ТИМ РАФГАРДЕН

СОВЕРШЕННЫЙ АЛГОРИТМ

АЛГОРИТМЫ ДЛЯ NP-ТРУДНЫХ ЗАДАЧ

COMPUTER SCIENCE

Algorithms Illuminated

Part 4: Algorithms for NP-Hard Problems

Tim Roughgarden

ТИМ РАФГАРДЕН

СОВЕРШЕННЫЙ АЛГОРИТМ

АЛГОРИТМЫ ДЛЯ NP-ТРУДНЫХ ЗАДАЧ

COMPUTER SCIENCE

ПИТЕР®

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

2021

ББК 32.973.2-018
УДК 004.42
Р26

Рафгарден Тим

Р26 Совершенный алгоритм. Алгоритмы для NP-трудных задач. — СПб.: Питер, 2021. — 304 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1799-4

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.

Если вы уже достаточно прокачались в асимптотическом анализе, жадных алгоритмах и динамическом программировании, самое время рассмотреть понятие NP-трудности, которое часто вызывает неподдельный страх. Тим Рафгарден покажет, как распознать NP-трудную задачу, расскажет, как избежать решения с нуля, и поможет найти эффективные пути решения.

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.

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).

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

ББК 32.973.2-018
УДК 004.42

ISBN 978-0999282960 англ.
ISBN 978-5-4461-1799-4

Оглавление

Предисловие

Перед вами четвертая из серии книг, написанных на базе проводимых мною с 2012 года онлайн-курсов по алгоритмам. Эти курсы, в свою очередь, появились благодаря лекциям для студентов, которые я читал в Стэнфордском университете в течение многих лет. *Четвертая часть* исходит из того, что читатель уже немного знаком с асимптотическим анализом и *O*-большим, поиском в графах и алгоритмами кратчайшего пути, жадными алгоритмами и динамическим программированием (все эти темы освещены в предыдущих частях).

О чем эти книги

Четвертая часть серии «Совершенный алгоритм» посвящена NP-трудным[1] задачам и работе с ними.

Алгоритмические инструменты для решения NP-трудных задач. Многие реальные задачи являются NP-трудными и кажутся не поддающимися решению теми типами всегда правильных и всегда быстрых алгоритмов, которые были представлены в предыдущих частях. При встрече с NP-трудной

[1] Класс *NP-трудных задач* (NP, nondeterministic polynomial time — недетерминированный полиномиально-временной) — это класс сложности, используемый для классификации задач принятия решений. NP — это множество задач, для которых экземпляры задач с ответом «да» имеют доказательства, проверяемые за полиномиальное время детерминированной машиной Тьюринга. — *Примеч. пер.*

задачей придется пойти на компромисс между правильностью и скоростью. Вы увидите старые методы (жадные алгоритмы) и новые (локальный поиск) для разработки быстрых «приближенно правильных» эвристических алгоритмов для работы с приложениями по задачам планирования, максимизации влияния в социальных сетях и задаче коммивояжера. Мы также рассмотрим старые методы (динамическое программирование) и новые (решатели задач смешанного целочисленного программирования и задач выполнимости булевых формул) для улучшения работы алгоритмов исчерпывающего поиска. Приложения будут включать задачу коммивояжера, поиск сигнальных путей в биологических сетях и переупаковку телевизионных станций на аукционе радиочастотного спектра в США.

Распознавание NP-трудных задач. Эта книга научит вас быстро распознавать NP-трудную задачу и не тратить время на разработку идеального алгоритма для ее решения. Вы познакомитесь с многочисленными широко известными и базовыми NP-трудными задачами, начиная от задач выполнимости и раскраски графов и заканчивая задачей о гамильтоновом пути. Вы попробуете доказать NP-трудность с помощью редукций.

Для более подробного ознакомления с содержанием книги загляните в разделы «Выводы», где выделены наиболее важные моменты. «Полевое руководство по разработке алгоритмов» на с. 283 даст представление о том, как темы этой книги вписываются в общую алгоритмическую картину.

Разделы книги, помеченные звездочками, — самые продвинутые. Читатели, испытывающие нехватку времени, могут пропустить их при первом чтении без потери непрерывности.

Темы, затронутые в первых трех частях. *Первая часть серии «Совершенный алгоритм»* охватывает асимптотические обозначения (*O*-большое и его близких родственников), алгоритмы «разделяй и властвуй» и основной метод, рандомизированные алгоритмы, быструю сортировку и ее анализ, а также линейно-временные алгоритмы отбора. Во *второй части* рассмотрены различные структуры данных (кучи, сбалансированные деревья поиска, хеш-таблицы, фильтры Блума), графовые примитивы (поиск сначала в ширину и сначала в глубину, связность, кратчайшие пути) и области их применения (от дедупликации до анализа социальных сетей). *Третья часть* посвящена жадным алгоритмам (задаче планирования, определению минимального

остовного дерева графа, кластеризации, кодам Хаффмана), а также динамическому программированию (задаче о рюкзаке, выравниванию рядов, поиску кратчайших путей, построению деревьев оптимального поиска).

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 несколько полезных структур для организации данных, которые можете применять непосредственно в ваших собственных программах. Реализация и применение этих алгоритмов расширят и улучшат ваши навыки программирования. Вы также узнаете основные приемы разработки алгоритмов, которые актуальны для решения разнообразных задач в широких областях, получите инструменты для прогнозирования производительности этих алгоритмов. Такие шаблоны могут быть вам полезны для разработки новых алгоритмов решения задач, которые возникают в вашей собственной работе.

Развитие аналитических способностей. Алгоритмические описания, мыслительная работа над алгоритмами дают большой опыт. Посредством математического анализа вы получите углубленное понимание конкретных алгоритмов и структур данных, описанных в этой книге. Вы приобретете навыки работы с несколькими математическими методами, которые широко применяются для анализа алгоритмов.

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

Знакомство с величайшими достижениями информатики. Изучение алгоритмов напоминает просмотр эффектного клипа с многочисленными суперхитами минувших шестидесяти лет развития информатики. Вы больше

не будете чувствовать себя чужим на фуршете для специалистов в области computer science, когда кто-то отпустит шутку по поводу алгоритма Дейкстры. Прочитав эти книги, вы будете точно знать, что он имеет в виду.

Успешность при прохождении собеседования. На протяжении многих лет бесчисленные студенты развлекали меня рассказами о том, как знания, почерпнутые из этих книг, позволяли им успешно справляться с любым техническим вопросом, который им задавали во время собеседования.

В чем особенность этой книги

Эта книга предназначена только для одного: *постараться научить основам алгоритмизации максимально доступным способом*. Воспринимайте ее как конспект лекций, которые опытный наставник по алгоритмам будет давать вам на протяжении серии индивидуальных уроков.

Существует ряд прекрасных, гораздо более традиционных и энциклопедически выверенных учебников по алгоритмам. Любой из них с пользой украсит эту серию книг дополнительными деталями, задачами и темами. Хотелось бы, чтобы вы поискали и нашли что-то полезное среди этих книг для себя. Кроме того, есть книги, которые ориентируются на программистов, ищущих готовые реализации алгоритмов на конкретном языке программирования. Множество соответствующих примеров также находятся в свободном доступе в интернете.

Для кого эта книга?

Весь смысл этой книги, как и онлайн-курсов, на основе которых она создана, — быть широко- и легкодоступной настолько, насколько это возможно. На моих онлайн-курсах широко представлены люди всех возрастов, профессионального опыта и слоев общества, есть немало учащихся и студентов, разработчиков программного обеспечения (как состоявшихся, так и начинающих), ученых и профессионалов со всех уголков мира.

Эта книга не является введением в программирование, и было бы просто идеально, если бы вы уже обладали основными навыками программирования

на каком-либо распространенном языке (например, Java, Python, C, Scala, Haskell). Если вам требуется развить свои навыки программирования, то для этих целей есть несколько прекрасных бесплатных онлайн-курсов, обучающих основам программирования.

По мере необходимости мы также используем математический анализ, чтобы разобраться в том, как и почему алгоритмы действительно работают. Свободно доступные конспекты лекций «Математика для Computer Science» Эрика Лемана и Тома Лейтона являются превосходным и освежающим память пособием по системе математических обозначений (например, $\sum$ и $\forall$), основам теории доказательств (метод индукции, доказательство от противного и др.), дискретному распределению вероятностей и многому другому.[1]

Дополнительные ресурсы

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

Видео. Если вы больше настроены смотреть и слушать, чем читать, обратитесь к материалам с «Ютьюба», доступным на сайте www.algorithmsilluminated.org. Эти видео затрагивают все темы этой серии книг. Надеюсь, что они пропитаны тем же заразительным энтузиазмом в отношении алгоритмов, который, увы, невозможно полностью воспроизвести на печатной странице.

