выравнивании последовательностей состоит в том, чтобы вычислить выравнивание, которое минимизирует суммарный штраф.

ЗАДАЧА: ВЫРАВНИВАНИЕ ПОСЛЕДОВАТЕЛЬНОСТЕЙ

Вход: две символьные цепочки X, Y над алфавитом $\Sigma = \{A, C, G, T\}$, штраф α_{xy} для каждой пары символов $x, y \in \Sigma$ и неотрицательный штраф за зазор $\alpha_{gap} \geq 0$[1].

Выход: выравнивание X и Y с минимально возможным суммарным штрафом.

Один из способов интерпретировать выравнивание с минимальным штрафом — это «наиболее правдоподобно объяснить» то, как одна из цепочек могла эволюционировать в другую. Мы можем думать о зазоре как об отмене удаления, которое произошло когда-то в прошлом, а о несовпадении — как об отмене мутации. Минимальный штраф выравнивания двух цепочек известен как отметка *Нидлмана—Вунша*, или *NW-отметка* символьных цепочек[2]. Две цепочки считаются «похожими» тогда и только тогда, когда их NW-отметка относительно мала.

УПРАЖНЕНИЕ 17.1

Предположим, что существует штраф 1 за каждый зазор и штраф 2 за сопоставление двух разных символов в столбце. Какова NW-отметка цепочек *AGTACG* и *ACATAG*?

а) 3

б) 4

в) 5

г) 6

(Решение и пояснение см. в разделе 17.1.8.)

[1] Логика подсказывает, что все штрафы неотрицательны с $\alpha_{xx} = 0$ для всех $x \in \Sigma$ и $\alpha_{xy} = \alpha_{yx}$ для всех $x, y \in \Sigma$. Тем не менее следует помнить, что алгоритм динамического программирования требует, чтобы штраф за зазор был неотрицательным.

[2] Назван в честь его изобретателей, Саула Б. Нидлмана и Кристиана Д. Вунша [Saul B. Needleman & Christian D. Wunsch]. Опубликован в статье «Общий метод, применимый для поиска сходств в аминокислотной последовательности двух белков» (Journal of Molecular Biology, 1970).

NW-отметка для специалистов в области геномики была бы бесполезной без эффективной процедуры ее вычисления. Число выравниваний двух символьных цепочек растет экспоненциально вместе с их объединенной длиной, поэтому за пределами неинтересно малых экземпляров исчерпывающий поиск не будет завершен за наши жизни. Динамическое программирование сэкономит день; повторяя тот же тип мысленного эксперимента, который мы использовали для задачи о взвешенном независимом множестве на путевых графах и задаче о ранце, мы придем к эффективному алгоритму вычисления NW-отметки[1].

17.1.3. Оптимальная подструктура

Вместо того чтобы чрезмерно пугаться того, насколько фундаментальной является задача о выравнивании последовательностей, давайте следовать нашему обычному рецепту динамического программирования и посмотрим, что произойдет. Если у вас уже черный пояс по динамическому программированию, то вы сможете угадать правильную коллекцию подзадач.

Рассмотрим в качестве примера выравнивание двух цепочек с минимальным штрафом. Как оно должно выглядеть? Сколькими разными способами можно было бы его построить из оптимальных решений более мелких подзадач? В задаче о взвешенном независимом множестве на путевых графах и задаче о ранце мы сосредоточивались на последнем решении — принадлежит ли последняя вершина пути или последний предмет экземпляра ранца окончательному решению? Продолжая шаблон, казалось бы, мы должны сосредоточиться на последнем столбце выравнивания:

$$\underbrace{\begin{matrix} A & G & G & G & C \\ A & G & G & - & C \end{matrix}}_{\text{остальная часть выравнивания}} \quad \underbrace{\begin{matrix} T \\ A \end{matrix}}_{\text{последний столбец}}$$

В наших первых двух тематических исследованиях конечная вершина или предмет либо входили, либо выходили из решения, что представляет собой

[1] Алгоритмы предопределили развитие вычислительной геномики в качестве самостоятельной области научных знаний. Не будь эффективного алгоритма для вычисления показателя NW, Нидлман и Вунш наверняка предложили бы иное определение сходства генома!

две разные возможности. Сколько же релевантных возможностей существует в задаче о выравнивании последовательностей для содержимого последнего столбца?

УПРАЖНЕНИЕ 17.2

Пусть $X = x_1, x_2, ..., x_m$ и $Y = y_1, y_2, ..., y_n$ равны двум входным символьным цепочкам, где каждый символ x_i или y_j принадлежит $\{A, C, G, T\}$. Сколько релевантных возможностей существует для содержимого конечного столбца оптимального выравнивания?

а) 2

б) 3

в) 4

г) mn

(Решение и пояснение см. в разделе 17.1.8.)

Последующий анализ тематических случаев позволяет выяснить, что для оптимального выравнивания есть только три кандидата — по одному кандидату для каждого из возможного содержимого последнего столбца. Это приведет к рекуррентному соотношению, вычисляемому путем исчерпывающего поиска по трем возможностям, и алгоритму динамического программирования, который использует это рекуррентное соотношение для систематического решения всех релевантных подзадач. Рассмотрим оптимальное выравнивание двух непустых символьных цепочек $X = x_1, x_2, ..., x_m$ и $Y = y_1, y_2, ..., y_n$. Пусть $X' = x_1, x_2, ..., x_{m-1}$ и $Y' = y_1, y_2, ..., y_{n-1}$ обозначают соответственно X и Y с убранным последним символом.

Случай 1: x_m и y_n совпадают в последнем столбце выравнивания. Предположим, что оптимальное выравнивание не использует зазор в конечном столбце, предпочитая сочетать конечные символы x_m и y_n входных цепочек. Пусть P обозначает суммарный штраф, понесенный этим выравниванием. Мы можем рассматривать остальную часть выравнивания (за исключением последнего столбца) как выравнивание оставшихся символов — выравнивание более коротких цепочек X' и Y':

$$\underbrace{\text{——— } X' + \textit{зазоры} \text{ ———}}_{\text{остальная часть выравнивания}} \begin{matrix} x_m \\ y_n \end{matrix}$$

Это выравнивание X' и Y' имеет суммарный штраф P минус штраф $\alpha_{x_m y_n}$, который был оплачен ранее в последнем столбце. И это не просто любое старое выравнивание X' и Y' — это оптимальное выравнивание. Если какое-то другое выравнивание X' и Y' имело меньший суммарный штраф $P^* < P - \alpha_{x_m y_n}$, дополнение к нему конечного столбца, сочетающего x_m и y_n, произвело бы выравнивание X и Y с суммарным штрафом $P^* + \alpha_{x_m y_n} < \left(P - \alpha_{x_m y_n}\right) + \alpha_{x_m y_n} = P$, что противоречит оптимальности первоначального выравнивания X и Y.

Другими словами, если оптимальное выравнивание X и Y сочетает x_m и y_n в последнем столбце, то несложно понять, как выглядит остальная часть: оптимальное выравнивание X' и Y'.

Случай 2: x_m совпал с зазором в последнем столбце выравнивания. Поскольку y_n не появляется в последнем столбце, индуцированное выравнивание относится к X' и первоначальной второй цепочке Y:

$$\underbrace{\text{——— } X' + \textit{зазоры} \text{ ———}}_{\text{остальная часть выравнивания}} \begin{matrix} x_m \\ \text{———} Y + \textit{зазоры} \text{———} \end{matrix} [\text{зазор}].$$

Более того, индуцированное выравнивание является оптимальным выравниванием X' и Y; аргумент аналогичен аргументу в случае 1 (в чем вы должны убедиться сами).

Случай 3: y_n совпал с зазором в последнем столбце выравнивания. В этом случае индуцированное выравнивание относится к X и Y':

$$\underbrace{\text{——— } X + \textit{зазоры} \text{ ———}}_{\text{остальная часть выравнивания}} \begin{matrix} [\textit{зазор}] \\ \text{———} Y' + \textit{зазоры} \text{———} \end{matrix} y_n.$$

Более того, как и в первых двух случаях, это выравнивание является оптимальным выравниванием данного типа (что тоже следует проверить). Смысл анализа — в том, чтобы сузить возможности оптимального решения до не более чем трех кандидатов.

Лемма 17.1 (оптимальная подструктура выравнивания последовательностей). *Оптимальное выравнивание двух непустых символьных цепочек* $X = x_1, x_2, ..., x_m$ *и* $Y = y_1, y_2, ..., y_n$ *равно либо*:

(i) оптимальному выравниванию X' *и* Y', *дополненному сочетанием* x_m *и* y_n *в последнем столбце*;

(ii) оптимальному выравниванию X' *и* Y, *дополненному сочетанием* x_m *и зазора в конечном столбце*;

(iii) оптимальному выравниванию X *и* Y', *дополненному сочетанием зазора и* y_n *в конечном столбце*,

где X' *и* Y' *обозначают соответственно* X *и* Y *с удаленными конечными символами* x_m *и* y_n.

УПРАЖНЕНИЕ 17.3

Предположим теперь, что одна из двух входных цепочек (скажем, Y) является пустой. Какова NW-отметка X и Y?

а) 0

б) $\alpha_{gap} \times$ (длина X)

в) $+\infty$

г) не определено

(Решение и пояснение см. в разделе 17.1.8.)

17.1.4. Рекуррентное соотношение

Упражнение 17.3 обрабатывает базовый случай пустой входной цепочки. Для непустых входных цепочек из трех вариантов в лемме 17.1 оптимальным решением является вариант с наименьшим суммарным штрафом. Эти наблюдения приводят к следующему рекуррентному соотношению, которое вычисляет лучший из трех вариантов исчерпывающим поиском:

Следствие 17.2 (рекуррентное соотношение выравнивания последовательностей). *С допущениями и обозначениями леммы 17.1 пусть $P_{i,j}$ обозначает суммарный штраф оптимального выравнивания $X_i = x_1, x_2, ..., x_i$, первые i символов X, и Y_j $y_1, y_2, ..., y_j$, первые j символов Y. (Если $j = 0$ либо $i = 0$, то интерпретировать $P_{i,j}$ соответственно как $i \times \alpha_{gap}$ либо $j \times \alpha_{gap}$.) Тогда*

$$P_{m,n} = \min\left\{\underbrace{P_{m-1,n-1} + \alpha_{x_m y_n}}_{\text{Случай 1}}, \underbrace{P_{m-1,n} + \alpha_{gap}}_{\text{Случай 2}}, \underbrace{P_{m,n-1} + \alpha_{gap}}_{\text{Случай 3}}\right\}.$$

В более общем случае для каждого $i = 1, 2, ..., m$ и $j = 1, 2, ..., n$

$$P_{i,g} = \min\left\{\underbrace{P_{i-1,j-1} + \alpha_{x_i y_j}}_{\text{Случай 1}}, \underbrace{P_{i-1,j} + \alpha_{gap}}_{\text{Случай 2}}, \underbrace{P_{i,j-1} + \alpha_{gap}}_{\text{Случай 3}}\right\}.$$

Более общее утверждение в следствии 17.2 формулируется путем вызова первого утверждения для каждого $i = 1, 2, ..., m$ и $j = 1, 2, ..., n$, где X_i и Y_j играют роль входных цепочек X и Y.

17.1.5. Подзадачи

Как и в задаче о ранце, подзадачи в рекуррентном соотношении (следствие 17.2) индексируются двумя разными параметрами i и j. Подзадачи ранца могут сжиматься в двух разных смыслах (удаляя предмет или удаляя емкость ранца), и то же происходит с подзадачами выравнивания последовательностей (удаляя символ из первой или второй входной цепочки). Варьируя в диапазоне релевантных значений двух параметров, мы получаем следующую коллекцию подзадач[1]:

ВЫРАВНИВАНИЕ ПОСЛЕДОВАТЕЛЬНОСТЕЙ: ПОДЗАДАЧИ

Вычислить $P_{i,j}$, минимальный суммарный штраф выравнивания первых i символов X и первых j символов Y.

(Для каждого $i = 0, 1, 2, ..., m$ и $j = 0, 1, 2, ..., n$.)

[1] Каждый рекурсивный вызов извлекает либо последний символ из первой входной строки, либо последний символ из второй входной строки или оба названных символа. Возникающие вследствие этого подзадачи относятся к префиксам исходных входных строк.

Самая большая подзадача (с $i = m$ и $j = n$) аналогична первоначальной задаче.

17.1.6. Алгоритм динамического программирования

Итак, подзадачи определены. Имеется рекуррентное соотношение для решения подзадачи с учетом решений меньших подзадач. Ничто не может помешать нам использовать его для систематического решения всех подзадач, начиная с базовых случаев и заканчивая первоначальной задачей.

NW

Вход: цепочки $X = x_1, x_2, ..., x_m$ и $Y = y_1, y_2, ..., y_n$ над алфавитом $\Sigma = \{A, C, G, T\}$, штраф α_{xy} для каждой $x, y \in \Sigma$ и штраф за зазор $\alpha_{gap} \geq 0$.

Выход: NW-отметка X и Y.

```
// решения подзадач (индексируемых с 0)
```
A := $(m + 1) \times (n + 1)$ двумерный массив
```
// базовый случай #1 (j = 0)
```
for $i = 0$ to m **do**

    $A[i][0] = i \times \alpha_{gap}$

```
// базовый случай #2 (i = 0)
```
for $j = 0$ to n **do**

    $A[0][j] = j \times \alpha_{gap}$

```
// систематическое решение всех подзадач
```
for $i = 1$ to m **do**

    for $j = 1$ to n **do**

        // использовать рекуррентное соотношение
        // из следствия 17.2
        $A[i][j]$:=

$$\min\left\{\begin{array}{ll} A[i-1][j-1]+\alpha_{x_i y_j} & (\text{Случай 1}) \\ A[i-1][j]+\alpha_{gap} & (\text{Случай 2}) \\ A[i][j-1]+\alpha_{gap} & (\text{Случай 3}) \end{array}\right\}$$

return $A[m][n]$ // решение самой крупной подзадачи

Как и в задаче о ранце, поскольку подзадачи индексируются двумя разными параметрами, алгоритм использует двумерный массив для хранения решений подзадач и сдвоенный цикл `for` для его заполнения. К тому времени, когда итерация цикла должна вычислить решение подзадачи $A[i][j]$, значения $A[i-1][j-1]$, $A[i-1][j]$ и $A[i][j-1]$ трех релевантных меньших подзадач уже были вычислены и готовы в ожидании, что их отыщут за постоянное время. Это означает, что алгоритм тратит время $O(1)$ на решение каждой из $(m+1)(n+1) = O(mn)$ подзадач с суммарным временем выполнения $O(mn)$[1]. Как и наши предыдущие алгоритмы динамического программирования, правильность алгоритма `NW` может быть доказана по индукции; индукция выполняется на значении $i+j$ (размер подзадачи), с рекуррентным соотношением в следствии 17.2, оправдывающем индукционный шаг.

Теорема 17.3 (свойства алгоритма `NW`). *Для каждого экземпляра задачи о выравнивании последовательностей алгоритм `NW` возвращает правильную NW-отметку и выполняется за время $O(mn)$, где m и n — это длины двух входных символьных цепочек.*

Пример алгоритма `NW` в действии см. в задаче 17.1[2].

17.1.7. Реконструкция

В реконструкции оптимального выравнивания из значений массива, которое вычисляет алгоритм `NW`, нет никаких сюрпризов. Работая в обратном направлении, алгоритм сначала проверяет, какой случай рекурренции использовался для заполнения элемента массива $A[m][n]$, соответствующего самой большой подзадаче (произвольно разрывая совпадения значений)[3]. Если это был слу-

---

[1] В выражении (16.1) $f(n) = O(mn)$, $g(n) = O(1)$ и $h(n) = O(1)$.

[2] Существует ли лучшее решение? В особых случаях — да (см. задачу 17.6), но не в случае с задачами общего плана. В этой связи имеет смысл перечитать статью «Редактирование расстояния не может быть вычислено за сильно субквадратичное время (в предположении, что SETH не ложно)» Артурса Бакурса и Петра Индика (SIAM Journal of Computing, 2018).

[3] В зависимости от деталей реализации эта информация могла быть кэширована алгоритмом NW, что позволяло ее найти (см. также сноску 4 на с. 154). В противном случае алгоритм восстановления может пересчитать ответ с нуля за $O(1)$ раз.

чай 1, то последний столбец оптимального выравнивания сочетает x_m и y_n, и реконструкция возобновляется из элемента $A[m-1][n-1]$. Если это был случай 2 или 3, то последний столбец выравнивания соответствует либо x_m (в случае 2), либо y_n (в случае 3) с зазором, и процесс возобновляется с позиции $A[m-1][n]$ (в случае 2) либо $A[m][n-1]$ (в случае 3). Когда алгоритм реконструкции попадает в базовый случай, он завершает выравнивание, добавляя соответствующее число зазоров в цепочку, в которой закончились символы. Поскольку алгоритм выполняет $O(1)$-ю работу на итерацию и каждая итерация уменьшает сумму длин оставшихся префиксов, его время выполнения равно $O(m+n)$. Подробный псевдокод мы оставляем заинтересованному читателю.

17.1.8. Решение упражнений 17.1–17.3

Решение упражнения 17.1

Правильный ответ: (б). Вот одно выравнивание с двумя зазорами и одним несовпадением при суммарном штрафе 4:

$$\begin{array}{ccccccc} A & - & G & T & A & C & G \\ A & C & A & T & A & - & G \end{array}$$

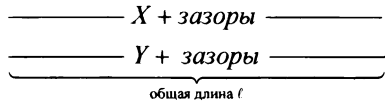
Вот выравнивание с четырьмя зазорами и без несовпадений, а также с суммарным штрафом 4:

$$\begin{array}{ccccccc} A & - & - & G & T & A & C & G \\ A & C & A & - & T & A & - & G \end{array}$$

Ни одно выравнивание не имеет суммарного штрафа 3 или меньше. Поскольку входные цепочки имеют одинаковую длину, каждое выравнивание вставляет одинаковое число зазоров в каждую, то есть общее число зазоров является четным. Выравнивание с четырьмя или более зазорами имеет суммарный штраф не менее 4. Выравнивание с нулевыми зазорами имеет четыре несовпадения и суммарный штраф 8. Каждое выравнивание, вставляющее только один зазор в каждую цепочку, приводит, по крайней мере, к одному несовпадению при суммарном штрафе не менее 4.

Решение упражнения 17.2

Правильный ответ: (б). Рассмотрим оптимальное выравнивание цепочек X и Y:



Что может находиться в правом верхнем углу, в последнем столбце первой строки? Это может быть или зазор, или символ. Если это символ, то он должен быть из X (потому что это верхняя строка), и это должен быть последний символ x_m (потому что он находится в последнем столбце). Точно так же нижний правый угол должен быть либо зазором, либо последним символом y_n второй строки Y.

С двумя вариантами для каждого из двух элементов в последнем столбце существует четыре возможных сценария. Но один из них не имеет значения! Бессмысленно иметь два зазора в одном столбце — имеется неотрицательный штраф в расчете на зазор, а их удаление производит новое, «улучшающее» выравнивание. Это оставляет нам три релевантных возможности для содержимого последнего столбца оптимального выравнивания: (i) x_m и y_n сочетаются, (ii) x_m сочетается с зазором либо (iii) y_n сочетается с зазором.