Тестовые задания. Как узнать, 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 об ответе, прежде чем читать далее.

Задачи в конце главы. В конце каждой главы вы найдете несколько относительно простых вопросов для проверки усвоения материала (*S*), а затем более трудные и менее ограниченные по времени сложные задачи (*H*). Подсказки

[1] См. Mathematics for Computer Science, Eric Lehman, Tom Leighton: http://www.boazbarak.org/cs121/LehmanLeighton.pdf.

или решения всех этих задач, отмеченных соответственно знаками (*H*) и (*S*), приводятся в конце книги. Читатели могут взаимодействовать со мной и друг с другом по поводу оставшихся задач в конце главы 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 www.algorithmsilluminated.org.

Форумы. Существенной причиной успеха онлайн-курсов являются реализованные через дискуссионные форумы возможности общения для слушателей. Это позволяет им помогать друг другу лучше понимать материал курса, а также отлаживать свои программы. 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 тысяч 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 на более ранний проект этой книги, среди них: Тоня Бласт, Юань Цао, Лесли Дэймон, Тайлер Дэ Девлин, Роман Гафитню, Бланка Хуэрго, Карлос Гуйя, Джим Хумельсин, Тим Кернс, Владимир Кокшенев, Байрам Кулиев, Клейтон Вонг, Лексин Йе и Даниэль Зингаро. Также благодарю нескольких экспертов, которые предоставили техническую консультацию: Амира Аббуда, Винсента Коницера, Кристиана Коницера, Авиада Рубинштайна и Илайю Сигала.

Я всегда ценю предложения и исправления от читателей. О них лучше всего сообщать через дискуссионные форумы, описанные выше.

Тим Рафгарден
Нью-Йорк, штат Нью-Йорк
Апрель 2020

От издательства

Не удивляйтесь, что эта книга начинается с девятнадцатой главы. С одной стороны, она является четвертой частью курса «Совершенный алгоритм» Тима Рафгардена, а с другой — самостоятельным изданием, в котором рассматриваются вопросы NP-трудных задач. 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.

19 Что такое NP-трудность?

Вводные книги по алгоритмам, включая предыдущие части этой серии книг, страдают от предвзятости отбора. Они сосредоточены на вычислительных задачах, которые поддаются решению умными быстрыми алгоритмами — в конце концов, что может быть лучше гениального алгоритмического пути напрямик? Хорошая новость: к таким задачам относятся многие фундаментальные и практически значимые задачи (сортировка, графовый поиск, кратчайшие пути, коды Хаффмана, минимальные остовные деревья, выравнивание рядов и т. д.). Но было бы нечестно обучать вас только одной коллекции задач, игнорируя призрак труднорешаемости,[1] который появляется возле серьезных проектов. К сожалению, вы встретите много важных вычислительных задач, которые о быстрых алгоритмах не слышали и, по общему мнению специалистов, в ближайшее время не услышат.

Эта суровая реальность вызывает два вопроса. Во-первых, как распознать сложные задачи в своей работе и как на них реагировать, чтобы не тратить время на поиск несуществующего алгоритма? Во-вторых, когда такая задача важна для приложения, как пересмотреть конечные цели и какие алгоритми-

[1] Задача, которая может быть решена в теории (например, при наличии больших, но ограниченных ресурсов, в особенности времени), но для решения которой на практике требуется слишком много ресурсов, называется *труднорешаемой* (intractable). — *Примеч. пер.*

ческие инструменты применить для их достижения? Эта книга снабдит вас исчерпывающими ответами на оба вопроса.

19.1. Задача о минимальном остовном дереве и задача коммивояжера: алгоритмическая загадка

Трудные вычислительные задачи могут выглядеть как простые и чтобы различить их, требуется наметанный глаз. Сравните знакомую задачу о минимальном остовном дереве и ее более требовательную двоюродную сестру — задачу коммивояжера.

19.1.1. Задача о минимальном остовном дереве

Одной из известных вычислительных задач, поддающейся решению ослепительно быстрым алгоритмом, является *задача о минимальном остовном дереве* (MST, minimum spanning tree), рассмотренная в главе 15 *Третьей части*.[1]

ЗАДАЧА О МИНИМАЛЬНОМ ОСТОВНОМ ДЕРЕВЕ

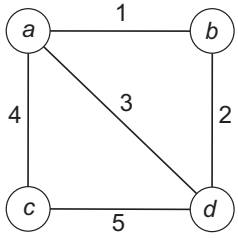
Вход: связный неориентированный граф $G = (V, E)$ и вещественнозначная (real-valued) стоимость c_e для каждого ребра $e \in E$.

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

Напомним, что граф $G = (V, E)$ является *связным*, если для каждой пары $v, w \in V$ вершин граф содержит путь из v в w. *Остовное дерево* графа G есть

[1] В качестве напоминания, *граф* $G = (V, E)$ имеет два ингредиента: множество *вершин* V и множество *ребер* E. В *неориентированном* графе каждое ребро $e \in E$ соответствует неупорядоченной паре $\{v, w\}$ вершин (записываемой как $e = (v, w)$ или $e = (w, v)$). В *ориентированном* графе каждое ребро (v, w) является упорядоченной парой, причем ребро направлено из v в w. Числа $|V|$ и $|E|$ вершин и ребер обычно обозначаются соответственно как n и m.

подмножество $T \subseteq E$ ребер, позволяющих подграфу (V, T) быть как связным, так и ациклическим. Например, в графе



минимальное остовное дерево включает ребра $(a; b)$, $(b; d)$ и $(a; c)$ при совокупной стоимости 7.

Граф может иметь экспоненциально растущее число остовных деревьев, поэтому исчерпывающий поиск возможен только для самых малых графов.[1] Но задача о минимальном остовном дереве может быть решена умными быстрыми алгоритмами, такими как алгоритмы Прима и Краскала. Развернув надлежащие структуры данных (кучи и непересекающиеся (дизъюнктивные) множества соответственно), оба алгоритма получают очень быстрые реализации со временем выполнения $O((m + n) \log n)$, где m и n — это число ребер и вершин входного графа.

19.1.2. Задача коммивояжера

Другой широко известной задачей, отсутствующей в предыдущих частях, но занимающей видное место в этой книге, является *задача коммивояжера* (TSP, traveling salesman problem). От задачи о минимальном остовном дереве ее отличают *туры* — простые циклы, охватывающие все вершины, — которые играют роль остовных деревьев.

---

[1] Например, *формула Кэли* является широко известным результатом из комбинаторики, которая утверждает, что n-вершинный полный граф (где присутствуют все $\binom{n}{2}$ возможных ребер) имеет ровно n^{n-2} разных остовных деревьев. Это больше, чем оценочное число атомов в известной Вселенной, при $n \geq 50$.

ЗАДАЧА: ТУРЫ КОММИВОЯЖЕРА

Вход: полный неориентированный граф $G = (V, E)$ и вещественнозначная стоимость c_e для каждого ребра $e \in E$.[1]

Выход: тур $T \subseteq E$ графа G с минимально возможной суммой $\sum_{e \in T} c_e$ реберных стоимостей.

Формально *тур* — это цикл, который посещает каждую вершину ровно один раз (причем два ребра инцидентны для каждой вершины).

ТЕСТОВОЕ ЗАДАНИЕ 19.1

Сколько отдельных туров $T \subseteq E$ существует в экземпляре $G = (V, E)$ задачи коммивояжера с $n \geq 3$ вершинами? (В приведенных ниже ответах $n! = n \times (n-1) \times (n-2) \ldots 2 \times 1$ обозначает факториальную функцию.)

а) 2^n

б) $\frac{1}{2}(n-1)!$

в) $(n-1)!$

г) $n!$

(Ответ и анализ решения см. в разделе 19.1.4.)

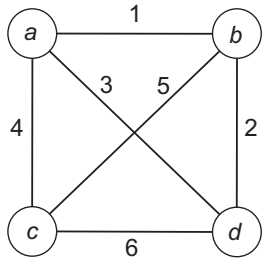
Если все остальное не срабатывает, то задачу коммивояжера можно решить, исчерпывающе перечислив все туры (их конечное число) и запомнив наилучший из них. Попробуйте исчерпывающий поиск на небольшом примере.

---

[1] В *полном* графе присутствуют все $\binom{n}{2}$ возможных ребер. Допущение о том, что граф является полным, не утрачивает общепримени мости, поскольку произвольный входной граф можно без вреда превратить в полный граф путем добавления всех недостающих ребер и присвоения им очень высоких стоимостей.

ТЕСТОВОЕ ЗАДАНИЕ 19.2

Какова минимальная сумма реберных стоимостей тура в следующем графе? (Каждое ребро помечено своей стоимостью.)



а) 12

б) 13

в) 14

г) 15

(Ответ и анализ решения см. в разделе 19.1.4.)

Задача коммивояжера может быть допустимо решена путем исчерпывающего поиска только для самых малых экземпляров. *Можем ли мы добиться лучшего?* Может ли существовать, по аналогии с задачей о минимальном остовном дереве, алгоритм, который волшебным образом находит самую дешевую иголку в экспоненциально растущем стоге сена туров коммивояжера? Несмотря на внешнее сходство формулировок этих двух задач, решение задачи коммивояжера представляется гораздо более трудным.

19.1.3. Безуспешные попытки решить задачу коммивояжера

Я мог бы рассказать дурацкую историю о коммивояжере, но не хочу вас запутать. Если увидите серию операций, расположенных в ряде, при том что

стоимость или время выполнения операции зависит от предыдущей операции, — это замаскированная задача коммивояжера.

Например, операции могут представлять сборку автомобилей на заводе. Время сборки автомобиля равно сумме фиксированной стоимости сборки и стоимости настройки, которая зависит от разницы заводских конфигураций для этого и предыдущего автомобиля. Максимально быстрая сборка всех автомобилей сводится к минимизации суммы стоимостей настройки, что и составляет задачу коммивояжера.

Возьмем совершенно другое применение. Представьте кучу перекрывающихся фрагментов генома, которые нужно правдоподобно упорядочить. Имея в руках «меру правдоподобия», назначающую стоимость каждой паре фрагментов (например, производную от длины их наибольшей общей подстроки), вы можете свести задачу упорядочения к задаче коммивояжера.[1]

Соблазненные практическим применением и эстетической привлекательностью задачи коммивояжера, многие величайшие умы в области оптимизации чуть ли не с начала 1950-х годов старались решить ее крупномасштабные экземпляры.[2] Но несмотря на десятилетия привлечения интеллектуальной мощи:

ФАКТ

Известного быстрого алгоритма для задачи коммивояжера на момент написания этой книги не существует.

---

[1] Оба примера, возможно, лучше моделируются как задачи *пути* коммивояжера, где вычисляется самый дешевый бесцикловый путь, проходящий по каждой вершине (не возвращаясь к старту). Любой алгоритм решения задачи коммивояжера может быть легко конвертирован в алгоритм, решающий путевую версию задачи, и наоборот (задача 19.7).

[2] Читателям, интересующимся историей или дополнениями задачи коммивояжера, следует ознакомиться с первыми четырьмя главами книги «Задача коммивояжера: вычислительное исследование» («*The Traveling Salesman Problem: A Computational Study», by David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook, Princeton University Press,* 2006).

Что мы подразумеваем под быстрым алгоритмом? Еще в *Первой части* мы согласились с тем, что:

> *Быстрый алгоритм — это алгоритм, время выполнения которого в наихудшем случае растет медленно вместе с размером входа.*

И что мы подразумеваем под «растет медленно»? Для большей части этой серии книг Святым Граалем были алгоритмы, работающие за линейное или почти линейное время. Забудьте о таких алгоритмах — никто не знает алгоритма для задачи коммивояжера, который всегда работает даже за время $O(n^{100})$ на n-вершинных экземплярах или даже за время $O(n^{10\,000})$.

Есть два конкурирующих объяснения этого мрачного состояния дел: либо быстрый алгоритм для задачи коммивояжера существует, но пока не найден, либо такого алгоритма нет. Большинство экспертов верят во второе.

УМОЗРИТЕЛЬНОЕ ЗАКЛЮЧЕНИЕ

Быстрого алгоритма для задачи коммивояжера не существует.

Еще в 1967 году Джек Эдмондс написал:

> *Я предполагаю, что хорошего алгоритма для задачи коммивояжера не существует. Мои доводы те же, что и для любого математического предположения: (1) это законная математическая возможность, и (2) я не знаю.*[1]

К сожалению, проклятие труднорешаемости не ограничивается задачей коммивояжера. Вы увидите, что многие другие практически релевантные задачи тоже от него страдают.

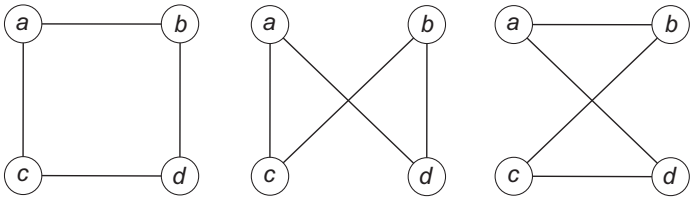
---

[1] Из статьи «Разветвления оптимума» Джека Эдмондса (*«Optimum Branchings», Journal of Research of the National Bureau of Standards, Series B*, 1967). Под «хорошим» алгоритмом Эдмондс понимает алгоритм с временем выполнения, ограниченным сверху некоторой полиномиальной функцией от размера входных данных.

19.1.4. Решения к тестовым заданиям 19.1–19.2

Решение к тестовому заданию 19.1

Правильный ответ: (б). Существует интуитивное соответствие между упорядочением вершин (которых существует $n!$) и турами (которые проходят по вершинам по одному разу в некотором порядке), поэтому напрашивается ответ (г). Однако это соответствие подсчитывает каждый тур $2n$ разными способами: по одному разу для каждого из n вариантов первой вершины и по одному разу для каждого из двух направлений прохождения тура. Таким образом, суммарное число туров равно $\frac{n!}{2n} = \frac{1}{2}(n-1)!$. Например, при $n = 4$ существует три четко различимых тура:



Решение к тестовому заданию 19.2

Правильный ответ: (б). Мы можем перечислить туры, начав с вершины *a* и пробуя все шесть возможных упорядочений других трех вершин, приняв, что тур заканчивается переходом из последней вершины обратно в *a*. (На самом деле это перечисление подсчитывает каждый тур дважды, по одному разу в каждом направлении.) Результаты:

Упорядочение вершин	Стоимость соответствующего тура
a, *b*, *c*, *d* или *a*, *d*, *c*, *b*	15
a, *b*, *d*, *c* или *a*, *c*, *d*, *b*	13
a, *c*, *b*, *d* или *a*, *d*, *b*, *c*	14

Самый короткий тур — второй с суммарной стоимостью 13.

19.2. Возможные уровни профессиональной компетенции

Одни вычислительные задачи проще, чем другие. Суть теории NP-трудности состоит в классификации задач — выделении легкорешаемых (подобных задаче о минимальном остовном дереве) и труднорешаемых (подобных задаче коммивояжера). Эта книга нацелена как на читателей, ищущих пособие начального уровня по этой теме, так и на тех, кто стремится к компетенции профессионального уровня. В этом разделе даются рекомендации, как подойти к остальной части книги в зависимости от ваших целей и ограничений.

Каковы ваши текущие и желаемые уровни профессиональной компетенции в распознавании и решении NP-трудных задач?[1]

Уровень 0: «Что такое NP-трудная задача?»

Это полное невежество — вы никогда не слышали об NP-трудности и не знаете, что многие практически релевантные вычислительные задачи считаются не поддающимися решению быстрым алгоритмом. Если я все сделал правильно, то эта книга будет доступной даже читателям уровня 0.

Уровень 1: «О, задача-то NP-трудная? Думаю, что мы должны либо переформулировать задачу (снизить поставленные нами цели), либо вложить больше ресурсов в ее решение».

Это осведомленность на уровне коктейльной вечеринки и, по меньшей мере, владение слухами об NP-трудности.[2] Например, управляя своим софтверным проектом, используете ли вы алгоритмическую либо оптимизационную компоненты? Если да, то вам следует приобрести знания уровня 1 на случай, если один из членов вашей команды столкнется с NP-трудной задачей и захочет обсудить с вами возможные дальнейшие шаги. Для этого изучите разделы 19.3, 19.4 и 19.6.