Решение упражнения 17.3

Правильный ответ: (б). Если Y является пустой цепочкой, то оптимальное выравнивание вставляет достаточное число зазоров в Y так, чтобы она имела ту же длину, что и X. Это влечет за собой штраф α_{gap} в расчете на зазор, при суммарном штрафе α_{gap}, умноженном на длину X.

*17.2. Оптимальные бинарные деревья поиска

В главе 11 *части 2* мы изучили бинарные деревья поиска, поддерживающие полное упорядочение на эволюционирующем множестве объектов,

а также богатый набор быстрых операций. В главе 14 мы определили беспрефиксные коды и разработали жадный алгоритм вычисления наилучшего в среднем кода для заданного множества частот символов. Далее рассматривается аналогичная задача для деревьев поиска: вычисление наилучшего в среднем дерева поиска с учетом статистики о частотах разных поисков — более сложная, чем задача оптимального беспрефиксного кода, но вполне решаемая в рамках парадигмы динамического программирования.

17.2.1. Обзор бинарного дерева поиска

Бинарное дерево поиска — это структура данных, которая действует как динамическая версия отсортированного массива: поиск объекта так же прост, как и в отсортированном массиве, но он также позволяет осуществлять быструю вставку и удаление. Указанная структура данных хранит объекты, ассоциированные с ключами (и, возможно, много других данных), с одним объектом для каждого узла дерева[1]. Каждый узел имеет левый и правый дочерние указатели, любой из которых может быть нулевым (null). Левое поддерево узла x определяется как узлы, достижимые из x через его левый дочерний указатель (аналогично для правого поддерева). Определяющим свойством *дерева поиска* является:

СВОЙСТВО ДЕРЕВА ПОИСКА

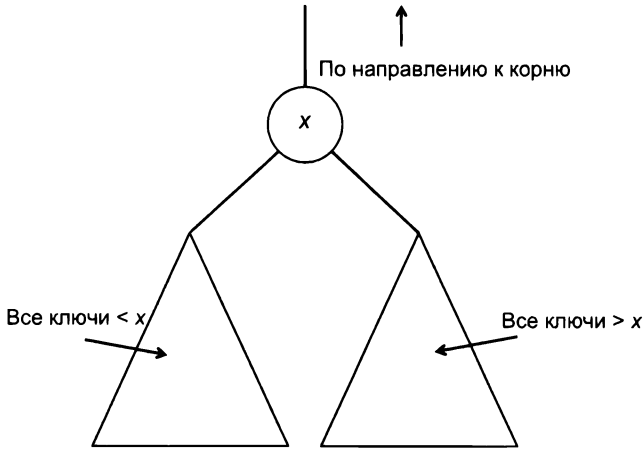
Для каждого объекта x:

1. Объекты в левом поддереве x имеют ключи меньше, чем в x.
2. Объекты в правом поддереве x имеют ключи больше, чем в x.

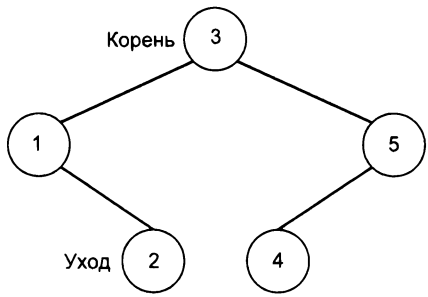
В этом разделе для простоты мы будем считать, что никакие два объекта не имеют одного и того же ключа.

[1] Мы ссылаемся на узлы дерева и соответствующие объекты взаимозаменяемо.

Свойство дерева поиска накладывает требование на каждый узел дерева поиска, а не только на корень:



Например, вот дерево поиска, содержащее объекты с ключами {1,2, 3, 4, 5}:

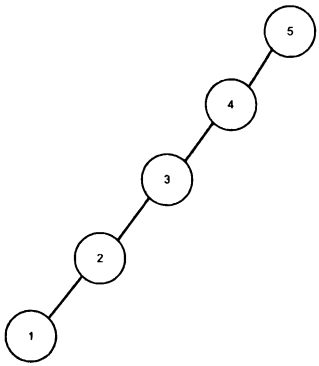


Смысл свойства дерева поиска в том, чтобы свести поиск объекта с заданным ключом к следованию по прямой, что напоминает двоичный поиск в отсортированном массиве. Например, вы ищете объект с ключом 17. Если объект, находящийся в корне дерева, имеет ключ 23, то искомый объект находится в левом поддереве корня. Если ключ корня равен 12, то нужно выполнить рекурсивный поиск объекта в правом поддереве.

17.2.2. Среднее время поиска

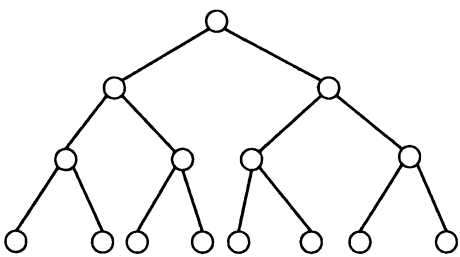
Время поиска ключа k в двоичном дереве поиска T равно числу узлов, посещенных при поиске узла с ключом k (включая сам этот узел). В приведенном выше дереве ключ «3» имеет время поиска 1, ключи «1» и «5» имеют время поиска 2, а ключи «2» и «4» имеют время поиска 3[1].

Разные деревья поиска для одного и того же множества объектов приводят к разному времени поиска. Например, вот второе дерево поиска, содержащее объекты с ключами {1, 2, 3, 4, 5}:



где «1» теперь имеет время поиска 5.

Какое же дерево поиска является «лучшим»? Мы задали этот вопрос в главе 11 *части 2*. Ответом было идеально сбалансированное дерево:

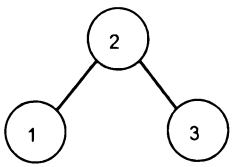


[1] По аналогии, время поиска равно единице плюс глубина соответствующего узла в дереве.

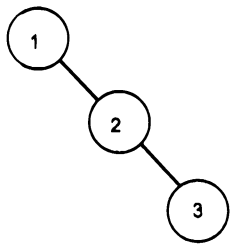
Обоснование? Идеально сбалансированное дерево минимизирует длину самого длинного пути от корня к листу ($\approx \log_2 n$ для n объектов) или, что эквивалентно, максимальное время поиска. Сбалансированные структуры данных бинарного дерева поиска, такие как красно-черные деревья, специально разработаны для того, чтобы поддерживать дерево поиска близко к идеально сбалансированному (см. раздел 11.4 *части 2*). Минимизация времени поиска имеет смысл, если у вас нет данных о надежности вариантов поиска. Но что делать, если у вас есть соответствующая статистика[1]?

УПРАЖНЕНИЕ 17.4

Рассмотрим следующие два дерева поиска, в которых хранятся объекты с ключами 1, 2 и 3:



и



и частоты поисков

Ключ	Частота поиска
1	0,8
2	0,1
3	0,1

[1] Допустим, мы реализовали проверку орфографии в виде бинарного дерева поиска, в котором хранятся все правильно написанные слова. Проверка орфографии документа сводится к поиску каждого из его слов по очереди, в сочетании в том числе и с неудачными поисками, соответствующими словам с ошибками. Оценить частоту различных поисков можно путем подсчета количества вхождений различных слов (как правильно, так и неправильно написанных) в достаточно большом наборе документов.

Каково среднее время поиска в двух деревьях соответственно?

а) 1,9 и 1,2

б) 1,9 и 1,3

в) 2 и 1

г) 2 и 3

(Решение и пояснение см. в разделе 17.2.9.)

17.2.3. Определение задачи

Упражнение 17.4 показывает, что лучшее бинарное дерево поиска для работы зависит от частот поиска, при этом несбалансированные деревья потенциально превосходят сбалансированные деревья, когда частоты поиска неравномерны. Это наблюдение важно для разработки алгоритма: какое бинарное дерево поиска будет лучшим с учетом частоты поиска для множества ключей?

ЗАДАЧА: ОПТИМАЛЬНЫЕ БИНАРНЫЕ ДЕРЕВЬЯ ПОИСКА

Вход: отсортированный список ключей k_1, k_2, ..., k_n и неотрицательная частота p_i для каждого ключа k_j.

Выход: бинарное дерево поиска T, содержащее ключи (k_1, k_2, ..., k_n} с минимально возможным взвешенным временем поиска:

$$\sum_{i=1}^{n} p_i \times \underbrace{(\text{время поиска для } k_i \text{ в } T)}_{=(\text{глубина } k_i \text{ в } T)+1}. \qquad (17.1)$$

Три замечания. Во-первых, имена ключей не имеют значения и поэтому могут быть обозначены как (1, 2, ..., n}. Во-вторых, формулировка задачи не предполагает, что p_i в сумме составляет 1 (отсюда формулировка «взвешенное» время поиска вместо «среднего» времени поиска). При необходимости вы можете свободно нормализовать частоты, разделив каждую из них на их

сумму $\sum_{j=1}^{n} p_j$ — это не изменит задачу. В-третьих, задача, как указано, не связана с безуспешными поисками, то есть поисками ключа, отличного от того, что находится в заданном множестве $\{k_1, k_2, ..., k_n\}$. Следует проверить, что решение на основе динамического программирования распространяется на случай, когда также подсчитываются времена безуспешных поисков, при условии, что вход указывает частоты таких поисков.

Задача оптимального бинарного дерева поиска имеет некоторое сходство с задачей оптимального беспрефиксного кода (глава 14). В обеих задачах вход задает множество частот над символами или ключами, выходом является бинарное дерево, а целевая функция связана с минимизацией средней глубины. Разница заключается в ограничении, которому должно удовлетворять бинарное дерево. В задаче оптимального беспрефиксного кода единственным ограничением является то, что символы появляются только на листьях. Решение задачи оптимального бинарного дерева поиска должно удовлетворять более сложному свойству дерева поиска (с. 88). Вот почему жадные алгоритмы недостаточно хороши для последней задачи, в связи с чем придется прибегнуть к парадигме динамического программирования.

17.2.4. Оптимальная подструктура

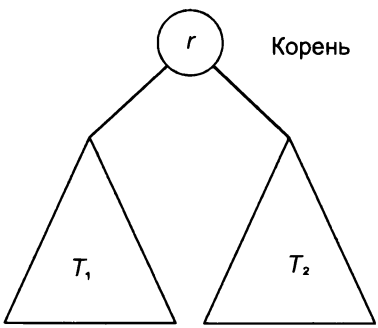
Прежде всего важно понимание способов построения оптимального решения из оптимальных решений более мелких подзадач. Предположим, мы приняли (обреченный на неудачу) подход «разделяй и властвуй» к задаче оптимального бинарного дерева поиска. Каждый рекурсивный вызов алгоритма «разделяй и властвуй» берет единственное разбиение задачи на одну или несколько более мелких подзадач. Какое из них мы должны использовать? Сразу напрашивается установка объекта с медианным ключом в корне с последующим рекурсивным вычислением левого и правого поддеревьев. Однако при неоднородных частотах поиска нет оснований ожидать, что медиана будет хорошим выбором для корня (см. упражнение 17.4). Выбор корня имеет непредсказуемые последствия. Как заранее правильно разделить задачу на две меньшие подзадачи?

Если бы мы только знали корень. В задаче о взвешенном независимом множестве на путевых графах (раздел 16.2.1), зная о принадлежности последней

вершины оптимальному решению, можно без труда установить, как выглядит остальная часть. В задаче о ранце (раздел 16.5.2), зная о принадлежности последнего предмета оптимальному решению, можно установить, как выглядит остальная часть. В задаче о выравнивании последовательностей (раздел 17.1.3), зная содержимое последнего столбца оптимального выравнивания, также можно установить, как выглядит остальная часть. Мы изучили все возможности применительно к задачам о взвешенном независимом множестве и о ранце, как и о задаче о выравнивании последовательностей. Возможно, и решение задачи об оптимальном бинарном дереве поиска должно основываться на *опробовании всех возможных корней*. В следующем упражнении предстоит выяснить, какой тип леммы оптимальной подструктуры может быть верным для задачи об оптимальном бинарном дереве поиска.

УПРАЖНЕНИЕ 17.5

Предположим, что оптимальное бинарное дерево поиска для ключей $\{1, 2, ..., n\}$ и частот $p_1, p_2, ..., p_n$ имеет ключ r в своем корне, с левым поддеревом T_1 и правым поддеревом T_2:



Из следующих ниже четырех утверждений выберите то, которое, как вы подозреваете, является истинным.

а) Ни T_1, ни T_2 не должны быть оптимальными для ключей, которые оно содержит;

б) По крайней мере один из T_1, T_2 является оптимальным для ключей, которые оно содержит;

в) Каждый из T_1, T_2 является оптимальным для ключей, которые оно содержит;

г) T_1 является оптимальным бинарным деревом поиска для ключей $\{1, 2, ..., r-1\}$ и аналогичным образом T_2 — для ключей $\{r+1, r+2, ..., n\}$.

(Решение и пояснение см. в разделе 17.2.9.)

Как обычно, формализация оптимальной подструктуры сводится к анализу случаев, с одним случаем для каждой возможности того, как может выглядеть оптимальное решение. Рассмотрим оптимальное бинарное дерево поиска T с ключами $\{1, 2, ..., n\}$ и частотами $p_1, p_2, ..., p_n$. Любой из n ключей может появиться в корне оптимального решения, поэтому существует n разных случаев. Мы можем рассуждать обо всех сразу.

Случай r: корень из T имеет ключ r. Пусть T_1 и T_2 обозначают левое и правое поддеревья корня. Из свойства дерева поиска вытекает, что резидентами T_1 являются ключи $\{1, 2, ..., r-1\}$ и аналогичным образом для T_2 и $\{r+1, r+2, ..., n\}$. Более того, T_1 и T_2 являются допустимыми деревьями поиска для своих множеств ключей (то есть как T_1, так и T_2 удовлетворяют свойству дерева поиска). Далее мы покажем, что оба являются оптимальными бинарными деревьями поиска для их соответствующих подзадач с частотами $p_1, p_2, ..., p_{r-1}$ и $p_{r+1}, p_{r+2}, ..., p_n$, унаследованными от первоначальной задачи[1].

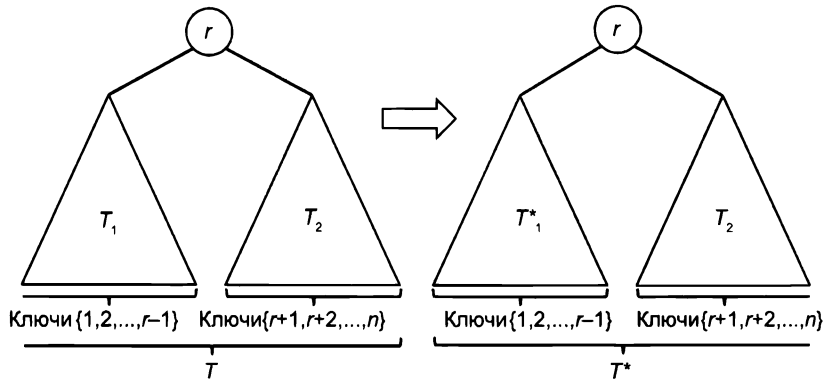
Предположим, от противного, что хотя бы одно из поддеревьев — скажем, T_1 — не является оптимальным решением соответствующей ему подзадачи. Это означает, что существует другое дерево поиска T_1^* с ключами $\{1, 2, ..., r-1\}$ и со строго меньшим взвешенным временем поиска:

$$\sum_{k=1}^{r-1} p_k \times \left(k \text{ время поиска в } T_1^*\right) < \sum_{k=1}^{r-1} p_k \times \left(k \text{ время поиска в } T_1\right). \qquad (17.2)$$

Далее, используя неравенство (17.2), покажем дерево поиска первоначальной задачи, превосходящее T, что противоречит предполагаемой оптимальности T. Получим дерево T^*, выполнив операцию на T, вырезав его левое поддерево T_1 и вставив T_1^* на его место:

---

[1] При $r = 1$ или $r = n$ одно из двух поддеревьев является пустым, причем пустое дерево вполне подходит для пустого набора ключей.



Последний шаг — сравнение взвешенных времен поиска в T^* и T. Разбив сумму в (17.1) на две части, ключи $\{1, 2, ..., r-1\}$ и ключи $\{r, r+1, ..., n\}$, мы можем записать эти времена поиска соответственно как

$$\sum_{k=1}^{r-1} p_k \times \underbrace{\left(k \text{ время поиска в } T^*\right)}_{=1+\left(\text{время поиска в } T_1^*\right)} < \sum_{k=r}^{n} p_k \times \left(k \text{ время поиска в } T^*\right)$$

и

$$\sum_{k=1}^{r-1} p_k \times \underbrace{\left(k \text{ время поиска в } T\right)}_{=1+\left(\text{время поиска в } T_1\right)} < \sum_{k=r}^{n} p_k \times \underbrace{\left(k \text{ время поиска в } T\right)}_{\text{так же как в } T^* (\text{как } k \geq r)}.$$

Поскольку деревья T^* и T имеют один и тот же корень r и одно и то же правое поддерево T_2, время поиска ключей $r, r+1, ..., n$ одинаково для обоих деревьев. Поиск ключа в $\{1, 2, ..., r-1\}$ сначала посещает корень r, а затем рекурсивно производит поиск в левом поддереве. Таким образом, время поиска такого ключа на единицу больше в T^*, чем в T_1^*, и на единицу больше в T, чем в T_1. Это означает, что взвешенные времена поиска в T^* и T могут быть записаны соответственно как

$$\underbrace{\sum_{k=1}^{r-1} p_k \times \left(k \text{ время поиска в } T_1^*\right)}_{\text{левая сторона (17.2)}} + \sum_{k=1}^{r-1} p_k + \sum_{k=r}^{n} p_k \times \left(k \text{ время поиска в } T^*\right)$$

и

$$\underbrace{\sum_{k=1}^{r-1} p_k \times \left(k \text{ время поиска в } T_1\right)}_{\text{правая сторона (17.2)}} + \sum_{k=1}^{r-1} p_k + \sum_{k=r}^{n} p_k \times \left(k \text{ время поиска в } T^*\right).$$

Второй и третий члены одинаковы в обоих выражениях. Следуя допущению (17.2), первый член меньше в первом выражении, чем во втором, что означает, что взвешенное время поиска в T^* меньше, чем в T. Это приводит к ожидаемому противоречию и завершает доказательство ключевого утверждения о том, что T_1 и T_2 являются оптимальными бинарными деревьями поиска для их соответствующих подзадач.

Лемма 17.4 (оптимальная подструктура оптимального бинарного дерева поиска). *Если T является оптимальным бинарным деревом поиска с ключами $\{1, 2, \ldots, n\}$, частотами $p_1, p_2, \ldots, p_n$, корнем r, левым поддеревом T_1 и правым поддеревом T_2, то*:

(а) T_1 является оптимальным бинарным деревом поиска для ключей $\{1, 2, \ldots, r-1\}$ и частот $p_1, p_2, \ldots, p_{r-1}$ и

(б) T_2 является оптимальным бинарным деревом для ключей $\{r+1, r+2, \ldots, n\}$ и частот $p_{r+1}, p_{r+2}, \ldots, p_n$.

Другими словами, знание корня оптимального бинарного дерева поиска позволяет установить, как выглядят его левое и правое поддеревья.

17.2.5. Рекуррентные соотношения

Лемма 17.4 сужает возможности для оптимального бинарного дерева поиска до n и только n кандидатов, где n — число ключей во входе (то есть число вариантов корня). Лучший из этих n кандидатов должен быть оптимальным решением.