---

[1] Что не так с термином «NP»? См. раздел 19.6.

[2] Которые ассоциируются с занудными коктейльными вечеринками!

Уровень 2: «О, задача-то NP-трудная? Дайте мне шанс применить мой алгоритмический опыт и посмотрим, насколько далеко я смогу зайти».

Разработчики ПО при достижении уровня 2 приобретают уверенность и богатый инструментарий для разработки практически полезных алгоритмов решения или аппроксимации NP-трудных задач. Серьезные программисты должны быть нацелены на этот уровень (или выше). К счастью, все алгоритмические парадигмы, которые мы развернули в предыдущих частях серии, также полезны для продвижения к NP-трудным задачам. Главы 20 и 21 приведут вас к уровню 2, в разделе 19.4 вы найдете обзор, а в главе 24 — исследование инструментария уровня 2 и его применение.

Уровень 3: «Расскажите поподробнее о задаче. [. . .внимательно слушает. . .] Мои соболезнования, она — NP-трудная».

Вы можете быстро распознавать NP-трудность: знаете несколько известных NP-трудных задач и можете доказать NP-трудность дополнительных задач. Эти навыки позволяют вам консультировать по алгоритмическим вопросам коллег, студентов или инженеров в промышленности. Глава 22 содержит курс молодого бойца для повышения вашего уровня до 3, а раздел 19.5 — обзор.

Уровень 4: «Давайте я вам объясню вот тут на доске предположение, что $P \neq NP$».

Самый продвинутый уровень предназначен для начинающих теоретиков и тех, кто ищет строгого математического понимания NP-трудности и рассмотрения вопроса «P против NP». Если этот уровень вас не отпугивает, прочтите дополнительную главу 23.

19.3. «Легкие» и «трудные» задачи

Противопоставление «легкой» и «трудной» задач в теории NP-трудности простыми словами звучит так:

«легкая» может быть решена полиномиально-временным алгоритмом,

«трудная» в худшем случае требует экспоненциального времени.

Это краткое изложение NP-трудности упускает из виду несколько важных тонкостей (см. раздел 19.3.9). Но если через десять лет о смысле NP-трудности вы вспомните только эти несколько слов, то они будут самыми точными.

19.3.1. Полиномиально-временные алгоритмы

Чтобы перейти к определению «легкой» задачи, давайте вспомним время выполнения некоторых известных алгоритмов (из предыдущих частей серии):

Задача	Алгоритм	Время выполнения
Сортировка	`MergeSort`	$O(n \log n)$
Упорядочение сильно связных компонент	`Kosaraju`	$O(m + n)$
Поиск кратчайшего пути	`Dijkstra`	$O((m + n) \log n)$
Определение минимального остовного дерева	`Kruskal`	$O((m + n) \log n)$
Выравнивание рядов	`NW`	$O(mn)$
Поиск кратчайшего пути для всех пар вершин	`Floyd-Warshall`	$O(n^3)$

Точный смысл n и m зависит от конкретной задачи, но во всех случаях они связаны с размером входных данных.[1] Согласно таблице, хотя время выполнения алгоритмов варьируется, *все они зависят от размера входных данных в рамках полиномиальной функции*. В общем случае:

ПОЛИНОМИАЛЬНО-ВРЕМЕННЫЕ АЛГОРИТМЫ

Полиномиально-временной алгоритм — это алгоритм с временем выполнения, в наихудшем случае равным $O(n^d)$, где n обозначает размер входных данных и d — это константа (независимая от n).

[1] В задаче сортировки n обозначает длину входной кучи. В четырех графовых задачах n и m обозначают соответственно число вершин и ребер. В задаче о выравнивании рядов n и m обозначают длины двух входных строк.

Все шесть алгоритмов являются полиномиально-временными (с достаточно малыми экспонентами d).[1] Все ли естественные алгоритмы выполняются за полиномиальное время? Нет. Например, для многих задач исчерпывающий поиск выполняется за время, экспоненциально зависящее от размера входных данных (как указано во второй сноске к задаче о минимальном остовном дереве). В полиномиально-временных алгоритмах, которые мы ранее изучили, есть кое-что особенное.

19.3.2. Полиномиальное время против экспоненциального

Не забывайте, что любая экспоненциальная функция в конечном счете растет намного быстрее любой полиномиальной функции. Между типичным полиномиальным и экспоненциальным временем выполнения существует огромная разница, даже для очень малых экземпляров. Рассмотрите график, на котором изображены полиномиальная функция $100n^2$ и экспоненциальная функция 2^n.

Закон Мура утверждает, что вычислительная мощность, доступная по данной цене, удваивается каждые 1–2 года. Означает ли это, что разница между полиномиально- и экспоненциально-временными алгоритмами со временем исчезнет? На самом деле наоборот! Вычислительные амбиции растут вместе с вычислительной мощностью, и с течением времени мы видим все более крупные размеры входных данных и страдаем от все более огромной пропасти между полиномиальным и экспоненциальным периодами выполнения.

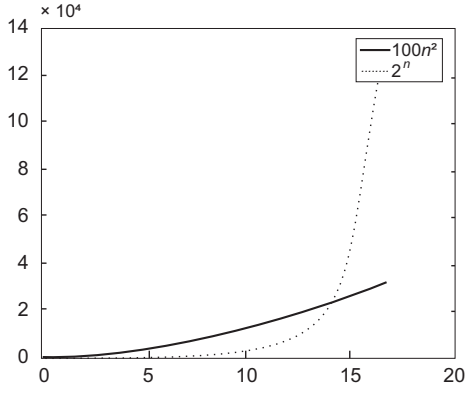
Представьте фиксированный бюджет времени, например час или день. Как масштабируется поддающийся решению размер входных данных вместе с добавочной вычислительной мощностью? С полиномиально-временным алгоритмом он увеличивается на постоянный коэффициент (например, с 1 000 000 до 1 414 213) с каждым удвоением вычислительной мощности.[2] С алгоритмом, который работает со временем, пропорциональным 2^n, где n — это размер входных данных, каждое удвоение вычислительной мощно-

---

[1] Вспомните, что логарифмический фактор может быть ограничен сверху (небрежно) линейным фактором; например, если $T(n) = O(n \log n)$, то и $T(n) = O(n^2)$.

[2] С линейно-временным алгоритмом вы можете решать задачи, которые вдвое больше предыдущих. С квадратично-временным алгоритмом — в $\sqrt{2} \approx 1.414$ раза больше. С кубически-временным алгоритмом — в $\sqrt[3]{2} \approx 1.26$ раза и т. д.

сти увеличивает поддающийся решению размер входных данных только на единицу (например, с 1 000 000 до 1 000 001)!



19.3.3. Легкорешаемые задачи

Теория NP-трудности определяет, что «легкие» задачи поддаются решению полиномиально-временным алгоритмом или, что эквивалентно, алгоритмом, для которого поддающийся решению размер входных данных (для фиксированного бюджета времени) масштабируется мультипликативно вместе с увеличением вычислительной мощности:[1]

ЗАДАЧИ, ПОДДАЮЩИЕСЯ РЕШЕНИЮ ЗА ПОЛИНОМИАЛЬНОЕ ВРЕМЯ

Вычислительная задача *поддается решению за полиномиальное время*, если существует полиномиально-временной алгоритм, который решает ее правильно для каждого элемента входных данных.

Например, все шесть перечисленных задач поддаются решению за полиномиальное время.

[1] Это определение было предложено независимо Аланом Кобэмом и Джеком Эдмондсом (см. сноску на с. 25) в середине 1960-х годов.

Технически алгоритм (бесполезный на практике), который выполняется за время $O(n^{100})$ на n-размерных входных данных, считается полиномиально-временным, и задача, решенная таким алгоритмом, квалифицируется как поддающаяся решению за полиномиальное время. Если перевернуть это утверждение, окажется, что если задача, такая как задача коммивояжера, не поддается решению за полиномиальное время, то не существует даже $O(n^{100})$-временного или $O(n^{10\,000})$-временного алгоритма, который ее решит (!).

СМЕЛОСТЬ, ОПРЕДЕЛЕНИЯ И КРАЙНИЕ СЛУЧАИ

Отождествление понятия «легкая» с понятием «поддающаяся решению за полиномиальное время» страдает несовершенством. Задача может быть решена в теории (с помощью алгоритма, который технически выполняется за полиномиальное время), но не в реальности (посредством эмпирически быстрого алгоритма), или наоборот. Любой, у кого хватит смелости написать точное математическое определение (например, полиномиально-временной решаемости), чтобы выразить беспорядочную концепцию реального мира (например, «легко решить посредством компьютера в физическом мире»), должен быть готов к трениям между двоичной природой определения и нечеткостью реальности. Определение неизбежно будет включать или исключать некоторые крайние случаи, которые хочется обойти, но это не повод игнорировать или отклонять хорошее определение. Полиномиально-временная решаемость эффективна[1] в разделении задач на «легкие» и «трудные» на эмпирической основе. Имея за плечами полвека свидетельств, мы можем с уверенностью сказать, что естественные задачи, поддающиеся решению за полиномиальное время, в типичной ситуации могут решаться практическими общецелевыми алгоритмами и что задачи, считающиеся не поддающимися решению за полиномиальное время, в типичной ситуации требуют значительно больше работы и компетенции в предметной области.

---

[1] О необоснованной (необъяснимой) эффективности математики см. https://en.wikipedia.org/wiki/The_Unreasonable_Effectiveness_of_Mathematics_in_the_Natural_Sciences. — *Примеч. пер.*

19.3.4. Относительная труднорешаемость

Предположим, вы заподозрили, что задача, такая как задача коммивояжера, «не является легкой», имея в виду, что она не поддается решению любым полиномиально-временным алгоритмом (независимо от того, насколько крупным является полином). Как это доказать? Самым убедительным аргументом, конечно, будет герметичное математическое доказательство. Но статус задачи коммивояжера остается в подвешенном состоянии: никто не нашел полиномиально-временного алгоритма, который ее решает, или доказательства, что такого алгоритма не существует.

Как развить теорию, которая с пользой дифференцирует «легкие» и «трудные» задачи при нашем недостаточном понимании всех возможностей алгоритмов? Блестящая идея теории NP-трудности состоит в классификации задач на основе их *относительной* (а не абсолютной) трудности и объявления задачи «трудной», если она «по меньшей мере так же трудна, как» подавляющее число других нерешенных задач.

19.3.5. Трудные задачи

Безуспешные попытки решить задачу коммивояжера (раздел 19.1.3) дают косвенные свидетельства, что эта задача, возможно, не решается за полиномиальное время.

> **СЛАБЫЕ ДОКАЗАТЕЛЬСТВА ТРУДНОСТИ**
>
> Полиномиально-временной алгоритм решил бы задачу коммивояжера, которая сопротивлялась усилиям сотен (если не тысяч) умов на протяжении десятилетий.

Есть ли более убедительный аргумент? Основная идея NP-трудности состоит в том, чтобы показать, что такая задача, как задача коммивояжера, по меньшей мере так же трудна, как и огромное количество нерешенных задач

из множества различных научных областей. По сути, в этих задачах легко распознаются решения, стоит только вам понять задачу. А значит, гипотетический полиномиально-временной алгоритм для задачи коммивояжера автоматически будет решать и остальные нерешенные задачи!

СИЛЬНЫЕ ДОКАЗАТЕЛЬСТВА ТРУДНОСТИ

Полиномиально-временной алгоритм для задачи коммивояжера позволил бы решить *тысячи* задач, которые на протяжении десятилетий противостояли усилиям *десятков (если не сотен) тысяч* умов.

В сущности, теория NP-трудности показывает, что тысячи вычислительных задач являются замаскированными вариантами одной и той же задачи, и полиномиально-временной алгоритм для одной из NP-трудных задач решит все остальные.[1]

Мы называем задачу *NP-трудной*, если есть веские свидетельства труднорешаемости в указанном выше смысле:

NP-ТРУДНОСТЬ (ГЛАВНАЯ ИДЕЯ)

Задача является *NP-трудной*, если она по меньшей мере так же трудна, как и любая задача, решения которой вы легко распозна́ете, если поймете ее суть.

[1] Выступая в качестве адвоката дьявола, сотни (если не тысячи) блестящих умов также не смогли доказать другое направление, что задача коммивояжера *не* поддается решению за полиномиальное время. Симметричным образом, разве это не предполагает, что, возможно, такого доказательства не существует? Разница в том, что мы, по всей видимости, гораздо лучше доказываем решаемость (быстрыми алгоритмами, известными для бесчисленных задач), чем нерешаемость. Отсюда, если бы задача коммивояжера поддавалась решению за полиномиальное время, то было бы странно, что мы еще не нашли для нее полиномиально-временной алгоритм; если нет, то неудивительно, что мы еще не выяснили, как его доказать.

Эта идея будет полностью подтверждена в разделе 23.3.4. А пока мы поработаем с предварительным определением NP-трудности в терминах широко известного математического предположения, что P ≠ NP.

19.3.6. Предположение, что P ≠ NP

Возможно, вы слышали о предположении, что P ≠ NP. Что оно собой представляет? Раздел 23.4 содержит его точную математическую формулировку, но на данный момент мы остановимся на неформальной версии, которая звучит как девиз при проверке домашнего задания:

> **ПРЕДПОЛОЖЕНИЕ, ЧТО P ≠ NP (НЕФОРМАЛЬНАЯ ВЕРСИЯ)**
>
> Проверить предполагаемое решение задачи точно проще, чем создать собственное решение с нуля.

Здесь P и NP относятся соответственно к задачам, которые могут быть решены с нуля за полиномиальное время, и к тем задачам, решения которых могут быть проверены за полиномиальное время (формальные определения P и NP в главе 23).

Конечно, проверка предложенного кем-то решения судоку или кенкен выглядит проще самостоятельного разбора головоломки. Как и легко проверить, что предложенный кем-то тур коммивояжера является хорошим (с суммарной стоимостью, скажем, не более 1000), просуммировав стоимости его ребер. Не ясно, как быстро можно придумать такой тур с нуля. Поэтому пояснение ниже настаивает: предположение, что P ≠ NP, является истинным.[1, 2]

---

[1] В задаче 23.2 вы увидите: предположение, что P ≠ NP, эквивалентно утверждению Эдмондса (с. 24) о том, что задача коммивояжера не может быть решена за полиномиальное время.

[2] Почему предположение, что P ≠ NP, не «очевидно» истинно? Потому что пространство полиномиально-временных алгоритмов является непостижимо насыщенным, с многочисленными гениальными обитателями. (Вспомните умопомрачительный субкубический алгоритм умножения матриц Штрассена из главы 3 *Первой части*.) Доказательство того факта, что ни один из бесконечно многих алгоритмов не решает задачу коммивояжера, кажется довольно пугающим!

19.3.7. Предварительное определение NP-трудности

Условно мы будем называть задачу NP-трудной, если, исходя из *допущения* об истинности неравенства P ≠ NP, она не может быть решена никаким полиномиально-временным алгоритмом.

> **NP-ТРУДНАЯ ЗАДАЧА (ПРЕДВАРИТЕЛЬНОЕ ОПРЕДЕЛЕНИЕ)**
>
> Вычислительная задача является *NP-трудной*, если полиномиально-временной алгоритм, ее решающий, опровергает предположение, что P ≠ NP.

Таким образом, рабочий полиномиально-временной алгоритм для NP-трудной задачи подразумевает ложность предположения, что P ≠ NP. Полиномиально-временной алгоритм для каждой отдельной задачи, решения для которой могут быть распознаны за полиномиальное время, — это чрезмерная алгоритмическая щедрость. Вероятно, если предположение, что P ≠ NP является истинным, то никакая NP-трудная задача не решится за полиномиальное время, даже алгоритмом, работающим за $O(n^{100})$- или $O(n^{10\,000})$-е время на n-размерных входных данных.

19.3.8. Рандомизированные и квантовые алгоритмы

Наше определение решаемости за полиномиальное время на с. 31 рассматривает только детерминированные алгоритмы. Как известно, рандомизация бывает мощным инструментом проектирования алгоритмов (например, таких как QuickSort). Могут ли рандомизированные алгоритмы избежать пут NP-трудности?