Следствие 17.5 (рекуррентное соотношение оптимального бинарного дерева поиска). *С допущениями и обозначениями леммы 17.4 пусть $W_{i,j}$ обозначает взвешенное время поиска в оптимальном бинарном дереве поиска с ключами $\{i, i+1, \ldots, j\}$ и частотами $p_i, p_i+1, \ldots, p_j$. Если $i > j$, то интерпретировать $W_{i,j}$ как 0. Тогда*

$$W_{1,n} = \sum_{k=1}^{n} p_k + \min_{r \in \{1,2,\ldots,n\}} \underbrace{\{W_{1,r-1} + W_{r+1,n}\}}_{\text{Случай } r}. \qquad (17.3)$$

В более общем случае для каждого $i, j \in \{1, 2, ..., n\}$, где $i \leq j$,

$$W_{i,j} = \sum_{k=i}^{j} p_k + \min_{r \in \{i, i+1, ..., j\}} \underbrace{\{W_{i,r-1} + W_{r+1,j}\}}_{\text{Случай } r}.$$

Более общее утверждение в следствии 17.5 формулируется путем вызова первого утверждения для каждого $i, j \in \{1, 2, ..., n\}$, где $i \leq j$, с ключами $\{i, i + 1, ..., j\}$ и их частотами, играющими роль первоначального входа.

В рекуррентном соотношении (17.3) «min» реализует исчерпывающий поиск по n разным кандидатам оптимального решения. Член $\sum_{k=1}^{n} p_k$ необходим, потому что установка оптимальных поддеревьев под новым корнем добавляет 1 ко всем временам поиска их ключей[1]. Обратите внимание, что этот дополнительный член необходим для того, чтобы рекуррентность была правильной, даже если есть только один ключ (и взвешенное время поиска является частотой этого ключа).

17.2.6. Подзадачи

В задаче о ранце (раздел 16.5.3) подзадачи индексировались двумя параметрами, поскольку «размер» подзадачи был двумерным (при этом один параметр отслеживал префикс предметов, а другой — оставшуюся емкость ранца). Схожим образом в задаче о выравнивании последовательностей (раздел 17.1.5) был один параметр, отслеживающий префикс каждой из двух входных цепочек. Рассматривая рекуррентное соотношение в следствии 17.5, мы видим, что подзадачи снова индексируются двумя параметрами (i и j), но на этот раз по разным причинам. Подзадачи формы $W_{1,r-1}$ в рекуррент-

[1] Стоит подробнее рассмотреть дерево T с корнем r и левым и правым поддеревьями T_1 и T_2. Время поиска ключей $\{1, 2, ..., r - 1\}$ в T на один больше, чем в T_1, а время поиска для ключей $\{r + 1, r + 2, ..., n\}$ в T на один больше, чем в T_2. Таким образом, взвешенное время поиска (17.1) может быть обозначено как $\sum_{k=1}^{r-1} p_k \times$ (время поиска $1 + k$ в T_1) $+ p_r \times 1 + \sum_{k=r+1}^{n} p_k \times$ (время поиска $1 + k$ в T_2), с последующим упрощением до $\sum_{k=1}^{n} p_k$ + (взвешенное время поиска в T_1) + (взвешенное время поиска в T_2).

ном соотношении (17.3) определяются префиксами множества ключей (как обычно), а подзадачи формы $W_{r+1,n}$ определяются *суффиксами* ключей. Чтобы подготовиться к обоим случаям, будем отслеживать как первый (самый малый), так и последний (самый большой) ключи, принадлежащие подзадаче[1]. Таким образом, мы получаем двумерное множество подзадач, несмотря на кажущийся одномерным вход. Варьируя в диапазоне релевантных значений двух параметров, получаем искомую коллекцию подзадач.

ОПТИМАЛЬНОЕ БИНАРНОЕ ДЕРЕВО ПОИСКА: ПОДЗАДАЧИ

Вычислить $W_{i,j}$, минимальное взвешенное время поиска в бинарном дереве поиска с ключами $\{i, i+1, ..., j\}$ и частотами p_i, p_{i+1}, ..., p_j.

(Для каждого $i, j \in \{1, 2, ..., n\}$, где $i \leq j$.)

Самая большая подзадача (с $i = 1$ и $j = n$) в точности совпадает с первоначальной задачей.

17.2.7. Алгоритм динамического программирования

Располагая подзадачами и рекуррентным соотношением, следует ожидать, что алгоритм динамического программирования напишется сам собой, если правильно определить последовательность решения подзадач. Самый простой способ определения «размера подзадачи» — по числу ключей на входе.

[1] Каждый рекурсивный вызов отбрасывает один или несколько наименьших либо, напротив, наибольших ключей. Возникающие при этом подзадачи соответствуют непрерывным подмножествам исходного набора ключей — множествам вида $\{i, i+1, ..., j\}$ для некоторого $i, j \in \{1, 2, ..., n\}$ с $i \leq j$. Скажем, для рекурсивного вызова может быть задан префикс исходного ввода $\{1, 2, ..., 100\}$, например $\{1, 2, ..., 22\}$, но при этом некоторые из его собственных рекурсивных вызовов будут включены в суффиксы его ввода — к примеру, $\{18, 19, 20, 21, 22\}$.

Поэтому имеет смысл сначала решить все подзадачи с входом из одного ключа, затем подзадачи с входами из двух ключей и т. д. В следующем ниже псевдокоде переменная s управляет текущим размером подзадачи[1].

OptBST

Вход: ключи $\{1, 2, ..., n\}$ с неотрицательными частотами $p_1, p_2, ..., p_n$.

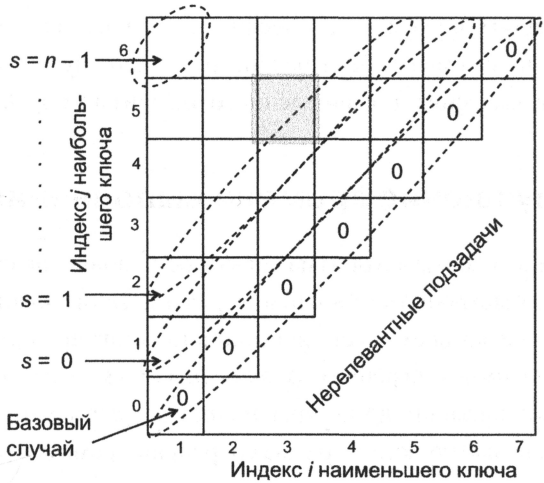
Выход: минимальное взвешенное время поиска (17.1) в бинарном дереве поиска с ключами $\{1, 2, ..., n\}$.

```
// подзадачи (i индексирует от 1, j от 0)
A := (n + 1) × (n + 1) двумерный массив
// базовые случаи (i = j + 1)
for i = 1 to n + 1 do
    A[i][i − 1] := 0
// систематически решить все подзадачи (i ≤ j)
for s = 0 to n − 1 do        // s = размер подзадачи-1
    for i = 1 to n − s do    // i + s играет роль j
        // использовать рекуррентное соотношение
        // из следствия 17.5
        A[i][i + s] :=

$$\sum_{k=i}^{i+s} p_k + \min_{r=i}^{i+s} \{\underbrace{A[i][r-1] + A[r+1][i+s]}_{\text{Случай } r}\}$$

return A[1][n] // решение наибольшей подзадачи
```

В итерации цикла, ответственной за вычисление решения подзадачи $A[i][i+s]$, все члены вида $A[i][r-1]$ и $A[r+1][i+s]$ соответствуют решениям меньших подзадач, вычисляемых в более ранних итерациях внешнего цикла `for` (или в базовых случаях). Эти значения готовы и ожидают поиска за постоянное время. Можно визуализировать массив A в алгоритме `OptBST` как двумерную

[1] См. работу алгоритма на примере задачи 17.4.

таблицу, где каждая итерация внешнего цикла `for` соответствует диагонали, а внутренний цикл `for` заполняет элементы диагонали с «юго-запада» на «северо-восток»:



Во время вычисления элемента массива $A[i][i + s]$ все релевантные члены формы $A[i][r - 1]$ и $A[r + 1][i + s]$ лежат на (ранее вычисленных) нижних диагоналях. Как и во всех наших алгоритмах динамического программирования, правильность алгоритма `OptBST` следует по индукции (на размере подзадачи) с рекуррентным соотношением из следствия 17.5, оправдывающим индукционный шаг.

Что касается времени выполнения, то не следует думать, что строка псевдокода, выполняемая в каждой итерации цикла, преобразовывается в постоянное число примитивных компьютерных операций. Вычисление суммы $\sum_{k=i}^{i+s} p_k$ и исчерпывающий поиск по $s + 1$ случаям рекуррентного соотношения занимает $O(s) = O\{n\}$ времени[1]. Существует $O\{n^2\}$ итераций (по одной на подзадачу) при совокупном времени выполнения $O(n^3)$[2].

---

[1] Как насчет оптимизации, позволяющей избежать вычисления суммы $\sum_{k=i}^{i+s} p_k$ с нуля для каждой из подзадач?

[2] В выражении (16.1) $f(n) = O(n^2)$, $g(n) = O(n)$ и $h(n) = O(1)$. Отметим, это первый пример, в котором работа подзадачи $g(n)$ не ограничена константой.

Теорема 17.6 (свойства алгоритма `OptBST`). *Для каждого множества {1, 2, ..., n} ключей и неотрицательных частот $p_1, p_2, ..., p_n$ алгоритм* `OptBST` *выполняется за время $O(n^3)$ и возвращает минимальное взвешенное время поиска в бинарном дереве поиска с ключами {1, 2, ..., n}.*

Аналогично нашим другим тематическим исследованиям, оптимальное бинарное дерево поиска может быть реконструировано путем обратной трассировки конечного массива A, вычисленного алгоритмом `OptBST`[1].

17.2.8. Улучшение времени выполнения

Кубическое время работы алгоритма `OptBST`, безусловно, не квалифицируется как невероятно быстрое. Указанный алгоритм намного быстрее, чем исчерпывающий поиск во всех (экспоненциально многочисленных) бинарных деревьях поиска, позволяя решать задачи с n, измеряемым сотнями, за разумный промежуток времени, но не задачи с n, измеряемым тысячами. В то же время небольшая настройка алгоритма сокращает время его работы до $O(n^2)$[2],

---

[1] Если алгоритм `OptBST` модифицирован для кэширования корней, определяющих значения повторяемости для каждой подзадачи (то есть значения r таковы, что $A[i]\,[i+s] = A[i]\,[r-1] + A[r]\,[i+s]$), то алгоритм восстановления выполняется за $O(n)$ времени. В противном случае он должен пересчитать корни за время $O(n^2)$.

[2] Возможен следующий вариант действий. Во-первых, заранее вычислим все суммы вида $\sum_{k=i}^{j} p_k$ за время $O(n^2)$. Затем вместе с каждым решением подзадачи $A[i]\,[j]$ сохраним выбор корня $r(i, j)$, минимизирующего $A[i]\,[r-1] + A\,[r+1]\,[j]$ или, эквивалентно, корень оптимального дерева поиска для подзадачи (при наличии нескольких корней используйте наименьший). Далее, используя свойство монотонности, добавим новый максимальный (или соответственно минимальный) элемент в подзадачу, что позволит увеличить (уменьшить) корень оптимального поиска дерева. Ясно, что любое изменение в корне должно служить перебалансированию общей частоты ключей между его левым и правым поддеревьями.

Исходя из этого предположения, для каждой подзадачи с $i < j$ оптимальный корень $r(i, j)$ будет не менее $r(i, j-1)$ и не более $r(i+1, j)$. Если же $i = j$, то $r(i, j)$ должно быть i. Таким образом, нет смысла искать абсолютно все корни между i и j (корни между $r(i, j-1)$ и $r(i+1, j)$). В худшем случае число таких корней равно n. Однако в совокупности по всем $\Theta(n^2)$ подзадачам количество изученных корней равно $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} (r(i+1, j) - r(i, j-1) + 1)$, а после отмены — только $O(n^2)$ (последнее требует проверки). Дополнительные сведения см. в статье «Оптимальные деревья бинарного поиска» Дональда Кнута (Acta Informatica, 1971).

что позволяет решать задачи с n, измеряемым в тысячах и, возможно, даже в десятках тысяч за разумный промежуток времени.

17.2.9. Решения упражнений 17.4–17.5

Решение упражнения 17.4

Правильный ответ: (б). Для первого дерева поиска «1» вносит в среднее время поиска 0,8 × 2 = 1,6 (потому что его частота равна 0,8, а время поиска равно 2), «2» вносит 0,1 × 1 = 0,1 и «3» вносит 0,1 × 2 = 0,2, при суммарном значении 1,6 + 0,1 + 0,2 = 1,9. Второе дерево поиска имеет большее максимальное время поиска (3 вместо 2), но счастливый случай поиска корня теперь гораздо вероятнее. «1» теперь вносит в среднее время поиска 0,8 × 1 = 0,8, «2» вносит 0,1 × 2 = 0,2 и «3» вносит 0,1 × 3 = 0,3, при суммарном значении 0,8 + 0,2 + 0,3 = 1,3.

Решение упражнения 17.5

Правильный ответ: (г). Наше видение алгоритма динамического программирования, «перебирающего» все возможности для корня с рекурсивным вычислением или поиском в оптимальных левых и правых поддеревьях для каждой возможности, представляется безнадежным, если только левое и правое поддеревья оптимального бинарного дерева поиска не были бы гарантированно оптимальными для соответствующей подзадачи. Таким образом, чтобы наш подход увенчался успехом, ответ должен быть либо (в), либо (г). Кроме того, по свойству дерева поиска, с учетом корня r мы знаем демографию его двух поддеревьев — ключи меньше r принадлежат левому поддереву корня, а те, которые больше r, — его правому поддереву.

ВЫВОДЫ

- В задаче о выравнивании последовательностей вход содержит две символьные цепочки и штрафы за зазоры и несовпадения, а цель состоит в том, чтобы вычислить выравнивание двух цепочек с минимально возможным суммарным штрафом.

- Задача о выравнивании последовательностей может быть решена с помощью динамического программирования за время $O(mn)$, где m и n — это длины входных цепочек.
- Подзадачи соответствуют префиксам двух входных цепочек. Существует три разных способа построения оптимального решения из оптимальных решений более мелких подзадач, что приводит к рекурренции с тремя случаями.
- В задаче об оптимальном бинарном дереве поиска вход представляет собой множество из n ключей и неотрицательных частот для них, а цель состоит в том, чтобы вычислить бинарное дерево поиска, содержащее эти ключи с минимально возможным взвешенным временем поиска.
- Задача об оптимальном бинарном дереве поиска может быть решена с помощью динамического программирования за время $O(n^3)$, где n — это число ключей. Небольшая настройка алгоритма уменьшает время работы до $O(n^2)$.
- Подзадачи соответствуют смежным подмножествам входных ключей. Существует n разных способов построения оптимального решения из оптимальных решений более мелких подзадач, что приводит к рекурренции с n случаями.

Задачи на закрепление материала

Задача 17.1. Каковы окончательные элементы массива алгоритма `NW` для входа в задачу о выравнивании последовательностей в упражнении 17.1 из раздела 17.1?

Задача 17.2. Алгоритм `Knapsack` из раздела 16.5 и алгоритм `NW` из раздела 17.1 заполняют двумерный массив с использованием удвоенного цикла for. Предположим, мы изменили порядок циклов for — буквально вырезали и вставили второй цикл перед первым, не меняя псевдокод каким-либо другим способом. Являются ли полученные алгоритмы четко определенными и правильными?

а) Ни один из алгоритмов не остается четко определенным и правильным после изменения порядка следования циклов for;

б) Оба алгоритма остаются четко определенными и правильными после изменения порядка следования циклов for;

в) Алгоритм `Knapsack` остается четко определенным и правильным после изменения порядка следования циклов for, но алгоритм `NW` — нет;

г) Алгоритм `NW` остается четко определенным и правильным после изменения порядка следования циклов for, но алгоритм `Knapsack` — нет.

Задача 17.3. Все следующие задачи принимают на входе две символьные цепочки, X и Y, длиной m и n над некоторым алфавитом Σ. Какие из них могут быть решены за время $O(mn)$? Выберите все, что применимо.

а) Рассмотрим вариант выравнивания последовательностей, в котором вместо единственного штрафа за зазор α_{gap} вам даются два положительных числа — a и b. Штраф за вставку k зазоров в цепочку теперь определяется как $ak + b$, а не $k \times \alpha_{gap}$. Остальные штрафы (за сочетание двух символов) определены, как и ранее. Цель состоит в том, чтобы вычислить минимально возможный штраф выравнивания по этой новой модели стоимости;

б) Вычислить длину самой длинной общей подпоследовательности X и Y (подпоследовательность необязательно должна содержать символы подряд. Например, самая длинная общая подпоследовательность «abcdef» и «afebcd» равна «abcd»)[1];

в) Допустим, X и Y имеют одинаковую длину n. Определите, существует ли перестановка f, отображающая каждый $i \in \{1, 2, ..., n\}$ в разное значение $f(i) \in \{1, 2, ..., n\}$, такая, что $X_i = Y_{f(i)}$ для каждого $i = 1, 2, ..., n$;

г) Вычислить длину самой длинной общей подцепочки X и Y (подцепочка — это подпоследовательность, состоящая из последовательных символов. Например, «bcd» является подцепочкой «abcdef», а «bdf» — нет).

[1] Алгоритм динамического программирования для самой длинной задачи общих подпоследовательностей лежит в основе команды `diff`, хорошо знакомой пользователям Unix и Git.

Задача 17.4. Рассмотрим экземпляр задачи об оптимальном бинарном дереве поиска с ключами {1, 2, ..., 7} и следующие ниже частоты:

Символ	Частота
1	20
2	5
3	17
4	10
5	20
6	3
7	25

Каковы окончательные элементы массива алгоритма `OptBST` из раздела 17.2?

Задача 17.5. Вспомните алгоритм `WIS` (раздел 16.2), алгоритм `NW` (раздел 17.1) и алгоритм `OptBST` (раздел 17.2). Требования к пространству этих алгоритмов пропорциональны числу подзадач: $\Theta(n)$, где соответственно n — число вершин; $\Theta(mn)$, где m и n — длины входных цепочек; и $\Theta(n^2)$, где n — число ключей.

Предположим, мы хотим вычислить значение оптимального решения и не заботимся о реконструкции. Сколько места нам требуется для выполнения соответственно каждого из трех алгоритмов?

а) $\Theta(1)$, $\Theta(1)$ и $\Theta(n)$

б) $\Theta(1)$, $\Theta(n)$ и $\Theta(n)$

в) $\Theta(1)$, $\Theta(n)$ и $\Theta(n^2)$

г) $\Theta(n)$, $\Theta(n)$ и $\Theta(n^2)$

Задачи повышенной сложности

Задача 17.6. В задаче о выравнивании последовательностей предположим, что вы знаете, что входные цепочки являются относительно похожими, в том смысле, что существует оптимальное выравнивание, которое использует не

более k зазоров, где k намного меньше, чем длины m и n цепочек. Покажите, как вычислить NW-отметку за время $O((m + n)k)$.