И как насчет хваленых разрекламированных квантовых алгоритмов? (Ведь рандомизированные алгоритмы можно рассматривать как частный случай квантовых алгоритмов.) Следует признать, что крупномасштабные обще-

целевые квантовые компьютеры (если бы они были реализованы) изменили бы правила игры для нескольких задач, включая чрезвычайно важную задачу факторизации крупных целых чисел. Однако задача факторизации не признана NP-трудной, и эксперты предполагают, что даже квантовые компьютеры не способны решать NP-трудные задачи за полиномиальное время. Проблемы, связанные с NP-трудностью, в ближайшее время не исчезнут.[1]

19.3.9. Тонкости

Сверхупрощенное обсуждение на с. 28 наводит на мысль, что решение «трудной» задачи требует экспоненциального времени в худшем случае. Но определение из раздела 19.3.7 говорит нечто другое: NP-трудную задачу при $P \neq NP$ невозможно решить никаким полиномиально-временным алгоритмом.

Первое расхождение между этими определениями в том, что NP-трудность исключает полиномиально-временную решаемость только в том случае, когда предположение, что $P \neq NP$, является истинным. Но вопрос его истинности остается открытым.

Второе несоответствие в том, что даже в случае истинности предположения, что $P \neq NP$, для решения NP-трудной задачи в худшем случае требуется сверхполиномиальное (не экспоненциальное) время.[2] Однако эксперты полагают, что для большинства естественных NP-трудных задач, включая задачи из этой книги, действительно требуется экспоненциальное время в худшем случае.

---

[1] Большинство экспертов считают, что каждый рандомизированный полиномиально-временной алгоритм может быть *дерандомизирован* и превращен в эквивалентный детерминированный полиномиально-временной алгоритм (возможно, с более крупным полиномом в границе времени выполнения). Если это верно, то предположение, что $P \neq NP$, применимо и к рандомизированным алгоритмам.

В то же время эксперты сходятся во мнении, что квантовые алгоритмы *уже* фундаментально мощнее классических алгоритмов (но недостаточно мощны для решения NP-трудных задач за полиномиальное время). Разве это не удивительно — и не волнующе, — как много мы до сих пор еще не знаем?

[2] Примеры границ времени выполнения, которые являются сверхполиномиальными по размеру n входных данных, включают $n^{\log_2 n}$ и $2^{\sqrt{n}}$.

Это убеждение формализуется «гипотезой об экспоненциальном времени», более сильной формой предположения, что P ≠ NP (раздел 23.5).[1]

Наконец, несмотря на то что 99 % задач, с которыми вы столкнетесь, будут либо «легкими» (поддающимися решению за полиномиальное время), либо «трудными» (NP-трудными), несколько редких примеров окажутся посередине. То есть распределение на две группы охватывает большинство, но не все практически релевантные вычислительные задачи.[2]

19.4. Алгоритмические стратегии для NP-трудных задач

Предположим, вы определили задачу, от решения которой зависит успех проекта. Возможно, вы несколько недель, почти швыряясь посудой, пробовали все известные вам парадигмы проектирования алгоритмов, каждую структуру данных из книги, все бесплатные примитивы, но ничего не работало. Наконец, вы поняли, что проблема не в недостатке вашей изобретательности, а в том, что задача — NP-трудная. Вам стало легче, но это не умалило значимости задачи для проекта. Что делать?

19.4.1. Универсальный, правильный, быстрый (выбрать два)

Плохая новость: NP-трудные задачи распространены повсеместно. Прямо сейчас одна из них может скрываться в вашем последнем проекте. Хорошая

---

[1] Ни одна из вычислительных задач, изученных в этой серии книг, не требует для решения экспоненциального времени. Однако существуют другие задачи. Например, задача об остановке, которая не может быть решена за любой конечный (не говоря уже об экспоненциальном) промежуток времени. См. также раздел 23.1.2.

[2] Две важные задачи, которые не считаются ни решаемыми за полиномиальное время, ни NP-трудными, — это факторизация (поиск нетривиального множителя целого числа) и задача о графовом изоморфизме (определение идентичности двух графов вплоть до переименования вершин). Субэкспоненциально-временные (но не полиномиально-временные) алгоритмы известны для обеих задач.

новость: NP-трудность — не смертный приговор. Часто (но не всегда) такие задачи могут быть решены на практике по меньшей мере приближенно путем достаточного вложения ресурсов и алгоритмической изощренности.

NP-трудность бросает вызов разработчику алгоритмов и сообщает ему, чего ждать. Не надейтесь получить общецелевой и всегда быстрый алгоритм для NP-трудной задачи сродни тем, которые используются для сортировки, поиска кратчайшего пути или выравнивания рядов. Если вам не повезет столкнуться с необычно малыми или хорошо структурированными входными данными, придется потрудиться над решением NP-трудной задачи и, возможно, пойти на некоторые компромиссы.

Какого рода компромиссы? NP-трудность исключает алгоритмы, имеющие одновременно три свойства (исходя из предположения о том, что $P \neq NP$):

ТРИ СВОЙСТВА (ОДНИМ ИЗ НИХ НУЖНО ПОЖЕРТВОВАТЬ)

1. *Универсальный*. Алгоритм учитывает все возможные входы вычислительной задачи.
2. *Правильный*. Для каждого входа алгоритм решает задачу правильно.
3. *Быстрый*. Для каждого входа алгоритм выполняется за полиномиальное время.

Вы можете выбрать один компромисс: в отношении универсальности, правильности или скорости. Все три стратегии полезны и распространены на практике.

Остальная часть раздела посвящена этим стратегиям. Главы 20 и 21 подробно освещают последние две. Как всегда, обращаем внимание на принципы проектирования мощных и гибких алгоритмов для широкого круга задач. Вооружитесь этими принципами и применяйте их в соответствии с предметной областью задачи.

19.4.2. Компромисс в отношении универсальности

Одна из стратегий достижения прогресса в решении NP-трудной задачи — в отказе от универсальных алгоритмов и сосредоточении на частных случаях задачи, имеющих отношение к проекту. В лучшем случае это позволит определить специфические для предметной области ограничения на входные данные и разработать алгоритм, который всегда будет правильным и быстрым на этом подмножестве входных данных. Выпускники курса молодого бойца по динамическому программированию в *Третьей части* уже видели два примера этой стратегии.

Задача о взвешенном независимом множестве. На вход подается неориентированный граф $G = (V, E)$ и неотрицательный вес w_v для каждой вершины $v \in V$. Цель состоит в вычислении независимого множества $S \subseteq V$ с максимально возможной суммой $\sum_{v \in S} w_v$ вершинных весов, где независимое множество — это подмножество $S \subseteq V$ взаимно несмежных вершин (где $(v, w) \notin E$ для каждого v, а $w \in S$). Например, если ребра представляют конфликты (между людьми, курсами и т. д.), то независимые множества соответствуют бесконфликтным подмножествам. Эта задача является NP-трудной в общем случае (раздел 22.5). Частный случай задачи, в которой G является путевым графом (с вершинами $v_1, v_2, \ldots, v_n$ и ребрами $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$), решается за линейное время алгоритмом динамического программирования, который можно расширить за счет размещения всех ациклических графов (см. задачу 16.3 *Третьей части*).

Задача о рюкзаке. Входные данные задаются $2n + 1$ положительными целыми числами: n значениями $v_1, v_2, \ldots, v_n$ предметов, n размерами $s_1, s_2, \ldots, s_n$ предметов и вместимостью C рюкзака. Цель состоит в вычислении подмножества $S \subseteq \{1, 2, \ldots, n\}$ предметов с максимально возможной суммой $\sum_{i \in S} v_i$ значений, при условии наличия суммарного размера $\sum_{i \in S} S_i$, не превышающего C. Другими словами, цель заключается в использовании дефицитного ресурса самым ценным путем.[1] Эта задача является NP-трудной, как мы увидим в разделе 22.8

[1] Например, на какие товары и услуги вы должны потратить свою зарплату, чтобы извлечь наибольшую ценность? Или же, с учетом оперативного бюджета и имея перечень претендентов с разным уровнем продуктивности труда и требуемой заработной платой, кого вам следует нанять?

и задаче 22.7. Для нее существует $O(nC)$-временной алгоритм динамического программирования, который является полиномиально-временным в частном случае, когда C ограничена полиномиальной функцией от n.

ПОЛИНОМИАЛЬНО-ВРЕМЕННОЙ АЛГОРИТМ ДЛЯ РЮКЗАКА?