Задача 17.7. В игре «Тетрис» имеется семь видов плиток[1]. Разработайте алгоритм динамического программирования, который с учетом $x_1, x_2, ..., x_7$ копий каждой соответствующей плитки определяет, можно ли выложить игровое поле $10 \times n$ такими плитками (размещая их, однако, и там, где вы хотите — не обязательно в порядке, принятом в игре «Тетрис»). Время работы алгоритма должно быть полиномиальным в n.

Задачи по программированию

Задача 17.8. Реализуйте на своем любимом языке программирования алгоритмы `NW` и `OptBST` вместе с их алгоритмами реконструкции. Тестовые случаи и наборы данных для сложных задач см. на веб-сайте www.algorithmsilluminated.org.

[1] См. https://www.tetrisfriends.com/help/tips_beginner.php.

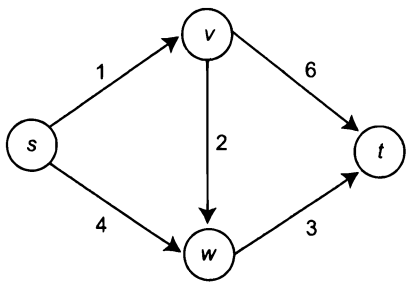
18 Кратчайшие пути повторно

Эта глава посвящена двум знаменитым алгоритмам динамического программирования для вычисления кратчайших путей в графе. Оба они медленнее, но более общи, чем алгоритм кратчайшего пути Дейкстры (глава 9 *части 2*) и алгоритм, аналогичный алгоритму Прима (разделы 15.2–15.4). Алгоритм Беллмана—Форда (Bellman—Ford) (раздел 18.2) решает задачу о кратчайшем пути с единственным истоком с отрицательными длинами ребер; его преимущество заключается также в том, что он является «более распределенным», чем алгоритм Дейкстры, и по этой причине глубоко повлиял на способ маршрутизации трафика в интернете. Алгоритм Флойда—Уоршелла (Floyd—Warshall) (раздел 18.4) также вмещает отрицательные длины ребер и вычисляет кратчайшее расстояние от *каждого* истока до каждого стока.

18.1. Кратчайшие пути с отрицательными длинами ребер

18.1.1. Задача о кратчайшем пути с единственным истоком

В задаче о кратчайшем пути с единственным истоком вход состоит из ориентированного графа $G = (V, E)$ с вещественной длиной ℓ_e для каждого ребра $e \in E$ и обозначенной стартовым пунктом $s \in V$, который называется *истоковой (исходной) вершиной* или *стартовой вершиной*. *Длина пути* — это сумма длин его ребер. Ответственность алгоритма заключается в вычислении для каждого возможного стока (пункта назначения) v минимальной длины $dist(s, v)$ ориентированного пути в G из s в v. Если такой путь не существует, то $dist(s, v)$ определяется как $+\infty$. Например, кратчайшее расстояние из s в графе



равно $dist(s, s) = 0$, $dist(s, v) = 1$, $dist(s, w) = 3$ и $dist(s, t) = 6$.

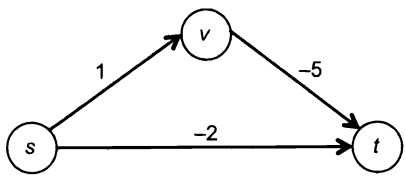
ЗАДАЧА: КРАТЧАЙШИЕ ПУТИ С ЕДИНСТВЕННЫМ ИСТОКОМ (ПРЕДВАРИТЕЛЬНАЯ ВЕРСИЯ)

Вход: ориентированный граф $G = (V, E)$, исходная вершина $s \in V$, вещественная длина ℓ_e для каждого ребра $e \in E$.[1]

Выход: кратчайшее расстояние $dist(s, v)$ для каждой вершины $v \in V$.

Например, если каждое ребро e имеет единичную длину $\ell_e = 1$, то кратчайший путь минимизирует число прыжков (то есть число ребер) между его истоком и стоком[2]. Либо, если граф представляет дорожную сеть и длина каждого ребра является ожидаемым временем в пути от одного конца до другого, то задача о кратчайшем пути с единственным истоком является задачей вычисления времени движения от начального пункта (исходной вершины) до всех возможных пунктов назначения.

Читатели *части 2* познакомились с невероятно быстрым алгоритмом Дейкстры для частного случая задачи о кратчайшем пути с единственным истоком, в которой длина каждого ребра ℓ_e является неотрицательной[3]. Алгоритм Дейкстры, при всех его достоинствах, *не* всегда правилен в графах с отрицательными длинами ребер. Он оказывается безуспешным даже в тривиальном примере, подобном следующему:



[1] Как и в задаче с минимальным остовным деревом, предположим, что входной граф не имеет параллельных ребер. Если имеется несколько ребер с одинаковым началом и концом, можно отбросить все, кроме самого короткого, не меняя сути задачи.

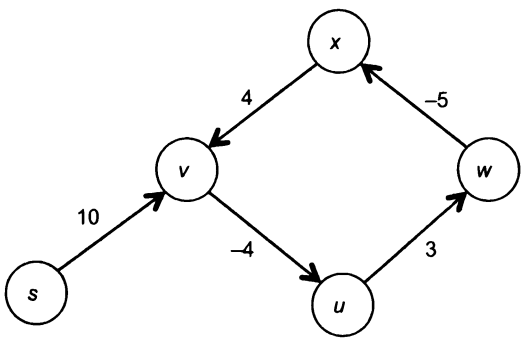
[2] Частный случай задачи кратчайшего пути из одного источника можно решить за линейное время с помощью поиска в ширину (см. раздел 8.2 *части 2*).

[3] Аналогично алгоритму Прима (раздел 15.3), реализация алгоритма Дейкстры на основе кучи выполняется за время $O((m + n) \log n)$, где m и n — число ребер и вершин входного графа соответственно.

Чтобы разместить отрицательные длины ребер, нам понадобится новый алгоритм отыскания кратчайшего пути[1].

18.1.2. Отрицательные циклы

Кого волнуют отрицательные длины ребер? Во многих приложениях, таких как вычисление направлений движения, длины ребер по умолчанию не являются отрицательными (за исключением машины времени), и беспокоиться не о чем. Но помните, что пути в графе могут представлять абстрактные последовательности решений. Например, возможно, вы хотите вычислить прибыльную последовательность финансовых операций, которая включает в себя как покупку, так и продажу. Эта задача соответствует отысканию кратчайшего пути в графе с положительными и отрицательными длинами ребер. С отрицательными длинами ребер следует быть осторожным уже при определении смысла «кратчайшего расстояния». Это достаточно ясно видно в приведенном выше примере с тремя вершинами, где $dist(s, s) = 0$, $dist(s, v) = 1$ и $dist(s, t) = -4$. А как насчет следующего ниже графа?



Этот граф имеет *отрицательный цикл*, то есть ориентированный цикл, в котором сумма длин его ребер является отрицательной. Что следует понимать под «кратчайшим путем s–v» в данном случае?

---

[1] Задача кратчайшего пути из одного источника с общими длинами ребер не может быть сведена к частному случаю неотрицательных длин ребер (с добавлением большой положительной постоянной к длине каждого ребра). В приведенном выше примере с тремя вершинами увеличение длины каждого ребра на 5 изменяет кратчайший путь с $s \to v \to t$ на $s \to t$.

Вариант № 1: разрешить циклы. Первый вариант — разрешить пути s–v, включающие один или несколько циклов. Но при наличии отрицательного цикла «кратчайший путь» попросту не может существовать! Например, на приведенном выше графе есть путь s–v с одним переходом длиной 10. Прикрепление обхода цикла в конце производит путь s–v с пятью переходами (с суммарной длиной 8). Добавление второго обхода увеличивает число переходов до 9, но уменьшает суммарную длину до 6... и так до бесконечности.

Таким образом, нет кратчайшего пути s–v, и единственным разумным определением $dist(s, v)$ является $-\infty$.

Вариант № 2: запретить циклы. Что, если мы рассмотрим пути только без циклов? При отсутствии повторяющихся вершин у нас есть только конечное число путей, о которых нужно беспокоиться. Тогда «кратчайший путь s–v» — тот, что имеет наименьшую длину. Это определение имеет смысл, но существует более тонкая проблема: в присутствии отрицательного цикла эта версия задачи о кратчайшем пути с единственным истоком называется «NP-трудной задачей». В *части 4* такие задачи рассматриваются подробно; на данный момент все, что вам нужно знать, это то, что NP-трудные задачи, в отличие от почти всех задач, которые мы изучали до сих пор в этой книжной серии, похоже, не допускают никакого алгоритма, который гарантированно будет правильным и будет работать за полиномиальное время[1].

Оба варианта являются «провальными», но следует ли нам отказаться от дальнейших попыток поиска решений? Ни в коем случае! Хотя отрицательные циклы и проблематичны, мы можем попытаться решить задачу о кратчайшем пути с единственным истоком в случаях, которые не имеют отрицательных циклов (см. пример с тремя вершинами на с. 210). Это подводит нас к пересмотренной версии задачи о кратчайшем пути с единственным истоком.

---

[1] Алгоритм с полиномиальным временем для любой NP-сложной задачи опроверг бы знаменитую гипотезу «$P \neq NP$», разрешив тем самым едва ли не важнейший вопрос всей компьютерной науки (подробнее см. *часть 4*).

ЗАДАЧА: КРАТЧАЙШИЙ ПУТЬ С ЕДИНСТВЕННЫМ ИСТОКОМ (ПЕРЕСМОТРЕННЫЙ ВАРИАНТ)

Вход: ориентированный граф $G = (V, E)$, истоковая вершина $s \in V$ и вещественная длина ℓ_e для каждого ребра $e \in E$.

Выход: один из следующих:

(i) расстояние кратчайшего пути $dist(s, v)$ для каждой вершины $v \in V$ либо

(ii) заявление о том, что G содержит отрицательный цикл.

Таким образом, мы ищем алгоритм, который либо вычисляет правильное расстояние кратчайшего пути, либо предлагает убедительное оправдание для отказа в нем (в форме отрицательного цикла). Любой такой алгоритм возвращает правильные расстояния кратчайшего пути во входных графах без отрицательных циклов[1]. Предположим, однако, что граф не имеет отрицательных циклов. Что это нам даст?

УПРАЖНЕНИЕ 18.1

Рассмотрим экземпляр задачи о кратчайшем пути с единственным истоком с n вершинами, m ребрами, стартовой вершиной s и без отрицательных циклов. Что из перечисленного является истинным? Выберите самое подходящее утверждение:

а) Для каждой вершины v, достижимой из истока s, существует кратчайший путь s–v не более чем с $n - 1$ ребрами;

б) Для каждой вершины v, достижимой из истока s, существует кратчайший путь s–v не более чем с m ребрами;

в) Для каждой вершины v, достижимой из истока s, существует кратчайший путь s–v не более чем с m ребрами;

[1] Допустим, что ребра входного графа и их длины представляют финансовые транзакции и их стоимость — с вершинами, соответствующими различным портфелям активов. Тогда отрицательный цикл соответствует риску запуска арбитражного производства. Во многих случаях такая возможность исключена, но при этом сохраняется интерес к выявлению условий, в которых «арбитражный» исход вероятен.

г) Нет конечной верхней границы (как функции от n и m) на наименьшем числе ребер в кратчайшем пути s–v.

(Решение и пояснение см. в разделе 18.1.3.)

18.1.3. Решение упражнения 18.1

Правильный ответ: (а). Имея путь P между истоковой вершиной s и некоторой стоковой вершиной v, содержащий, по крайней мере, n ребер, можно вернуть еще один s–v-путь P' с меньшим числом ребер, чем P, и длиной, не превышающей длину P. Из этого утверждения вытекает, что любой путь s–v может быть преобразован в путь s–v не более чем с $n - 1$ ребрами, который будет только короче; следовательно, существует самый короткий путь s–v не более чем с $n - 1$ ребрами.

Чтобы понять, почему это утверждение является истинным, обратите внимание, что путь P с не менее чем n ребрами посещает, по крайней мере, $n + 1$ вершин и таким образом повторяет посещение некоторой вершины w[1]. Вплетение циклического подпути между последовательными посещениями w создает путь P' с теми же конечными точками, что и P, но меньшим числом ребер; см. также рис. 15.2 и сноску 3 на с. 81. Длина P' совпадает с длиной P за вычетом суммы длин ребер в сплетенном цикле. *Поскольку входной граф не имеет отрицательных циклов*, эта сумма является неотрицательной, а длина P' меньше или равна длине P.

18.2. Алгоритм Беллмана—Форда

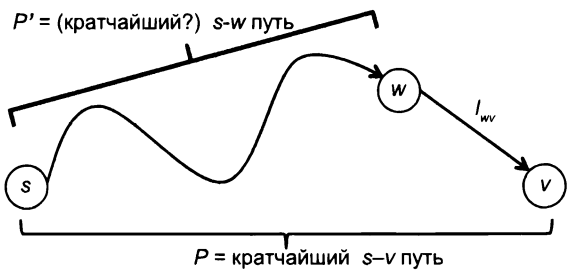
Алгоритм Беллмана—Форда решает задачу о кратчайшем пути с единственным истоком в графах с отрицательными длинами ребер в том смысле, что он либо вычисляет правильные расстояния кратчайшего пути, либо правильно

[1] Весьма сходно с принципом голубиных отверстий: независимо от того, как вы поместили $n + 1$ голубей в n отверстий, имеется одно отверстие, по крайней мере, с двумя голубями.

определяет, что входной граф имеет отрицательный цикл[1]. Этот алгоритм будет естественно вытекать из шаблона проектирования, который мы использовали в других тематических исследованиях динамического программирования.

18.2.1. Подзадачи

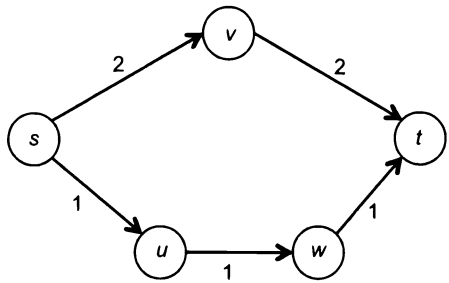
Как всегда, в том, что касается динамического программирования, наиболее важным шагом является понимание разных способов построения оптимального решения из оптимальных решений более мелких подзадач. Формулировка оптимального значения размера подзадачи может быть сложной для графовых задач. Ваша первая догадка может заключаться в том, что подзадачи должны соответствовать подграфам первоначального входного графа с размером подзадачи, равным числу вершин или ребер в подграфе. Эта идея хорошо работала в задаче о взвешенном независимом множестве на путевых графах (раздел 16.2), в которой вершины были по своей сути упорядочены, и было относительно ясно, на каких подграфах фокусироваться (префиксы входного графа). Однако в общем графе нет внутренне присущей упорядоченности вершин или ребер и мало подсказок о том, какие подграфы являются релевантными. Алгоритм Беллмана—Форда идет другим путем, вдохновленным последовательной природой *выхода* задачи о кратчайшем пути с единственным истоком (то есть в виде путей). Интуитивно можно ожидать, что префикс P' кратчайшего пути P сам по себе будет кратчайшим путем, хотя и к другому пункту назначения:



[1] Алгоритм был открыт сразу несколькими исследователями (независимо друг от друга) во второй половине 1950-х, в том числе Ричардом Э. Беллманом и Лестером Р. Фордом-младшим, хотя не исключено, что первенство здесь принадлежит именно Альфонсо Шимбелу (см. в этой связи статью Александра Шрайвера «Об истории кратчайшего пути» (Documenta Mathematica, 2012)).

Но даже если допустить, что это правда (что, как мы увидим, так и есть), то в каком смысле префикс P' решает «меньшую» подзадачу, чем первоначальный путь P? С отрицательными длинами ребер длина P' может быть даже больше, чем P. Несомненно, P' содержит меньше *ребер*, чем P, а это как раз и мотивирует одухотворенную идею алгоритма Беллмана—Форда: ввести параметр числа переходов i, который искусственно ограничивает число ребер, допустимых в пути, причем «бóльшие» подзадачи имеют бóльшие реберные бюджеты i. Тогда префикс пути действительно можно рассматривать как решение меньшей подзадачи.

Например, рассмотрим граф



и (для пункта назначения t) подзадачи, соответствующие идущим подряд значениям реберного бюджета i. Когда i равно 0 или 1, нет путей s–t с ребрами i или меньше, как нет и решений соответствующих подзадач. Расстояние кратчайшего пути с учетом ограничения числа переходов практически равно $+\infty$. Когда i равно 2, существует уникальный путь s–t не более чем с i ребрами ($s \to v \to t$) и значение подзадачи равно 4. Если мы поднимем i до 3 (или выше), то путь $s \to u \to w \to t$ становится приемлемым и уменьшает расстояние кратчайшего пути с 4 до 3.

АЛГОРИТМ БЕЛЛМАНА—ФОРДА: ПОДЗАДАЧИ

Вычислить $L_{i,v}$, длину кратчайшего пути не более чем с i ребрами из s в v в графе G с разрешенными циклами. (Если такого пути не существует, то определить $L_{i,v}$ как $+\infty$.)

(Для каждого $i \in \{0, 1, 2, ...\}$ и $v \in V$.)

Пути с циклами допускаются как варианты решения подзадачи. Если путь использует ребро многократно, то каждое использование учитывается в его бюджете подсчета переходов. Оптимальное решение вполне может проходить отрицательный цикл снова и снова, но в конечном итоге оно исчерпает свой (конечный) реберный бюджет. Для фиксированного пункта назначения v множество допустимых путей растет вместе с i, поэтому $L_{i,v}$ может уменьшаться только с увеличением i. В отличие от наших предыдущих примеров динамического программирования, каждая подзадача работает с полным входом (а не с префиксом или его подмножеством); гениальность этих подзадач — в том, как они управляют допустимым размером выхода.

Как уже говорилось, параметр i может быть сколь угодно большим положительным целым числом, и существует бесконечное число подзадач. Решение упражнения 18.1 подсказывает, что, возможно, не все из них важны. Вскоре мы увидим, что нет причин беспокоиться о подзадачах, в которых i больше числа вершин n, из чего вытекает, что существует $O(n^2)$ релевантных подзадач[1].

18.2.2. Оптимальная подструктура

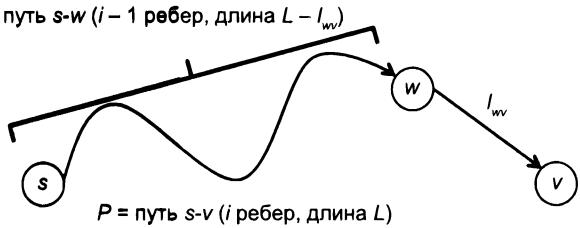
Имея на руках нашу умную коллекцию подзадач на выбор, мы можем изучить то, как оптимальные решения должны строиться из оптимальных решений меньших подзадач. Рассмотрим входной граф $G = (V, E)$ с исходной вершиной $s \in V$ и исправим подзадачу, которая определяется стоковой вершиной $v \in V$ и ограничением числа переходов $i \in \{1, 2, 3, ...\}$. Предположим, что P — это путь s–v не более чем с i ребрами, а также самый короткий такой путь. Как он должен выглядеть? Если P даже не потрудится использовать свой реберный бюджет, то ответ будет довольно простым.