Почему $O(nC)$-временной алгоритм для задачи о рюкзаке не опровергает предположение, что $P \neq NP$? Потому что это не полиномиально-временной алгоритм. Размер входных данных — число нажатий клавиш, необходимых для ввода этих данных, — масштабируется вместе с количеством *цифр* в числе, а не с *величиной* числа. Чтобы передать число «1 000 000», не требуется миллион нажатий клавиш — всего 7 (либо 20, если вы работаете с основанием 2). Например, в экземпляре с n предметами, вместимостью рюкзака 2^n и всеми значениями и размерами предметов не более 2^n размер входных данных равен $O(n^2) - O(n)$ числам с $O(n)$ цифрами каждое, тогда как время выполнения алгоритма динамического программирования экспоненциально более продолжительно (пропорционально $n \times 2^n$).

Алгоритмическая стратегия разработки быстрых и правильных алгоритмов (для частных случаев) использует весь алгоритмический инструментарий предыдущих частей. По этой причине ни одна глава этой книги не посвящена данной стратегии. Однако по пути мы столкнемся с примерами частных случаев NP-трудных задач, поддающихся решению за полиномиальное время, включая задачи коммивояжера, выполнимости (булевых формул) и раскраски графов (задачи 19.8 и 21.12).

19.4.3. Компромисс в отношении правильности

Вторая алгоритмическая стратегия, особенно популярная в критичных по времени приложениях, заключается в отказе от правильности. Алгоритмы, которые не всегда являются правильными, иногда называются *эвристическими алгоритмами*.[1]

---

[1] В предыдущих частях есть ровно один пример в основном правильного решения: фильтры Блума — структуры данных, занимающие малое пространство при сверхбыстрых операциях вставки и поиска за счет периодических ложных срабатываний.

В идеале эвристический алгоритм является «в основном правильным». Это соответствует одному из двух или сразу обоим утверждениям:

ОСЛАБЛЕНИЕ ПРАВИЛЬНОСТИ

1. Алгоритм является правильным на «большинстве» входных данных.[1]
2. Алгоритм является «почти правильным» на каждом элементе входных данных.

Второе свойство легче всего интерпретировать для задач оптимизации, где вычисляется допустимое решение (например, тур коммивояжера) с наилучшим значением целевой функции (например, минимальной суммарной стоимостью). «Почти правильный» здесь означает, что алгоритм выводит допустимое решение со значением целевой функции, близким к наилучшему из возможных, как тур коммивояжера с суммарной стоимостью ненамного большей, чем у оптимального тура.

Ваш алгоритмический инструментарий для разработки быстрых точных алгоритмов будет полезен и для создания эвристических алгоритмов. В разделах 20.1–20.3 описаны жадные эвристики (для задач от планирования до максимизации влияния в социальных сетях) с доказательствами «приближенной правильности», гарантирующими, что для каждого входа значение целевой функции на выходе находится в пределах скромного постоянного коэффициента ее наилучшего возможного значения.[2]

Разделы 20.4–20.5 дополнят ваш инструментарий парадигмой проектирования алгоритмов локального поиска. Локальный поиск и его обобщения эффективны на практике при решении многих NP-трудных задач, хотя он редко гарантирует приближенную правильность.

---

[1] Например, одна типичная реализация фильтра Блума имеет интенсивность ложных срабатываний 2 %, при этом 98 % операций поиска отвечают правильно.

[2] Некоторые авторы называют такие алгоритмы алгоритмами аппроксимации, оставляя термин «эвристические алгоритмы» для алгоритмов, которые не имеют таких доказательств приближенной правильности.

19.4.4. Компромисс в отношении скорости

Заключительная стратегия подходит для приложений, в которых недопустим компромисс в отношении правильности. Каждый правильный алгоритм для NP-трудной задачи должен выполняться за сверхполиномиальное время на некоторых входах (если P ≠ NP). Следовательно, нужен алгоритм, который будет как минимум максимально быстрым, чтобы превзойти исчерпывающий поиск. Это соответствует одному из двух или сразу обоим утверждениям:

ОСЛАБЛЕНИЕ ПОЛИНОМИАЛЬНОГО ВРЕМЕНИ ВЫПОЛНЕНИЯ

1. Алгоритм обычно выполняется быстро (например, за полиномиальное время) на входных данных, релевантных для приложения.
2. Алгоритм работает быстрее, чем исчерпывающий поиск на каждом элементе входных данных.

Во втором случае мы все еще ждем, что на некоторых входах алгоритм будет выполняться за экспоненциальное время — в конце концов, задача является NP-трудной. Раздел 21.1 задействует динамическое программирование, чтобы превзойти исчерпывающий поиск для задачи коммивояжера, сократив время выполнения с $O(n!)$ до $O(n^2 \times 2_n)$, где n — это число вершин. Раздел 21.2 совмещает рандомизацию с динамическим программированием, одерживая верх над исчерпывающим поиском для задачи отыскания длинных путей в графах (с временем выполнения $O((2e)^k \times m)$, а не $O(n^k)$, где n и m обозначают числа вершин и ребер во входном графе, k — длину целевого пути, и $e = 2{,}718\ldots$).

Достижение прогресса на относительно крупных экземплярах NP-трудных задач в типичной ситуации требует дополнительных инструментов, которые не гарантируют победу над скоростью исчерпывающего поиска, но во многих приложениях показали свою эффективность. Разделы 21.3–21.5 описывают многолетний опыт экспертов в разработке мощных решателей для задач смешанного целочисленного программирования (MIP, mixed integer programming) и задач выполнимости булевых формул (SAT, satisfiability). Многие NP-трудные задачи оптимизации (такие как задача коммивояжера) могут быть закодированы

как задачи MIP, а задачи о проверке допустимости (такие как проверка бесконфликтного закрепления учебных занятий за учебными аудиториями) легко выражаются как задачи SAT. Конечно, нет гарантии, что решатель задач MIP или SAT решит ваш конкретный экземпляр за разумное время, но для решения NP-трудных задач на практике сегодня ничего лучше вы не найдете.

19.4.5. Ключевые выводы

Если вы нацелены на получение знаний о NP-трудности уровня 1 (раздел 19.2), то самые важные вещи, которые нужно запомнить, таковы:

ТРИ ФАКТА ОБ NP-ТРУДНЫХ ЗАДАЧАХ

1. *Повсеместность*: практически релевантные NP-трудные задачи есть везде.
2. *Труднорешаемость*: согласно широко распространенному математическому предположению, ни одна NP-трудная задача не может быть решена любым алгоритмом, который всегда является правильным и всегда выполняется за полиномиальное время.
3. *Не смертельный приговор*: NP-трудные задачи часто (но не всегда) можно решить на практике, по меньшей мере приближенно, путем достаточного вложения ресурсов и алгоритмической изощренности.

19.5. Доказательство NP-трудности: простой рецепт

Как распознать NP-трудную задачу, чтобы скорректировать конечные цели и отказаться от поиска алгоритма, который якобы будет универсальным, правильным и быстрым? Никто не выиграет, если вы будете тратить недели или месяцы на попытки опровергнуть предположение, что $P \neq NP$.

Во-первых, имейте под рукой коллекцию простых и распространенных NP-трудных задач (например, 19 задач в главе 22, рис. 22.1) — в самом простом

сценарии ваше приложение сводится к одной из них. Во-вторых, оттачивайте умение сводить одну вычислительную задачу к другой. Такая редукция распространит легкорешаемость от второй задачи к первой. Причем труднорешаемость тоже распространится от первой задачи ко второй, поэтому, чтобы показать, что интересующая вас задача является NP-трудной, нужно лишь свести к ней известную NP-трудную задачу.

Остальная часть этого раздела посвящена данной теме и дает один простой пример (подробнее в главе 22).

19.5.1. Редукции

Любая задача *B*, которая является по меньшей мере такой же трудной, как и NP-трудная задача *A*, сама по себе является NP-трудной. Фраза «по меньшей мере такой же трудной, как» может быть формализована с помощью редукций.

РЕДУКЦИИ

Задача *A* *сводится* к задаче *B*, если алгоритм, решающий задачу *B*, может быть легко переведен в алгоритм, решающий задачу *A* (рис. 19.1).