Случай 1: P имеет $i - 1$ или меньше ребер. В этом случае путь P может быть немедленно интерпретирован как решение меньшей подзадачи с реберным бюджетом $i - 1$ (все еще с пунктом назначения v). Путь P должен

[1] Помните, что задача кратчайшего пути из одного пункта в действительности представляет собой совокупность подзадач (по одной на каждую вершину в качестве исходного пункта). В итоге число подзадач соответствует числу вариаций с вершинами, что имеет место и в отношении других алгоритмов динамического программирования.

быть оптимальным решением этой меньшей подзадачи, поскольку любой более короткий путь s–v не более чем с $i - 1$ ребрами также был бы и лучшим решением первоначальной подзадачи, противоречащим предполагаемой оптимальности P. Если же путь P использует свой реберный бюджет по полной, то мы следуем образцу нескольких предыдущих тематических исследований и отщипываем последнее ребро пути P, получая решение меньшей подзадачи.

Случай 2: P имеет i ребер. Пусть L обозначает длину P. Пусть P' обозначает первые $i - 1$ ребер пути P и (w, v) его конечный переход:



Префикс P' — это путь s–w не более чем с $i - 1$ ребрами и длиной $L - \ell_{wv}$.[1] Такой путь не может быть короче: если P^* является путем s–w не более чем с $i - 1$ ребрами и длиной $L^* < L - \ell_{wv}$, то дополнение ребра (w, v) к P^* произведет путь s–v не более чем с i ребрами и длиной $L^* + \ell_{wv} < (L - \ell_{wv}) + \ell_{wv} = L$, что противоречит оптимальности P для первоначальной подзадачи[2].

Этот анализ сужает возможности оптимального решения подзадачи до небольшого числа кандидатов.

Лемма 18.1 (оптимальная подструктура Беллмана—Форда). *Пусть $G = (V, E)$ равен ориентированному графу с вещественными длинами ребер и стоковой вершиной $s \in V$. Предположим, что $i \geq 1$ и $v \in V$, и пусть P равно кратчайшему пути s–v в G не более чем с i ребрами, с разрешенными циклами. Тогда P является либо*:

---

[1] Путь P' имеет $i - 1$ ребер. Требуется, однако, установить его превосходство над всеми конкурирующими s–w путями с $i - 1$ ребрами или меньше.

[2] Но если P^* уже включает вершину v, то добавление к ней ребра (w, v) образует цикл, что противоречит первоначальным условиям решаемой подзадачи.

(i) кратчайшим путем s–v не более чем с i − 1 ребрами;

либо

(ii) для некоторого w ∈ V кратчайшим путем s–w не более чем с i − 1 ребрами, дополненным ребром (w, v) ∈ E.

Сколько же именно насчитывается кандидатов?

УПРАЖНЕНИЕ 18.2

Сколько кандидатов существует для оптимального решения подзадачи с пунктом назначения v? Пусть n обозначает число вершин во входном графе. Полустепень захода и исхода вершины — это число соответственно входящих и исходящих ребер.

а) 2

б) 1 + полустепень захода вершины v

в) 1 + полустепень исхода вершины v

г) n

(Решение и пояснение см. в разделе 18.2.9.)

18.2.3. Рекурренция

Обычно следующим шагом является компиляция понимания оптимальной подструктуры в рекуррентное соотношение, реализующее исчерпывающий поиск возможных кандидатов для оптимального решения. Лучший из кандидатов, выявленных в лемме 18.1, должен быть оптимальным решением.

Следствие 18.2 (рекуррентность алгоритма Беллмана—Форда). *С допущениями и обозначениями леммы 18.1 пусть $L_{i,v}$ обозначает минимальную длину пути s–v не более чем с i ребрами с разрешенными циклами. (Если таких путей нет, то $L_{i,v} = +\infty$.) Для каждого $i \geq 1$ и $v \in V$*

$$L_{i,v} = \min \begin{Bmatrix} L_{i-1,v} & (\text{Случай 1}) \\ \min_{(w,v)\in E} \{L_{i-1,w} + \ell_{wv}\} & (\text{Случай 2}) \end{Bmatrix}. \qquad (18.1)$$

Внешнее «min» в рекурренции реализует исчерпывающий поиск над случаем 1 и случаем 2. Внутреннее «min» реализует исчерпывающий поиск внутри случая 2 по всем возможным вариантам для окончательного перехода кратчайшего пути. Если $L_{i-1,v}$ и все соответствующие $L_{i-1,w}$ равны $+\infty$, то v недостижима из s за i или меньше переходов, а мы интерпретируем рекурренцию как вычисляющую $L_{i,v} = +\infty$.

18.2.4. Когда следует остановиться?

Располагая большим опытом динамического программирования, вы (в порядке реагирования на рекуррентное соотношение в следствии 18.2) можете записать алгоритм динамического программирования, который используется многократно для систематического решения каждой подзадачи. Предположительно, алгоритм начнет с решения наименьших подзадач (с реберным бюджетом $i = 0$), за которыми будут другие наименьшие подзадачи (с $i = 1$) и т. д. Правда, в подзадачах, определенных в разделе 18.2.1, реберный бюджет i может быть сколь угодно большим положительным целым числом, что означает и бесконечное число подзадач. Как мы узнаем, когда следует остановиться?

Хороший критерий для выбора момента остановки следует из наблюдения, что решения данной партии подзадач с фиксированным бюджетом i и v варьированиями по всем возможным пунктам назначения зависят только от решений предыдущей партии подзадач (с бюджетом $i - 1$). Таким образом, если одна партия подзадач когда-либо будет иметь точно такие же оптимальные решения, что и предыдущая (со случаем 1 победы рекурренции для каждого пункта назначения), то эти оптимальные решения останутся неизменными навсегда.

Лемма 18.3 (критерий останова Беллмана—Форда). *В рамках допущений и обозначений следствия 18.2, если для некоторого $k \geq 0$*

$$L_{k+i,v} = L_{k,v} \text{ для каждого пункта назначения } v,$$

то:

а) $L_{i,v} = L_{k,v}$ *для каждого* $i \geq k$ *и пункта назначения* v; *и*

б) *для каждого пункта назначения* v $L_{k,v}$ *является правильным кратчайшим расстоянием* $dist(s, v)$ *от* s *до* v *в* G.

Доказательство: по принятому допущению вход в рекурренцию в (18.1) в $(k+2)$-й партии подзадач (то есть $L_{k+1,v}$) является таким же, как и для $(k+1)$-й партии (то есть $L_{k,v}$). Таким образом, выход рекурренции ($L_{k+2,v}$-й) также будет тем же, что и для предыдущей партии ($L_{k+1,v}$-й). Повтор аргумента требуемое количество раз показывает, что $L_{i,v}$ остаются одинаковыми для всех партий $i \geq k$. Это доказывает часть (а).

Что касается части (б), то предположим от противного, что $L_{k,v} \neq dist(s, v)$ для некоторого пункта назначения v. Поскольку $L_{k,v}$ является минимальной длиной пути s–v не более чем с k переходами, то должен иметься путь s–v с $i > k$ переходами и длиной меньше $L_{k,v}$. Но тогда $L_{i,v} < L_{k,v}$, что противоречит части (а) леммы. *Ч. Т. Д.*

В соответствии с леммой 18.3 целесообразно остановиться, как только решения подзадачи стабилизируются, то есть при $L_{k+1,v} = L_{k,v}$ для некоторых $k \geq 0$ и всех $v \in V$. Но произойдет ли это когда-нибудь? В общем случае нет. Однако если входной граф не имеет отрицательных циклов, то решения подзадач гарантированно стабилизируются к моменту, когда i достигнет n, числа вершин.

Лемма 18.4 (Беллман—Форд без отрицательных циклов). *В рамках допущений и обозначений следствия 18.2, а также при условии, что входной граф G не имеет отрицательных циклов, $L_{n,v} = L_{n-1,v}$ для каждого пункта назначения v, где n — это число вершин во входном графе.*

Доказательство: из решения упражнения 18.1 следует, что для каждого пункта назначения v существует кратчайший путь s–v не более чем с $n-1$ ребрами. Другими словами, увеличение реберного бюджета i с $n-1$ до n (либо до любого большего числа) не влияет на минимальную длину пути s–v. *Ч. Т. Д.*

Из леммы 18.4 следует, что если входной граф не имеет отрицательного цикла, то решения подзадач стабилизируются n-й партией. И наоборот, если решения подзадач не стабилизируются n-й партией, то входной граф *имеет* отрицательный цикл. В совокупности леммы 18.3 и 18.4 сообщают последнюю партию подзадач, с которыми нам предстоит иметь дело: партию с $i = n$. Если решения подзадач стабилизируются (при $L_{n,v} = L_{n-1,v}$ для всех $v \in V$), то из леммы 18.3 следует, что $L_{n-1,v}$ является правильным расстоянием кратчайшего пути. Если решения подзадач не стабилизируются (при $L_{n,v} \neq L_{n-1,v}$ для некоторого $v \in V$), то из противопоставления лемме 18.4 следует, что входной граф G

содержит отрицательный цикл, и в таком случае алгоритм освобождается от вычисления кратчайших расстояний. Вспомним в этой связи определение задачи в разделе 18.1.2.

18.2.5. Псевдокод

Теперь знаменитый алгоритм Беллмана—Форда напишется сам собой: используйте рекурренцию из следствия 18.2 для систематического решения всех подзадач, вплоть до реберного бюджета $i = n$.

BELLMAN—FORD

Вход: ориентированный граф $G = (V, E)$, представленный в виде списков смежности, истоковая вершина $s \in V$ и вещественная длина ℓ_e для каждого $e \in E$.

Выход: $dist(s, v)$ для каждой вершины $v \in V$ либо объявление, что G содержит отрицательный цикл.

```
// подзадачи (i индексируется от 0, v индексирует V)
A := (n + 1) × n двумерный массив
// базовые случаи (i = 0)
A[0][s] := 0
for each v ≠ s do
    A[0][v] := +∞
// систематически решить все подзадачи
for i = 1 to n do                    // размер подзадачи
    stable := TRUE                   // для досрочной остановки
    for v ∈ V do
        // использовать рекурренцию из следствия 18.2
        A[i][v] :=
```

$$\min\{\underbrace{A[i-1][v]}_{\text{Случай 1}},\ \underbrace{\min_{(w,v)\in E}\{A[i-1][w]+\ell_{wv}\}}_{\text{Случай 2}}\}$$

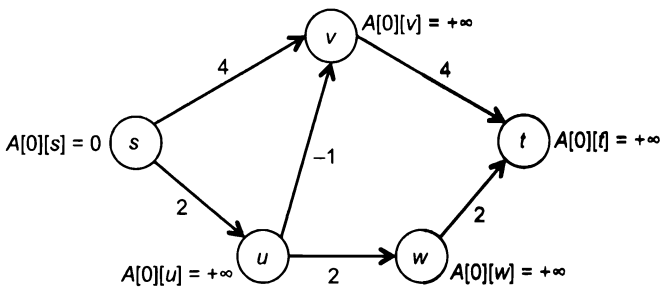
```
        if A[i][v] ≠ A[i − 1][v] then
```

```
        stable := FALSE
    if stable = TRUE then      // выполнено леммой 18.3
        return {A[i − 1][v]}_{v∈V}
// не удалось стабилизироваться на n итерациях
return «отрицательный цикл» // исправлено леммой 18.4
```

Сдвоенный цикл for отражает два параметра, используемые для определения подзадач: реберный бюджет i и стоковую вершину v. К моменту, когда итерация цикла должна вычислить решение подзадачи $A[i][v]$, все значения формы $A[i-1][v]$ или $A[i-1][w]$ уже были вычислены на предыдущей итерации внешнего цикла for (либо в базовых случаях) и готовы к тому, что их отыщут за постоянное время. Индукция (на i) показывает, что алгоритм `Bellman-Ford` правильно решает каждую подзадачу, присваивая элементу $A[i][v]$ правильное значение $L_{i,v}$; рекурренция в следствии 18.2 оправдывает индукционный шаг. Если решения подзадач стабилизируются, то указанный алгоритм возвращает правильные расстояния кратчайшего пути (по лемме 18.3). Если нет, то алгоритм правильно заявляет, что входной граф содержит отрицательный цикл (по лемме 18.4).

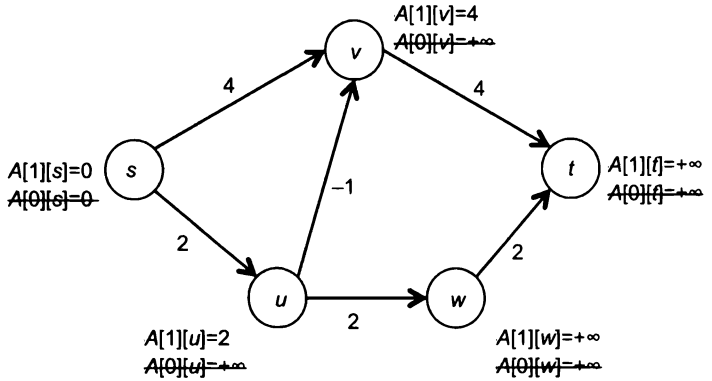
18.2.6. Пример

Для примера алгоритма `Bellman-Ford` в действии рассмотрим следующий входной граф:

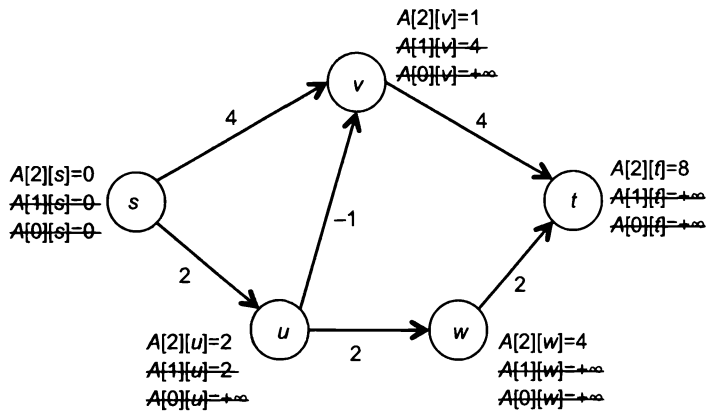


Вершины помечены решениями первой партии подзадач (при $i = 0$). Каждая итерация алгоритма оценивает рекурренцию (18.1) в каждой вершине, используя значения, вычисленные на предыдущей итерации. На первой итерации

рекурренция вычисляет 0 в s (s не имеет входящих ребер, поэтому случай 2 рекурренции является пустым), вычисляет 2 в U (потому что $A[0][s] + \ell_{su} = 2$), вычисляет 4 в B (потому что $A[0][s] + \ell_{sv} = 4$) и вычисляет $+\infty$ в w и t (потому что $A[0][u]$ и $A[0][v]$ оба равны $+\infty$):

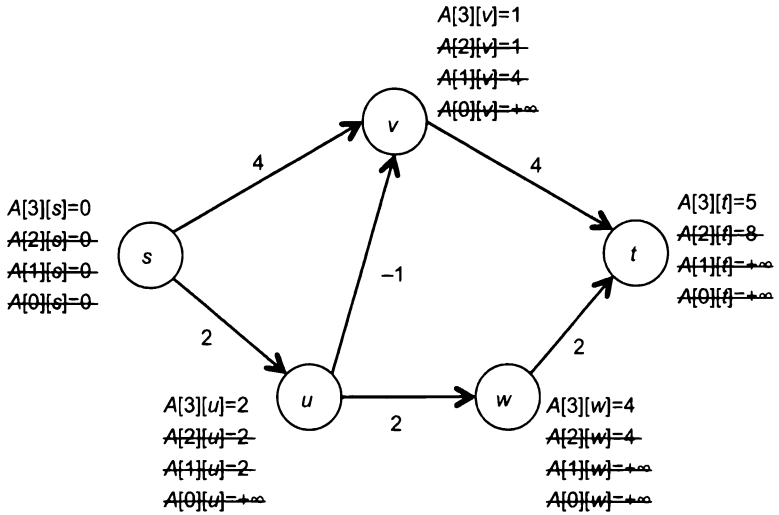


На следующей итерации как s, так и u наследуют решения от предыдущей итерации. Значение при v падает с 4 (соответствуя пути из одного перехода $s \to v$) до 1 (соответствуя пути из двух переходов $s \to u \to v$). Новые значения в w и t равны 4 (потому что $A[1][u] + \ell_{uw} = 4$) и 8 (потому что $A[1][v] + \ell_{vt} = 8$):

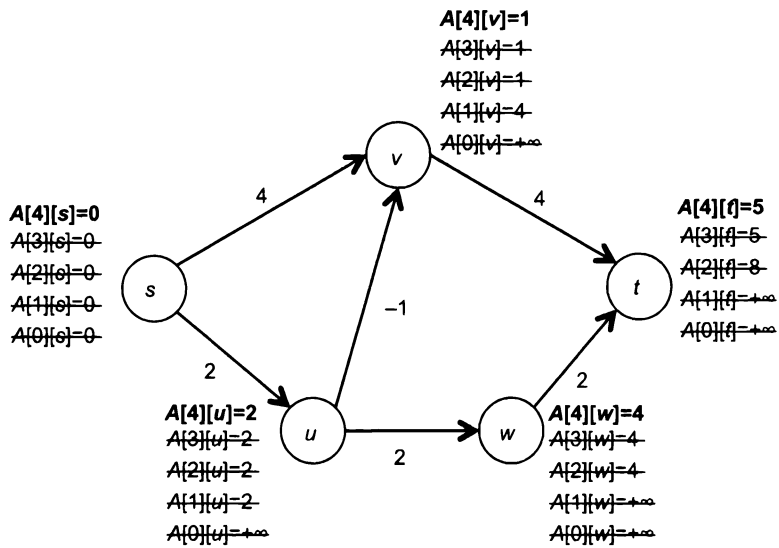


Обратите внимание, что уменьшение расстояния кратчайшего пути до v на этой итерации распространяется на t, начиная со следующей итерации.

На третьей итерации значение в t падает до 5 (потому что $A[2][v] + \ell_{vt} = 5$, что лучше, чем $A[2][t] = 8$ и $A[2][w] + \ell_{wt} = 6$), а остальные четыре вершины наследуют решения от предыдущей итерации:



На четвертой итерации ничего не меняется:



и в этот момент алгоритм останавливается с правильными кратчайшими расстояниями[1].

18.2.7. Время выполнения

Анализ времени выполнения алгоритма Беллмана—Форда более интересен, чем анализ, проводимый в отношении других алгоритмов динамического программирования.

УПРАЖНЕНИЕ 18.3

Каково время работы алгоритма `Bellman-Ford` как функции от m (числа ребер) и n (числа вершин)? Выберите наиболее верное утверждение:

а) $O(n^2)$

б) $O(mn)$

в) $O(n^3)$

г) $O(mn^2)$

(Решение и пояснение см. в разделе 18.2.9.)

Обобщая все, что мы теперь знаем об алгоритме `Bellman-Ford`, обратимся к следующей теореме.

Теорема 18.5 (свойства алгоритма `Bellman-Ford`). *Для каждого входного графа $G = (V, E)$ с n вершинами, m ребрами, вещественными длинами ребер и истоковой вершиной s алгоритм* `Bellman-Ford` *выполняется за время $O(mn)$ и либо:*

(i) возвращает кратчайшее расстояние от s до каждого пункта назначения $v \in V$,

либо

(ii) обнаруживает, что G содержит отрицательный цикл.

[1] Пример алгоритма в действии на входном графе с отрицательным циклом см. в задаче 18.1.

Как обычно, кратчайший путь может быть реконструирован путем обратной трассировки конечного массива A, вычисляемого алгоритмом Беллмана—Форда[1].

18.2.8. Маршрутизация интернета

Алгоритм Беллмана—Форда решает более общую задачу, чем алгоритм Дейкстры (потому что он вмещает в себя отрицательные длины ребер). Его второе преимущество заключается в том, что он более «распределен», чем алгоритм Дейкстры, и по этой причине сыграл более заметную роль в эволюции протоколов маршрутизации интернета[2]. Вычисление рекуррентного соотношения (18.1) в вершине v требует информации только о вершинах, непосредственно соединенных с v: вершинах w с ребром (w, v). Значит, алгоритм Беллмана—Форда может быть реализован даже в масштабе интернета, когда каждая машина «общается» только со своими ближайшими соседями, выполняя только локальные вычисления и не ведая о том, что происходит в остальной части сети. Действительно, на алгоритме Беллмана—Форда базируются ранние протоколы маршрутизации интернета RIP и RIP2 — пример того, как алгоритмы формируют мир таким, каким мы его знаем[3].

---

[1] Для восстановления рекомендуется добавить строку кода, которая кэширует с каждой вершиной v самый последний предшественник w, запускающий обновление Case 2 в форме $A[i][v] := A[i-1][w] + \ell_{wv}$. Например, в случае с входным графом (раздел 18.2.6) предшественник вершины v будет инициализирован нулем, сброшен в s после первой итерации, а затем в u после второй итерации. После этого можно восстановить кратчайший s–v (путь назад в течение $O(n)$ времени с использованием конечной партии предшественников, начиная с v и следуя за предшественником обратно к s). Поскольку каждый пакет решений и предшественников подзадач зависит только от тех, что были в предыдущем пакете, как для прямого, так и для восстановительного проходов требуется только пространство $O(n)$ (аналогично задаче 17.5). Подробнее см. видео на www.algorithmsilluminated.org.

[2] Алгоритм Беллмана—Форда выявлен за 10 лет до ARPANET, предшественницы интернета (то есть в 1959 году. — *Примеч. ред.*).

[3] «RIP», или «протокол маршрутизации информации». Подробности протоколов RIP и RIP2 описаны в RFC (букв. «запрос комментариев»; основной механизм, с помощью которого изменения в интернет-стандартах проверяются и распространяются) 1058 и 2453 соответственно. О некоторых сопутствующих технических проблемах см. видео на сайте www.algorithmsilluminated.org.

18.2.9. Решения упражнений 18.2–18.3

Решение упражнения 18.2

Правильный ответ: (б). Лемма оптимальной подструктуры (лемма 18.1) изложена так, как если бы было два кандидата на оптимальное решение, но случай 2 содержит несколько подслучаев: по одному для каждого возможного конечного перехода (w, v) пути s–v. Таким образом, случай 1 вносит одного кандидата, а случай 2 — ряд кандидатов, равный полустепени захода вершины v[1]. Эта полустепень захода может быть столь же большой, как $n - 1$ в ориентированном графе (без параллельных ребер), но, как правило, намного меньше, в особенности в разреженных графах.

Решение упражнения 18.3

Правильный ответ: (б). Алгоритм `Bellman-Ford` решает $(n + 1) \times n = O(n^2)$ разных подзадач, где n — это число вершин. Если бы алгоритм выполнял только постоянный объем работы (в расчете на подзадачу, как и во всех наших предыдущих алгоритмах динамического программирования, кроме `OptBST`), то время работы алгоритма также было бы $O(n^2)$. Но решение подзадачи для пункта назначения v сводится к вычислению рекурренции из следствия 18.2, которое, следуя упражнению 18.2, включает исчерпывающий поиск по 1 + in-deg(v) кандидатов, где in-deg(v) — это число ребер, входящих в v. Поскольку полустепень захода вершины может быть столь же большой, как и $n - 1$, это, казалось бы, дает временну́ю границу $O(n)$ для каждой подзадачи при совокупной временно́й границе $O(n^3)$. Дабы добиться лучшего, сфокусируемся на фиксированной итерации внешнего цикла for алгоритма с неким фиксированным значением i. Суммарная работа, выполняемая во всех итерациях внутреннего цикла for, пропорциональна

$$\sum_{v \in V} \left(1 + in\text{-}\deg(v)\right) = \underbrace{n + \sum_{v \in V} in\text{-}\deg(v)}_{=m}.$$

[1] При условии, что входной граф представлен с использованием списков смежности (в частности, что массив входящих ребер связан с каждой вершиной), исчерпывающий поиск может быть реализован за время, выраженное как 1 + in-deg (v).

Сумма полустепеней захода также имеет более простое имя: m, число ребер. Чтобы увидеть это, представьте себе удаление всех ребер из входного графа и добавление их обратно, по одному. Каждое новое ребро добавляет 1 и к совокупному числу ребер, и в полустепень захода ровно одной вершины (голову этого ребра).

Таким образом, суммарная работа, выполняемая на каждой из внешних итераций цикла for, равна $O(m + n) = O(m)$[1]. Существует не более n таких итераций. Работа $O(n)$ выполняется вне сдвоенного цикла for, приводя к совокупной временной границе $O(mn)$. В разреженных графах, где m является линейным или почти линейным в n, эта временная граница намного лучше, чем более наивная граница $O(n^3)$.

18.3. Задача о кратчайшем пути для всех пар

18.3.1. Определение задачи

Зачем довольствоваться вычислением кратчайших расстояний только из одной-единственной истоковой вершины? Например, алгоритм вычисления направлений движения должен учитывать любую возможную стартовую точку; это соответствует *задаче о кратчайшем пути для всех пар*. При этом мы по-прежнему допускаем отрицательные длины ребер и отрицательные циклы во входном графе.

ЗАДАЧА: КРАТЧАЙШИЕ ПУТИ ДЛЯ ВСЕХ ПАР

Вход: ориентированный граф $G = (V, E)$ с n вершинами и m ребрами и вещественная длина ℓ_e для каждого ребра $e \in E$.

Выход — один из следующих:

[1] Предполагается, что m является, по меньшей мере, постоянным временем n, как в случае, когда каждая вершина v достижима из исходной вершины s. Подумайте, как настроить алгоритм для получения границы времени за итерацию $O(m)$, отбросив предположение.

(i) кратчайшее расстояние $dist(v, w)$ для каждой упорядоченной пары вершин $v, w \in V$

либо

(ii) заявление о том, что G содержит отрицательный цикл.

В задаче о кратчайшем пути для всех пар нет истоковой вершины. В случае (i) алгоритм отвечает за вывод n^2 чисел[1].

18.3.2. Сведение до кратчайших путей с единственным истоком

Если вы ищете редукции (как и должно быть; см. с. 134), то вы, возможно, уже видите, как применить свой постоянно растущий алгоритмический арсенал к задаче о кратчайшем пути для всех пар. Одним из естественных подходов является повторное использование подпрограммы, которая решает задачу о кратчайшем пути с единственным истоком (например, алгоритм Беллмана—Форда).

УПРАЖНЕНИЕ 18.4

Сколько вызовов подпрограммы кратчайшего пути с единственным истоком необходимо для решения задачи о кратчайшем пути для всех пар? Как обычно, n обозначает число вершин.

а) 1

б) $n - 1$

в) n

г) n^2

(Решение и пояснение см. в разделе 18.3.3.)

[1] Применение алгоритмов при решении задачи кратчайшего пути для всех пар особенно значимо в случае вычисления транзитивного замыкания бинарного отношения. Последняя задача сходна с задачей о достижимости всех пар, когда для заданного графа требуется определить все пары вершин v, w, для которых в графе содержится хотя бы один путь v–w (то есть для которого расстояние по кратчайшему пути подсчета прыжков является конечным).

Включение алгоритма Беллмана—Форда (теорема 18.5) для подпрограммы кратчайшего пути с единственным истоком в упражнение 18.4 дает $O(mn^2)$-временной алгоритм для задачи о кратчайшем пути всех пар[1].

Отметим, граница времени выполнения $O(mn^2)$ особенно проблематична в плотных графах. Например, если $m = \Theta(n^2)$, то время выполнения является *квадратичным* в n — время выполнения, которое мы не встречали раньше и, надеюсь, никогда не увидим.

18.3.3. Решение упражнения 18.4

Правильный ответ: (в). Один вызов подпрограммы кратчайшего пути с единственным истоком вычислит расстояния кратчайшего пути от одной вершины s до каждой вершины графа (всего n чисел из n^2 необходимых). Однократный вызов подпрограммы для каждого из n вариантов для s вычисляет расстояния кратчайшего пути для каждого возможного истока и стока (пункта назначения)[2].

18.4. Алгоритм Флойда—Уоршелла

В этом разделе решается задача о кратчайшем пути для всех пар с нуля и представлено наше окончательное тематическое исследование парадигмы проектирования алгоритмов динамического программирования на основе алгоритма Флойда—Уоршелла[3].

18.4.1. Подзадачи

Графы — это многосложные объекты. Придумать правильное множество подзадач для решения динамическим программированием графовой задачи

---

[1] Алгоритм Дейкстры может заменить алгоритм Беллмана—Форда, если длины ребер неотрицательны, при этом время работы улучшается до $O(mn \log n)$. В разреженных графах (с $m = O(n)$ или близким к данному значению) это позволяет приблизиться к желаемому результату (поскольку простое написание вывода потребует уже квадратичного времени).

[2] Входной граф с отрицательным циклом будет обнаружен при вызове подпрограммы кратчайшего пути из одного источника.

[3] Назван в честь Роберта В. Флойда и Стивена Уоршалла; самостоятельно выявлен и рядом других исследователей на рубеже 1950–1960-х гг.

непросто. В основе подзадач в алгоритме Беллмана—Форда для задачи о кратчайшем пути с единственным истоком (раздел 18.2.1) лежит постулат о работе с исходным входным графом и наложении искусственного ограничения на число ребер, разрешенных в решении подзадачи. В этом случае реберный лимит служит мерой размера подзадачи, а префикс оптимального решения подзадачи можно интерпретировать как решение меньшей подзадачи (с тем же истоком, но другим стоком).

В алгоритме Флойда—Уоршелла предлагается пойти на шаг дальше, искусственно ограничив *идентификаторы вершин*, которые могут находиться в решении. Для определения подзадач рассмотрим входной граф $G = (V, E)$ и произвольно назначим его вершинам имена 1, 2, ..., n (где $n = |V|$). Подзадачи индексируются префиксами $\{1, 2, ..., k\}$ вершин, причем k служит мерой размера подзадачи, а также — одновременно — истоком v и стоком w.

АЛГОРИТМ ФЛОЙДА—УОРШЕЛЛА: ПОДЗАДАЧИ

Вычислить $L_{k,v,w}$, минимальную длину пути во входном графе G, который:

(i) начинается в v;

(ii) заканчивается в w;

(iii) использует вершины только из $\{1, 2, ..., k\}$ как внутренние вершины[1] и

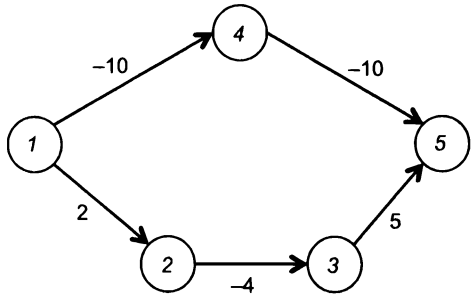
(iv) не содержит ориентированного цикла.

(Если такой путь не существует, то определить $L_{k,v,w}$ как $+\infty$.)

(Для каждого $k \in \{0, 1, 2, ..., n\}$ и $v, w \in V$.)

Существует $(n + 1) \times n \times n = O(n^3)$ подзадач, которые являются линейным числом для каждого из значений n^2 в выходе. Пакет самых больших подзадач (при $k = n$) соответствует первоначальной задаче. Для фиксированного истока v и стока w множество допустимых путей растет вместе с k, поэтому $L_{k,v,w}$ может уменьшаться только с увеличением k. Рассмотрим граф

[1] Внутренними вершинами пути являются все, за исключением конечных.



и (для истока 1 и стока 5) подзадачи, соответствующие идущим подряд значениям префиксной длины k. Когда k равно 0, 1 или 2, нет путей из 1 в 5, поэтому каждая внутренняя вершина принадлежит префиксу $\{1, 2, ..., k\}$, и решение подзадачи равно $+\infty$. Когда $k = 3$, путь $1 \to 2 \to 3 \to 5$ становится единственным подходящим путем; он имеет длину $2 + (-4) + 5 = 3$. Путь с двумя переходами дисквалифицируется, поскольку он включает вершину 4 в качестве внутренней вершины. Путь с тремя переходами квалифицируется, даже если вершина 5 не принадлежит префиксу $\{1, 2, 3\}$ в качестве стока; этой вершине предоставляется исключение. Когда $k = 4$ (или больше), решением подзадачи является длина истинного кратчайшего пути $1 \to 4 \to 5$, которая равна -20.

В следующем разделе мы увидим, что выигрыш от определения подзадач, таким образом, состоит в том, что есть только два кандидата для оптимального решения подзадачи (в зависимости от того, использует ли он последнюю допустимую вершину k или нет)[1]. Это приводит к алгоритму динамического программирования, который выполняет только $O(1)$-ю работу на подзадачу и, значит, быстрее, чем n вызовов алгоритма Беллмана—Форда (с временем выполнения $O(n^3)$ вместо $O(mn^2)$)[2].

18.4.2. Оптимальная подструктура

Рассмотрим входной граф $G = (V, E)$ с вершинами, помеченными от 1 до n, и зафиксируем подзадачу, определенную истоковой вершиной v, стоковой вер-

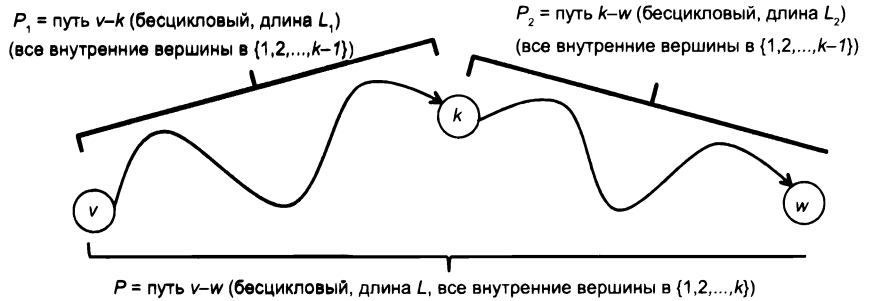
[1] Напротив, число возможных решений для подзадачи в алгоритме Беллмана—Форда зависит от степени назначения (упражнение 18.2).

[2] Отбросим случай, когда m намного меньше n (см. также сноску на с. 229).

шиной w и префиксной длиной $k \in \{1, 2, ..., n\}$. Предположим, что P является путем v–w без циклов и всех внутренних вершин в $\{1, 2, ..., k\}$ и, более того, является кратчайшим таким путем, при этом последняя допустимая вершина k либо появляется как внутренняя вершина P, либо нет.

Случай 1: вершина k не является внутренней вершиной P. В этом случае путь P может быть интерпретирован как решение меньшей подзадачи с префиксной длиной $k - 1$, по-прежнему с истоком v и стоком w. Путь P должен быть оптимальным решением меньшей подзадачи; любое более превосходящее решение также было бы для первоначальной подзадачи превосходящим.

Случай 2: вершина k является внутренней вершиной P. В этом случае путь P можно интерпретировать как объединение *двух* решений меньших подзадач: префикса P_1 пути P, который перемещается из v в k, и суффикса P_2 пути P, который перемещается из k в W.



Вершина k появляется в P однократно (P не имеет циклов) и поэтому не является внутренней вершиной P_1 или P_2. Таким образом, мы можем рассматривать P_1 и P_2 как решения меньших подзадач соответственно с истоками v и k и стоками k и w, где все внутренние вершины находятся в $\{1, 2, ..., k - 1\}$[1,2]. Вы можете попытаться угадать следующий шаг: мы хотим доказать, что P_1 и P_2 на самом деле являются *оптимальными* решениями этих меньших подзадач. Пусть L, L_1 и L_2 обозначают длины соответственно P, P_1 и P_2. Поскольку P является объединением P_1 и P_2, $L = L_1 + L_2$.

---

[1] Это объясняет, почему подзадачи Флойда—Уоршелла, в отличие от подзадач Беллмана—Форда, включают условие об отсутствии цикла (iv).

[2] Этот подход едва ли применим для задачи кратчайшего пути из одного источника, так как суффиксный путь P_2 будет иметь неверную исходную вершину.

Предположим от противного, что P_1 не является оптимальным решением его подзадачи; аргумент для P_2 аналогичен. Тогда существует путь без цикла P_1^* из v в k с внутренними вершинами в $\{1, 2, ..., k-1\}$ и длиной $L_1^* < L_1$. Но тогда конкатенация P_1^* и P_2 будет бесцикловым путем P^* из v в w с внутренними вершинами в $\{1, 2, ..., k\}$ и длиной $L_1^* + L_2 < L_1 + L_2 = L$, противореча предполагаемой оптимальности P.

УПРАЖНЕНИЕ 18.5

Видите ли вы какие-либо дефекты в приведенном выше аргументе? Выберите все, что применимо.

а) конкатенация P^* из P_1^* и P_2 не обязательно должна иметь исток v;

б) P^* не обязательно должен иметь сток w;

в) P^* не обязательно должен иметь внутренние вершины только в $\{1, 2, ..., k\}$;

г) P^* не обязательно должен быть бесцикловым;

д) P^* не обязательно должен иметь длину меньшую, чем L;

е) нет никаких дефектов.

(Решение и пояснение см. в разделе 18.4.6.)

Теперь предположим, что конкатенация P^* из P_1^* и P_2 содержит цикл. Многократно сплетая циклы между собой (как на рис. 15.2 и в сноске 3 на с. 81), можно извлечь из P^* бесцикловый путь $\hat{P}$ с тем же истоком (v) и стоком (w) и только с меньшим числом внутренних вершин. Длина равна длине L^* пути P^*, за вычетом суммы длин ребер в сплетенных циклах.

Если входной граф не имеет отрицательных циклов, то сплетение циклов может только сократить путь, и в этом случае длина $\hat{P}$ не превышает L^*. В этом случае мы спасли доказательство: $\hat{P}$ является бесцикловым путем v–w, в котором все внутренние вершины находятся в $\{1, 2, ..., k\}$ и длиной не более $L^* < L$, что противоречит предполагаемой оптимальности первоначального пути P. Тогда мы можем заключить, что вершина k действительно разбивает оптимальное решение P на оптимальные решения P_1 и P_2 для соот-

ветствующих меньших подзадач. Для оптимальной подструктуры алгоритма Флойда—Уоршелла справедлива следующая лемма.

Лемма 18.6 (оптимальная подструктура алгоритма Флойда—Уоршелла). *Пусть $G = (V, E)$ равно ориентированному графу с вещественными длинами ребер и без отрицательных циклов, с $V = \{1, 2, ..., n\}$. Предположим $k \in \{1, 2, ..., n\}$ и $v, w \in V$, и пусть P равно бесцикловому пути v–w минимальной длины в G, в котором все внутренние вершины находятся в $\{1, 2, ..., k\}$. Тогда P является либо*:

(i) бесцикловым путем v–w минимальной длины, в котором все внутренние вершины находятся в $\{1, 2, ..., k - 1\}$,

либо

(ii) конкатенацией бесциклового пути v–k минимальной длины, в котором все внутренние вершины находятся в $\{1, 2, ..., k - 1\}$, и бесциклового пути k–w минимальной длины, в котором все внутренние вершины находятся в $\{1, 2, ..., k - 1\}$.

Или в форме рекурренции:

Следствие 18.7 (рекуррентное соотношение Флойда—Уорсхолла). *С принятыми допущениями и обозначением леммы 18.6 пусть $L_{k,v,w}$ обозначает минимальную длину бесциклового пути v–w, в котором все внутренние вершины находятся в $\{1, 2, ..., k\}$. (Если таких путей нет, то $L_{k,v,w} = +\infty$.) Для каждого $k \in \{1, 2, ..., n\}$ и $v, w \in V$,*

$$L_{k,v,w} = \min\begin{Bmatrix} L_{k-1,\,v,w} & (\text{Случай 1}) \\ L_{k-1,v,k} + L_{k-1,\,k,w} & (\text{Случай 2}) \end{Bmatrix}. \qquad (18.2)$$

18.4.3. Псевдокод

Предположим, что входной граф не имеет отрицательных циклов, и в этом случае применимы лемма 18.6 и следствие 18.7. Используем рекурренцию для систематического решения всех подзадач, от самых маленьких до самых больших. Для начала выясним, каковы решения базовых случаев (при $k = 0$ и без внутренних вершин).

УПРАЖНЕНИЕ 18.6

Пусть $G = (V, E)$ равно входному графу. Какова $L_{0,v,w}$ в случае, когда (i) $v = w$; (ii) (v, w) является ребром G, и (iii) $v \neq w$ и (v, w) не является ребром G?

а) 0, 0 и $+\infty$

б) 0, ℓ_{vw} и ℓ_{vw}

в) 0, ℓ_{vw} и $+\infty$

г) $+\infty$, ℓ_{vw} и $+\infty$

(Решение и пояснение см. в разделе 18.4.6.)

Алгоритм Флойда—Уоршелла вычисляет базовые случаи с помощью решения упражнения 18.6 и остальные подзадачи, используя рекурренцию в следствии 18.7. Заключительный цикл for в псевдокоде проверяет наличие во входном графе отрицательного цикла и объясняется в разделе 18.4.4 (примеры алгоритма в действии см. в задачах 18.4 и 18.5).

FLOYD-WARSHALL

Вход: ориентированный граф $G = (V, E)$, представленный в виде списков смежности или матрицы смежности, и вещественная длина ℓ_e для каждого ребра $e \in E$.

Выход: $dist(v, w)$ для каждой пары вершин $v, w \in V$ или объявление, что G содержит отрицательный цикл.

```
Обозначить вершины V = {1, 2, ..., n} произвольно
// подзадачи (k индексируется с 0, v, w с 1)
A := (n + 1) × n × n трехмерный массив
// базовые случаи (k = 0)
for v = 1 to n do
    for w = 1 to n do
        if v = w then
            A[0][v][w] := 0
        else if (v, w) ребро G then
```

$A[0][v][w] := \ell_{vw}$

else

$A[0][v][w] := +\infty$

// систематически решить все подзадачи

for $k = 1$ to n **do** // размер подзадачи

for $v = 1$ to n **do** // исток

for $w = 1$ to n **do** // сток

// использовать рекурренцию из следствия 18.7

$$A[k][v][w] := \min\{\underbrace{A[k-1][v][w]}_{\text{Случай 1}},\ \underbrace{A[k-1][v][k] + A[k-1][k][w]}_{\text{Случай 2}}\}$$

// проверить на наличие отрицательного цикла

for $v = 1$ to n **do**

if $A[n][v][v] < 0$ **then**

return «отрицательный цикл» // см. лемму 18.8

return $\{A[n][v][w]\}_{v,w \in V}$

Указанный алгоритм использует трехмерный массив подзадач и соответствующий устроенный цикл `for`, поскольку подзадачи индексируются тремя параметрами (истоком, стоком и префиксом вершин). Важно, чтобы внешний цикл индексировался размером k подзадачи, вследствие чего все соответствующие члены $A[k-1][v][w]$ доступны для постоянно-временного поиска на каждой итерации внутреннего цикла (относительный порядок следования второго и третьего циклов не имеет значения). Существует $O(n^3)$ подзадач; указанный алгоритм выполняет $O(1)$-ю работу для каждой из них (в дополнение к $O(n^2)$-й работе вне строенного цикла `for`), поэтому его время выполнения равно $O(n^3)$[1,2]. Из индукции (на k) и правильности

[1] В отличие от большинства алгоритмов с четырьмя графами, алгоритм Флойда—Уоршелла достаточно быстро формирует простую матрицу (где элемент (v, w) матрицы равен ℓ_{vw}, если $(v, w) \in E$ и $+\infty$ в противном случае), применяемую также и для графов, представленных списками смежности.

[2] Поскольку решение проблемы подпроцессов зависит только от предыдущего пакета, алгоритм может быть реализован с использованием пространства $O(n^2)$ (аналогично задаче 17.5).

рекурренции (следствие 18.7) вытекает, что когда входной граф не имеет отрицательного цикла, алгоритм правильно вычисляет кратчайшее расстояние между каждой парой вершин[1].

18.4.4. Обнаружение отрицательного цикла

Если же входной граф имеет отрицательный цикл, то как узнать, можно ли доверять решениям финальной партии подзадач? «Диагональные» элементы массива подзадач будут подсказкой[2].

Лемма 18.8 (обнаружение отрицательного цикла). *Входной граф $G = (V, E)$ имеет отрицательный цикл, если в конце алгоритма* `Floyd-Warshall` *$A[n][v][v] < 0$ для некоторой вершины $v \in V$.*

Доказательство: если входной граф не имеет отрицательного цикла, то: (i) алгоритм `Floyd-Warshall` правильно вычисляет все расстояния кратчайшего пути; и (ii) нет пути их вершины v к самой себе короче пустого пути (который имеет длину 0). Таким образом, в конце алгоритма $A[n][v][v] = 0$ для всех $v \in V$.

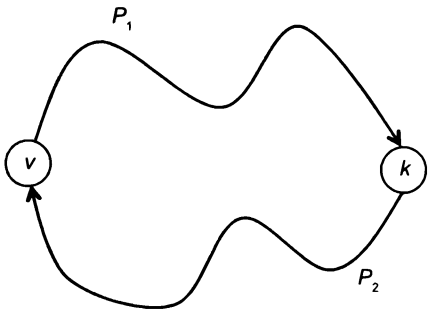
Для того чтобы доказать обратное, предположим, что G имеет отрицательный цикл. Это означает, что G имеет отрицательный цикл без повторяющихся вершин, кроме его начала и конца. (Понимаете почему?) Пусть C обозначает произвольный такой цикл. Алгоритму `Floyd-Warshall` не нужно точно вычислять расстояние кратчайшего пути, но это все равно тот случай, когда $A[k][v][w]$ является не *больше* чем минимальной длиной бесциклового пути v–w, в котором внутренние вершины ограничены $\{1, 2, ..., k\}$ (как вы должны убедиться сами по индукции на k).

Предположим, что вершина k цикла C имеет самую большую метку. Пусть $v \neq k$ будет некоторой другой вершиной C:

---

[1] Демонстрация алгоритма Флойда—Уоршелла до начала обучения динамическому программированию наверняка вызвала бы у читателей восклицания вроде: «Это впечатляюще элегантный алгоритм, который я едва ли способен придумать». Теперь, когда вы кое-чему научились, ваша реакция, скорее всего, будет такой: «Как я мог сам не придумать этот алгоритм?»

[2] О других подходах см. задачу 18.6.



Две стороны P_1 и P_2 цикла являются бесцикловыми путями v–k и k–v, в которых внутренние вершины ограничены $\{1, 2, ..., k-1\}$, поэтому $A[k-1][v][k]$ и $A[k-1][k][v]$ являются не более чем их соответствующими длинами. Таким образом, $A[k][v][v]$, который не больше $A[k-1][v][k] + A[k-1][k][v]$, не больше длины цикла C, которая меньше нуля. Конечное значение $A[n][v][v]$ может быть только меньше. *Ч. Т. Д.*

18.4.5. Резюме и открытые вопросы

Обобщая все, что мы теперь знаем об алгоритме `Floyd-Warshall`:

Теорема 18.9 (свойства алгоритма `Floyd-Warshall`). *Для каждого входного графа $G = (V, E)$ с n вершинами и вещественными длинами ребер алгоритм* `Floyd-Warshall` *работает за время $O(N^3)$ и либо*:

(i) возвращает кратчайшее расстояние между каждой парой $v, w \in V$ вершин; либо

(ii) обнаруживает, что G содержит отрицательный цикл.

Как обычно, самые короткие пути могут быть реконструированы путем обратной трассировки конечного массива A, вычисляемого алгоритмом `Floyd-Warshall`[1].

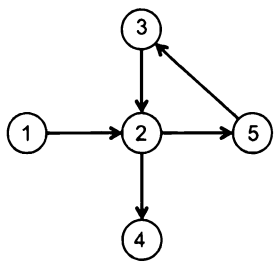
[1] По аналогии с алгоритмом Беллмана—Форда (см. сноску 1 на с. 227), следует поддерживать для каждой пары вершин v, w последний переход пути vw минимальной длины без цикла с внутренними вершинами, ограниченными $\{1, 2, ..., k\}$. В случае повторения по варианту 1 для пары вершин v, w в k-м пакете подзадач последний переход для пары остается неизменным. При повторении по варианту 2 последний

Как мы должны относиться к кубическому времени работы алгоритма Флойда—Уоршелла? Мы не можем ожидать, что время работы будет лучше квадратичного (с квадратичным числом значений для отчета), но существует большой разрыв между кубическим и квадратичным временем работы. При этом открытым остается вопрос о существовании алгоритма для задачи о кратчайшем пути для всех пар на n-вершинных графах, который выполняется, скажем, за время $O(n^{2,99})$[1].

18.4.6. Решения упражнений 18.5–18.6

Решение упражнения 18.5

Правильный ответ: (г). Конкатенация P^* путей P_1^* и P_2 наверняка начинается в v (потому что P_1^* там начинается) и заканчивается в w (так как там заканчивается P_2). Внутренние вершины P^* — такие же, как у P_1^* и P_2 (плюс новая внутренняя вершина k). Все внутренние вершины P_1^* и P_2 принадлежат $\{1, 2, ..., k-1\}$, все внутренние вершины P^* принадлежат $\{1, 2, ..., k\}$. Длина конкатенации двух путей является суммой их длин, поэтому P^* действительно имеет длину $L_1^* + L_2 < L$. Вопрос в том, что конкатенация двух бесцикловых путей не обязательно должна быть без циклов. Например, в графе



переход для v, w переназначается на самый последний прыжок для k, w. Восстановление для данной пары вершин требует $O(n)$ времени.

[1] Мы можем добиться большего, чем это позволяет алгоритм Флойда—Уоршелла, для менее плотных графиков. Так, можно свести проблему кратчайшего пути для всех пар (с отрицательной длиной ребра) только к одному вызову алгоритма Беллмана—Форда, за которым следует $n-1$ вызов алгоритма Дейкстры. Это сокращение (так называемый алгоритм Джонсона) наглядно представлено в видео на www.algorithmsilluminated.org и выполняется за время, выраженное как $O(mn) + (n-1) \times \times O(m \log n) = O(mn \log n)$: субкубическое в n, кроме случаев, когда m предельно близко к квадратичному в n.

конкатенирование пути $1 \to 2 \to 5$ с путем $5 \to 3 \to 2 \to 4$ создает путь, содержащий ориентированный цикл $2 \to 5 \to 3 \to 2$.

Решение упражнения 18.6

Правильный ответ: (в). Если $v = w$, то единственным путем v–w без внутренних вершин является пустой путь (длиной 0). Если $(v, w) \in E$, то единственным таким путем является путь с одним переходом $v \to w$ (длиной ℓ_{vw}). Если $v \neq w$ и $(v, w) \notin E$, то нет путей v–w без внутренних вершин и $L_{0,v,w} = +\infty$.

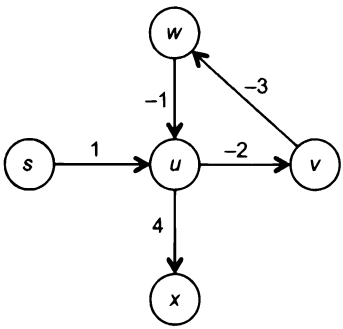
ВЫВОДЫ

- Совсем не очевидно, как определять кратчайший путь расстояния в графе с отрицательным циклом.
- В задаче о кратчайшем пути с единственным истоком вход состоит из ориентированного графа с длинами ребер и истоковой вершины. Цель состоит в том, чтобы вычислить длину кратчайшего пути от истоковой вершины до любой другой вершины либо обнаружить, что граф имеет отрицательный цикл.
- Алгоритм Беллмана—Форда — это алгоритм динамического программирования, который решает задачу о кратчайшем пути с единственным истоком за время $O(mn)$, где m и n — это соответственно число ребер и вершин входного графа.
- Ключевая идея в алгоритме Беллмана—Форда состоит в параметризации подзадачи с помощью реберного бюджета i (в дополнение к пункту назначения, то есть стоку) и рассмотрении только путей с i или меньшим числом ребер.
- Алгоритм Беллмана—Форда сыграл видную роль в эволюции протоколов маршрутизации интернета.
- В задаче о кратчайшем пути для всех пар вход состоит из ориентированного графа с длинами ребер. Цель состоит в том, чтобы либо вычислить длину кратчайшего пути от каждой вершины до каждой другой вершины, либо обнаружить, что граф имеет отрицательный цикл.

- Алгоритм Флойда—Уоршелла — это алгоритм динамического программирования, который решает задачу о кратчайшем пути для всех пар за время $O(n^3)$, где n — это число вершин входного графа.
- Ключевая идея алгоритма Флойда—Уоршелла состоит в параметризации подзадач с префиксом k вершин (в дополнение к истоку и стоку) и рассмотрении только бесцикловых путей, в которых все внутренние вершины находятся в $\{1, 2, ..., k\}$.

Задачи на закрепление материала

Задача 18.1. Определите для входного графа



окончательные элементы массива алгоритма `Bellman-Ford` из раздела 18.2.

Задача 18.2. Лемма 18.3 показывает, что как только решения подзадачи стабилизируются в алгоритме Беллмана—Форда (при $L_{k+1,v} = L_{k,v}$ для каждого пункта назначения v), они остаются неизменными навсегда (при $L_{i,v} = L_{k,v}$ для всех $i \geq k$ и $v \in V$). Является ли это также верным для каждой вершины? То есть верно ли, что всякий раз, когда $L_{k+1,v} = L_{k,v}$ для некоторого $k \geq 0$ и пункта назначения v, $L_{i,v} = L_{k,v}$ для всех $i \geq k$? Приведите доказательство либо контрпример.

Задача 18.3. Рассмотрим ориентированный граф $G = (V, E)$ с n вершинами, m ребрами, истоковой вершиной $s \in v$, вещественными длинами ребер и без отрицательных циклов. Предположим, вы знаете, что каждый кратчайший

путь в G из s до другой вершины имеет не более k ребер. Как быстро вы сможете решить задачу о кратчайшем пути с единственным истоком? Выберите истинное утверждение:

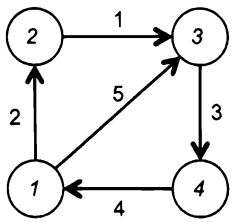
а) $O(m + n)$

б) $O(kn)$

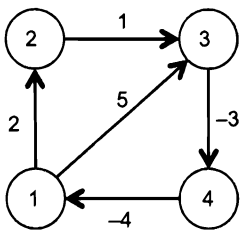
в) $O(km)$

г) $O(mn)$

Задача 18.4. Определите для входного графа окончательные элементы массива алгоритма `Floyd-Warshall` из раздела 18.4.



Задача 18.5. Определите для входного графа окончательные элементы массива алгоритма `Floyd-Warshall`.



Задачи повышенной сложности

Задача 18.6. Алгоритм `Floyd-Warshall` выполняется за время $O(n^3)$ на графах с n вершинами и m ребрами, независимо от того, содержит ли входной

граф отрицательный цикл или нет. Измените алгоритм так, чтобы он решал задачу о кратчайшем пути для всех пар за время $O(mn)$ для входных графов с отрицательным циклом и за время $O(n^3)$ в противном случае.

Задача 18.7. Какая из следующих ниже задач может быть решена за время $O(n^3)$, где n — число вершин во входном графе?

а) С учетом ориентированного графа $G = (V, E)$ с неотрицательными длинами ребер требуется вычислить максимальную длину кратчайшего пути между любой парой вершин (то есть $\max_{v,w \in V} dist(v, w)$);

б) С учетом ориентированного ациклического графа с вещественными длинами ребер требуется вычислить длину самого длинного пути между любой парой вершин;

в) С учетом ориентированного графа с неотрицательными длинами ребер требуется вычислить длину самого длинного бесциклового пути между любой парой вершин;

г) С учетом ориентированного графа с вещественными длинами ребер требуется вычислить длину самого длинного бесциклового пути между любой парой вершин.

Задачи по программированию

Задача 18.8. Реализуйте на вашем любимом языке программирования алгоритмы `Bellman-Ford` и `Floyd-Warshall`. В случае задачи о кратчайшем пути для всех пар насколько быстрее алгоритм Флойда—Уоршелла по сравнению с алгоритмом Беллмана—Форда? Для получения бонусных баллов реализуйте алгоритмы оптимизации пространства и линейно-временной реконструкции, описанные в сносках 1 на с. 227, 2 на с. 238, 1 на с. 240. Тестовые случаи и наборы данных для сложных задач см. на веб-сайте www.algorithmsilluminated.org.

Эпилог: руководство по разработке алгоритмов

Завершив знакомство с тремя частями книги по разработке алгоритмов, вы обладаете богатым алгоритмическим арсеналом, подходящим для решения широкого спектра вычислительных задач. Применяя его на практике, вы, возможно, обнаружите пугающе огромное количество алгоритмов, структур данных и парадигм проектирования. Как наиболее эффективным образом задействовать свои инструменты при столкновении с новой задачей? Вот один из проверенных рецептов, представленный для удобства усвоения по шагам (одновременно автор призывает каждого из читателей разработать свой собственный рецепт, основанный на их личном опыте).

1. Начните с ответа на вопрос, можно ли избежать решения задачи с нуля. Является ли она замаскированной версией, вариантом или частным случаем задачи, которую вы уже знаете, как решать? Например, можно ли свести ее к сортировке, поиску в графе или вычислению кратчайшего пути[1]? Если да, то используйте самый быстрый алгоритм, достаточный для решения задачи.

2. Оцените, можно ли упростить задачу, обрабатывая входные данные с помощью бесплатного примитива, такого как сортировка или вычисление связных компонент.

3. Если требуется разработать новый алгоритм с нуля, откалибровать, выявляя линию на песке, нарисованную «очевидным» решением (на-

[1] В процессе углубленного изучения алгоритмов, выходящего за рамки этой серии книг, могут появиться не столь очевидные проблемы. В качестве примеров укажем на быстрое преобразование Фурье, проблемы максимального потока и минимального разреза, двустороннее согласование, линейное и выпуклое программирование.

пример, исчерпывающим поиском), то является ли время выполнения очевидного решения достаточным?

4. Если очевидного решения недостаточно, проведите мозговой штурм как можно большего числа естественных жадных алгоритмов и протестируйте их на небольших примерах. Скорее всего, все окажется безуспешным. Но то, вследствие чего они окажутся безуспешными, поможет вам лучше понять задачу.

5. Если есть естественный способ разбить задачу на более мелкие подзадачи, оцените, насколько легко будет совместить их решения для решения первоначальной задачи. Если вы видите, как сделать это эффективно, то переходите к парадигме «разделяй и властвуй».

6. Попробуйте динамическое программирование. Можете ли вы утверждать, что решение должно быть построено из решений меньших подзадач одним из немногих способов? Можете ли вы сформулировать рекуррентное соотношение, для того чтобы быстро решить подзадачу с учетом решений скромного числа меньших подзадач?

7. В том случае, когда вы разрабатываете хороший алгоритм для задачи, можете ли вы сделать его еще лучше, развернув правильные структуры данных? Отыскивайте места, где ваш алгоритм выполняет значительные вычисления снова и снова (например, поиск или минимальные вычисления). Помните о принципе бережливости: выбирайте простейшую структуру данных, которая поддерживает все операции, требуемые вашим алгоритмом.

8. Можете ли вы сделать свой алгоритм проще или быстрее с помощью рандомизации? Например, если алгоритм должен выбирать один объект из многих, то что происходит, когда он выбирает его случайным образом?

9. Если все предыдущие шаги в итоге оказываются безуспешными, то подумайте о неудачной, но распространенной возможности того, что для вашей задачи нет эффективного алгоритма. Можете ли вы доказать, что ваша задача является вычислительно неразрешимой, сведя к ней известную NP-трудную задачу (подробности в *части 4*)?

10. Снова рассмотрите парадигму проектирования алгоритмов, на этот раз в поисках возможности для быстрой эвристики (особенно с жадными алгоритмами) и более точных алгоритмов, чем исчерпывающий поиск, в особенности с динамическим программированием (подробности в *части 4*).

Подсказки и решения избранных задач

Подсказка к задаче 13.1: один из жадных алгоритмов может быть доказан как правильный с помощью обмена аргументами, аналогичного приведенному в разделе 13.4.

Подсказка к задаче 13.2: для каждого неправильного алгоритма существует контрпример только с двумя заданиями.

Подсказка к задаче 13.3: пусть S_i обозначает множество заданий с i самыми ранними сроками завершения. Докажите по индукции на i, что выбранный вами жадный алгоритм выбирает максимально возможное число неконфликтующих заданий из S_i.

Решение задачи 14.1: (а). Достигается, например, с помощью кода

Символ	Кодирование
A	00
B	01
C	10
D	110
E	111

Решение задачи 14.2: (а). Достигается, например, с помощью кода

Символ	Кодирование
A	110
B	1110
C	0
D	1111
E	10

Подсказка к задаче 14.3: для нижней границы рассмотрите частоты символов, которые являются степенями 2.

Подсказка к задаче 14.4: в случае (в) докажите, что буква с частотой менее 0,33 участвует по крайней мере в одном слиянии до окончательной итерации. В случае (г) см. задачу 14.2.

Решение задачи 14.5: отсортировать символы по частоте и вставить их в порядке возрастания в очередь Q_1.[1] Инициализировать пустую очередь Q_2. Поддерживать следующие инварианты: (i) элементы Q_1 соответствуют одноузловым деревьям в текущем лесу F, хранящимся в порядке возрастания частоты; (ii) элементы Q_2 соответствуют многоузловым деревьям F, хранящимся в порядке возрастания суммы символьных частот. На каждой итерации алгоритма деревья T_1 и T_2 с наименьшими суммами символьных частот могут быть выявлены и удалены с помощью постоянного числа операций в головных частях Q_1 и Q_2. Слияние T_3 деревьев T_1 и T_2 вставляется в конце Q_2. Дополнительное упражнение: задайтесь вопросом, почему инвариант (ii) продолжает соблюдаться? Каждая операция очереди (удаление спереди или добавление сзади) выполняется за время $O(1)$, поэтому суммарное время выполнения $n - 1$ итераций главного цикла равно $O(n)$.

Подсказка к задаче 15.1: чтобы рассуждать о T, используйте следствие 15.8 или свойство минимального узкого места (с. 100). Чтобы рассуждать о P, подумайте о двух путях s–t с разным числом ребер.

Подсказка к задаче 15.2: примените лемму 15.7, для того чтобы доказать, что выход является остовным деревом. Докажите, что каждое ребро, которое не удовлетворяет свойству минимального узкого места (с. 100), исключено из конечного выхода, и используйте теорему 15.6.

Подсказка к задаче 15.3: три из четырех задач легко сводятся к задаче о минимальном остовном дереве. Для одного из них используйте тот факт, что $\ln(x \times y) = \ln x + \ln y$ для $x, y > 0$.

---

[1] Скажем, об очередях можно почитать в вашей любимой вводной книге по программированию (или в «Википедии»). Ведь очередь — это структура данных для ведения списка объектов, и вы можете удалить что-то из начала и добавить в конец. Один из способов реализации очереди — двусвязный список.

Решение задачи 15.4: предположим, что ребро $e = (v, w)$ минимального остовного дерева T графа G не удовлетворяет свойству минимального узкого места, и пусть P обозначает путь v–w в G, в котором каждое ребро стоило меньше, чем c_e. Удаление e из T создает две связные компоненты, S_1 (содержащую v) и S_2 (содержащую w). v–w-путь P включает ребро $e' = (x, y)$ с $x \in S_1$ и $y \in S_2$. Множество ребер $T' = T - \{e\} \cup \{e'\}$ является остовным деревом с суммарной стоимостью меньше, чем у T, что противоречит допущению о том, что T является минимальным остовным деревом.

Решение задачи 15.5: приведем аргумент для алгоритма Краскала (аналогичен аргументу для алгоритма Прима). Пусть $G = (V, E)$ равно связному неориентированному графу с вещественными реберными стоимостями, причем не обязательно разными. Допустим, что не все ребра имеют одинаковую стоимость и, в более общем случае, не все остовные деревья имеют одинаковую суммарную стоимость (почему?). Пусть δ_1 обозначает наименьшую строго положительную разницу между стоимостями двух ребер, M^* — стоимость минимального остовного дерева графа G, а M — минимальную стоимость неоптимального остовного дерева графа G. Определим δ_2 как $M - M^*$ и $\delta = \min\{\delta_1, \delta_2\} > 0$. Пусть e_i обозначает i-е ребро G, рассматриваемое алгоритмом Краскала (после произвольного разрыва совпадений значений на шаге сортировочной предобработки). Получим новый граф G' из G путем увеличения стоимости каждого ребра e_i из c_{e_i} до $c'_{e_i} = c_{e_i} + \delta/2^{(m-i+1)}$, где m — это число ребер. Стоимость каждого остовного дерева может только увеличиваться, причем не более чем на $\delta \cdot \sum_{i=1}^{m} 2^{(m-i+1)} = \delta \cdot \sum_{i=1}^{m} 2^{-i} < \delta$. Поскольку $\delta \leq \delta_2$, минимальное остовное T из G' также должно быть одним из G. Поскольку $\delta \leq \delta_1$, ребра G' имеют разные стоимости, где ребро e_j является i-м самым дешевым ребром G'. Алгоритм `Kruskal` исследует ребра G и G' в том же порядке и, следовательно, выводит одно и то же остовное дерево T^* в обоих случаях. Из доказательства правильности алгоритма `Kruskal` для графов с разными реберными стоимостями известно, что T^* является минимальным остовным деревом графа G', а следовательно, G тоже.

Подсказка к задаче 15.6: следуйте доказательству теоремы 15.6.

Решение задачи 15.7: в случае (а) предположим от противного, что существует минимальное остовное T, которое исключает $e = (v, w)$. Будучи остовным деревом, T содержит v–w-путь P. Поскольку v и w находятся на разных сторонах разреза (A, B), P включает ребро e', которое пересекает (A, B). По принятому допущению стоимость e' превышает стоимость e. Таким образом,

$T' = T \cup \{e\} - \{e'\}$ является остовным деревом со стоимостью меньше, чем у T, что следует рассматривать как противоречие. В случае (б) каждая итерация алгоритма Прима выбирает самое дешевое ребро e, пересекающее разрез $(X, V-X)$, где X — множество вершин, охватываемых текущим решением. Из свойства разреза вытекает, что каждое минимальное остовное дерево содержит каждое ребро конечного остовного дерева T алгоритма, и поэтому T является уникальным минимальным остовным деревом. В случае (в) сходным образом каждое ребро, выбранное алгоритмом Краскала, оправдывается свойством разреза. Каждое ребро $e = (v, w)$, добавленное алгоритмом, является самым дешевым, в котором конечные точки находятся в разных связных компонентах текущего решения (поскольку это именно те ребра, добавление которых не создаст цикл). В частности, e является самым дешевым ребром, пересекающим разрез (A, B), где A — это текущая связная компонента v, а $B = V - A$ — все остальные.

Подсказка к задаче 15.8: в случае (а) высокоуровневая идея заключается в выполнении бинарного поиска остовного дерева с минимальным узким местом (MBST). Вычислите медианную стоимость ребра во входном графе G (о том, как это сделать за линейное время, см. главу 6 *части 1*). Возьмите G' из G, отбросив все ребра со стоимостью выше медианы. Продолжайте, выполняя рекурсию на графе с половиной такого числа ребер G. Простейший случай — G является связным; но как выполнять рекурсию, если G не является связным? Для анализа времени выполнения используйте индукцию либо случай 2 основного метода (см. главу 4 *части 1*). Что касается пункта (б), то ответ, как представляется, будет отрицательным. Каждое минимальное остовное дерево является остовным деревом с минимальным узким местом (MBST), но не наоборот, как вы должны убедиться сами. Вопрос о том, существует ли детерминированный линейно-временной алгоритм для задачи о минимальном остовном дереве, остается открытым по сей день; полное изложение см. в бонусном видео по адресу www.algorithmsilluminated.org.

Решение задачи 16.1:

0	5	5	6	12	12	16	18

и первая, и четвертая, и седьмая вершины.

Подсказка к задаче 16.2: в случае (а) и (в) вернитесь к примеру с четырьмя вершинами на с. 143. В случае (г) используйте индукцию и лемму 16.1.

Подсказка к задаче 16.3: если G является деревом, укорените его в произвольной вершине и определите одну подзадачу для каждого поддерева. В случае произвольного графа G какими могут быть подзадачи?

Решение задачи 16.4: со столбцами, индексированными i, и строками c:

9	0	1	3	6	8	10
8	0	1	3	6	8	9
7	0	1	3	6	7	9
6	0	1	3	6	6	8
5	0	1	3	5	5	6
4	0	1	3	4	4	5
3	0	1	2	4	4	4
2	0	1	1	3	3	3
1	0	1	1	1	1	1
0	0	0	0	0	0	0
	0	1	2	3	4	5

То же — для второй, третьей и пятой позиций.

Подсказка к задаче 16.5: в случае (б) и (в) добавьте третий параметр к решению динамического программирования к первоначальной задаче о ранце в разделе 16.5. Как в случае (г) обобщение вашего решения (в) масштабируется вместе с числом m ранцев?

Решение задачи 17.1: со столбцами, индексированными i, и строками, индексированными j:

6	6	5	4	5	4	5	4
5	5	4	5	4	3	4	5
4	4	3	4	3	4	5	6
3	3	2	3	4	3	4	5
2	2	1	2	3	4	3	4
1	1	0	1	2	3	4	5
0	0	1	2	3	4	5	6
	0	1	2	3	4	5	6

Подсказка к задаче 17.2: на каждой итерации цикла были ли уже вычислены необходимые решения подзадачи на предыдущих итерациях (или в качестве базового случая)?

Решение задачи 17.3: задачи в (б) и (г) могут быть решены с использованием алгоритмов, аналогичных `NW`, с одной подзадачей для каждой пары X_i, Y_j префиксов входных символьных цепочек. В качестве альтернативы задача в (б) сводится к задаче о выравнивании последовательностей путем установки штрафа с шагом, равным 1, и штрафа за сочетание двух разных символов с очень большим числом.

Задача в (а) может быть решена путем обобщения алгоритма `NW`, отслеживающего, является ли вставленный зазор первым в последовательности зазоров (в этом случае он несет штраф $a + b$) или нет (в этом случае дополнительный штраф равен a). Для каждой пары префиксов входных цепочек вычислите суммарный штраф трех выравниваний: лучшее без зазоров в последнем столбце, лучшее с зазором в верхней строке последнего столбца и лучшее с зазором в нижней строке последнего столбца. Число подзадач и работа над каждой подзадачей увеличиваются с постоянным коэффициентом.

Задача (в) может быть решена эффективно без использования динамического программирования; просто подсчитайте частоту каждого символа в каждой цепочке. Перестановка f существует тогда и только тогда, когда каждый символ встречается ровно столько же раз в каждой строке (поясните почему).

Решение задачи 17.4: со столбцами, индексированными i, и строками, индексированными $j = i + s$:

7	223	158	143	99	74	31	25	0
6	151	105	90	46	26	3	0	
5	142	97	84	40	20	0		
4	92	47	37	10	0			
3	69	27	17	0				
2	30	5	0					
1	20	0						
0	0							
	1	2	3	4	5	6	7	8

Подсказка к задаче 17.5: идея — в повторном использовании пространства, после того как решение подзадачи становится неактуальным для будущих вычислений. Для выполнения всех вычислений алгоритм `WIS` должен помнить только две самые последние подзадачи. Алгоритм `NW` должен помнить решения подзадач для текущего и предыдущего значений i, а также для всех значений j (почему?). А как насчет алгоритма `OptBST`?

Подсказка к задаче 17.6: не трудитесь решать подзадачи для префиксов X_i и Y_j при $|i-j|>k$.

Подсказка к задаче 17.7: время работы вашего алгоритма должно быть ограничено полиномиальной функцией от n — очень и очень большого полинома!

Решение задачи 18.1: со столбцами, индексированными i, и строками, индексированными вершинами:

x	+∞	+∞	5	5	5	1
w	+∞	+∞	+∞	–4	–4	–4
v	+∞	+∞	–1	–1	–1	–7
u	+∞	1	1	1	–5	–5
s	0	0	0	0	0	0
	0	1	2	3	4	5

Решение задачи 18.2: нет. Контрпример см. в предыдущей задаче.

Подсказка к задаче 18.3: рассмотрите возможность досрочной остановки алгоритма кратчайшего пути.

Решение задачи 18.4: со столбцами, индексированными k, и строками, индексированными парами вершин:

(1 , 1)	0	0	0	0	0
(1 , 2)	2	2	2	2	2
(1 , 3)	5	5	3	3	3
(1 , 4)	+∞	+∞	+∞	6	6
(2 , 1)	+∞	+∞	+∞	+∞	8
(2 , 2)	0	0	0	0	0
(2 , 3)	1	1	1	1	1
(2 , 4)	+∞	+∞	+∞	4	4
(3 , 1)	+∞	+∞	+∞	+∞	7
(3 , 2)	+∞	+∞	+∞	+∞	9
(3 , 3)	0	0	0	0	0
(3 , 4)	3	3	3	3	3
(4 , 1)	4	4	4	4	4
(4 , 2)	+∞	6	6	6	6
(4 , 3)	+∞	9	7	7	7
(4 , 4)	0	0	0	0	0
	0	1	2	3	4

Решение задачи 18.5: со столбцами, индексированными k, и строками, индексированными парами вершин:

(1 , 1)	0	0	0	0	–4
(1 , 2)	2	2	2	2	–2
(1 , 3)	5	5	3	3	–1
(1 , 4)	+∞	+∞	+∞	0	–4
(2 , 1)	+∞	+∞	+∞	+∞	–6
(2 , 2)	0	0	0	0	–4
(2 , 3)	1	1	1	1	–3
(2 , 4)	+∞	+∞	+∞	–2	–6
(3 , 1)	+∞	+∞	+∞	+∞	–7
(3 , 2)	+∞	+∞	+∞	+∞	–5
(3 , 3)	0	0	0	0	–4
(3 , 4)	–3	–3	–3	–3	–7
(4 , 1)	–4	–4	–4	–4	–8
(4 , 2)	+∞	–2	–2	–2	–6
(4 , 3)	+∞	1	–1	–1	–5
(4 , 4)	0	0	0	–4	–8
	0	1	2	3	4

Решение задачи 18.6: модифицируйте входной граф $G = (V, E)$, добавив новую истоковую вершину s и новое ребро нулевой длины из s в каждую вершину $v \in V$. Новый граф G имеет отрицательный цикл, достижимый из s, только если G имеет отрицательный цикл. Выполните алгоритм Беллмана—Форда на G с истоковой вершиной s, чтобы проверить, содержит ли G отрицательный цикл. Если нет, то выполните алгоритм Флойда—Уоршелла на G.

Подсказка к задаче 18.7: задачи с самым длинным путем могут быть переформулированы как задачи с самым коротким путем после умножения всех длин ребер на –1. Вспомните задачу вычисления кратчайших бесцикловых путей в графах с отрицательными циклами и тот факт, что она, по-видимому, не допускает полиномиально-временного алгоритма. Имеет ли этот факт какое-либо значение для любой из четырех рассмотренных задач?

Тим Рафгарден

Совершенный алгоритм. Жадные алгоритмы и динамическое программирование

Перевел с английского *А. Логунов*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*О. Букатка*
Художник	*В. Мостипан*
Корректоры	*С. Беляева, М. Молчанова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01

Подписано в печать 24.01.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 20,640. Тираж 1200. Заказ 851.

Отпечатано в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8(499)270-73-59

Тим Рафгарден *— профессор Computer Science и Management Science and Engineering в Стэнфордском университете. Он изучает связи между информатикой и экономикой и занимается задачами разработки, анализа, приложений и ограничений алгоритмов. Среди его многочисленных наград — премии Калая (2016), Гёделя (2012) и Грейс Мюррей Хоппер (2009).*

Алгоритмы — это сердце и душа computer science. Без них не обойтись, они есть везде — от сетевой маршрутизации и расчетов по геномике до криптографии и машинного обучения. «Совершенный алгоритм» превратит вас в настоящего профи, который будет ставить задачи и мастерски их решать как в жизни, так и на собеседовании при приеме на работу в любую IT-компанию.

В новой книге Тим Рафгарден расскажет о жадных алгоритмах (задача планирования, минимальные остовные деревья, кластеризация, коды Хаффмана) и динамическом программировании (задача о рюкзаке, выравнивание последовательностей, кратчайшие пути, оптимальные деревья поиска).

Серия книг «Совершенный алгоритм» адресована тем, у кого уже есть опыт программирования, и основана на онлайн-курсах, которые регулярно проводятся с 2012 года. Вы перейдете на новый уровень, чтобы увидеть общую картину, разобраться в низкоуровневых концепциях и математических нюансах.

Познакомиться с дополнительными материалами и видеороликами автора (на английском языке) можно на сайте **www.algorithmsilluminated.org**

COMPUTER SCIENCE

ISBN:978-5-4461-1445-0

ПИТЕР®

Заказ книг:
тел.: (812) 703-73-74
books@piter.com

WWW.PITER.COM
каталог книг и интернет-магазин

instagram.com/**piterbooks**
youtube.com/**ThePiterBooks**
vk.com/**piterbooks**
facebook.com/**piterbooks**