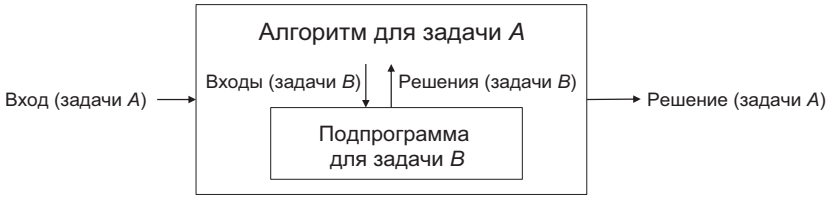


Рис. 19.1. Если задача *A* сводится к задаче *B*, то задача *A* может быть решена с помощью полиномиального (по входному размеру) числа вызовов подпрограммы для *B* и полиномиального объема дополнительной работы

В контексте NP-трудности словосочетание «легко перевести» означает, что задача *A* может быть решена за не более чем полиномиальное (по размеру

входных данных) число вызовов подпрограммы, решающей задачу B, вместе с полиномиальным объемом дополнительной работы (за пределами вызовов подпрограммы).

19.5.2. Использование упрощений для разработки быстрых алгоритмов

Опытные разработчики алгоритмов всегда ищут редукции, или упрощения, — зачем решать задачу с нуля, если это необязательно? Примеры из предыдущих частей, относящиеся к задачам из раздела 19.3.1, включают:

ЗНАКОМЫЕ ПРИМЕРЫ УПРОЩЕНИЙ

1. Поиск медианы кучи целых чисел сводится к сортировке кучи. (После сортировки кучи вернуть срединный элемент.)
2. Поиск кратчайшего пути для всех пар вершин сводится к поиску кратчайшего пути с одним истоком. (Вызвать одноистоковый алгоритм кратчайшего пути один раз с каждым возможным вариантом выбора стартовой вершины во входном графе.)
3. Поиск наибольшей общей подпоследовательности сводится к выравниванию рядов. (Вызвать алгоритм выравнивания рядов с двумя входными строками, штрафом 1 за вставляемый пробел (зазор) и очень большим штрафом за каждое несовпадение двух разных символов.)[1]

Эти упрощения принимают светлую сторону силы и служат почетной миссии создания новых быстрых алгоритмов из старых, тем самым расширяя границу легкорешаемости. Например, первое упрощение преобразовывает алгоритм `MergeSort` в $O(n \log n)$-временной алгоритм поиска медианы или, выражаясь шире, превращает любой $T(n)$-временной алгоритм сортировки

[1] Вспомните, что экземпляр задачи о выравнивании рядов задается двумя строками над некоторым алфавитом (например, $\{A, C, G, T\}$), штрафом α_{xy} для каждой пары символов $x, y \in \Sigma$ и неотрицательным штрафом за зазор α_{gap}. Цель — вычислить выравнивание входных строк с минимально возможным суммарным штрафом.

в $O(T(n))$-временной алгоритм поиска медианы, где n — это длина кучи. Второе — преобразовывает любой $T(m, n)$-временной алгоритм для поиска кратчайшего пути с одним истоком в $O(n \times T(m, n))$-временной алгоритм для поиска кратчайшего пути для всех пар вершин, где m и n обозначают соответственно число ребер и вершин. Третье — превращает $T(m, n)$-временной алгоритм для выравнивания рядов в $O(T(m, n))$-временной алгоритм для нахождения наибольшей общей подпоследовательности, где m и n обозначают длины двух входных строк.

ТЕСТОВОЕ ЗАДАНИЕ 19.3

Предположим, что задача A может быть решена вызовом подпрограммы для задачи B не более $T_1(n)$ раз и выполнением не более $T_2(n)$-й дополнительной работы (вне вызовов подпрограммы), где n обозначает размер входных данных. При наличии подпрограммы, решающей задачу B за время не более $T_3(n)$ на n-размерных входных данных, сколько времени нужно для решения задачи A? (Выберите самое сильное истинное утверждение. Считайте, что программа использует по меньшей мере s примитивных операций для построения s-размерных входных данных, чтобы вызвать подпрограмму.)

а) $T_1(n) + T_2(n) + T_3(n)$

б) $T_1(n) \times T_2(n) + T_3(n)$

в) $T_1(n) \times T_3(n) + T_2(n)$

г) $T_1(n) \times T_3(T_2(n)) + T_2(n)$

(Ответ и анализ решения см. в разделе 19.5.5.)

Тестовое задание 19.3 показывает, что при сведении задачи A к задаче B любой полиномиально-временной алгоритм для B может быть переведен в алгоритм для A.[1]

[1] Если функции $T_1(n)$, $T_2(n)$ и $T_3(n)$ в тестовом задании 19.3 ограничены полиномиальной функцией от n, то и их суммы, произведения и композиции тоже ограничены. Например, если $T_1(n) \leq a_1 n^{d_1}$ и $T_2(n) \leq a_2 n^{d_2}$, где a_1, a_2, d_1 и d_2 — это положительные константы (независимые от n), то $T_1(n) \times T_2(n) \leq (a_1 a_2) n^{(d_1 + d_2)}$ и $T_1(T_2(n)) \leq (a_1 a_2^{d_1}) n^{(d_1 d_2)}$.

УПРОЩЕНИЯ СПОСОБСТВУЮТ ЛЕГКОРЕШАЕМОСТИ

Если задача *A* сводится к задаче *B* и *B* может быть решена полиномиально-временным алгоритмом, то *A* тоже может быть решена полиномиально-временным алгоритмом (рис. 19.2).

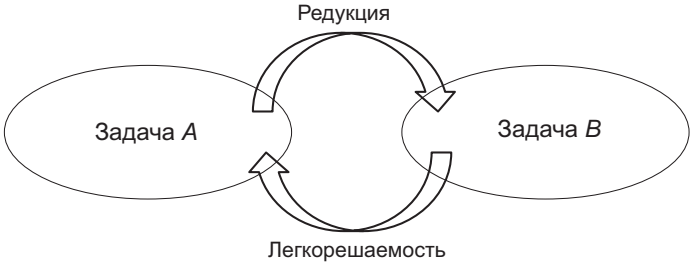


Рис. 19.2. Распространение легкорешаемости от *B* к *A*: если задача *A* сводится к задаче *B* и *B* — легкорешаемая задача, то *A* — тоже легкорешаемая

19.5.3. Использование упрощений для распространения NP-трудности

Теория NP-трудности следует темной стороне силы, часто используя редукции для распространения проклятия труднорешаемости (в противоположность рис. 19.2). Давайте перевернем предыдущее утверждение. Предположим, что задача *A* сводится к задаче *B*. Если *A* является NP-трудной, то полиномиально-временной алгоритм для нее опровергнет предположение, что P ≠ NP. Так вот. Полиномиально-временной алгоритм для *B* автоматически будет приводить к алгоритму для *A* (потому что *A* сводится к *B*) и тоже опровергнет предположение, что P ≠ NP. Это значит, что *B* тоже NP-трудная!

УПРОЩЕНИЯ СПОСОБСТВУЮТ ТРУДНОРЕШАЕМОСТИ

Если задача *A* сводится к задаче *B* и является NP-трудной, то *B* тоже является NP-трудной (рис. 19.3).

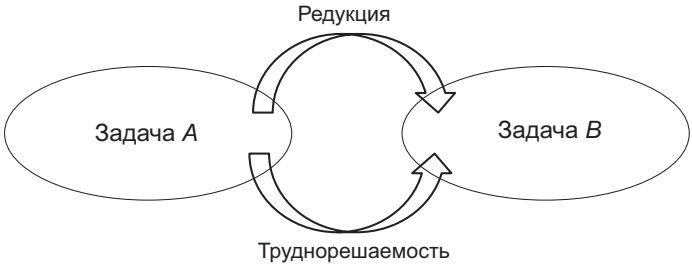


Рис. 19.3. Распространение труднорешаемости в от A к B: если задача A сводится к задаче B и A — труднорешаемая задача, то и B — тоже труднорешаемая

Вот удивительно простой двухшаговый рецепт доказательства NP-трудности:

КАК ДОКАЗАТЬ, ЧТО ЗАДАЧА ЯВЛЯЕТСЯ NP-ТРУДНОЙ

Чтобы доказать, что задача B NP-трудная, нужно:

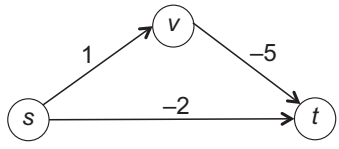
1. Выбрать NP-трудную задачу A.
2. Доказать, что A сводится к B.

Выполнение первого шага требует знания известных NP-трудных задач (см. главу 22). Второй шаг основан на ваших навыках поиска редукций между задачами (снова см. главу 22). Давайте поймем суть этого рецепта и вернемся к задаче о кратчайшем пути с одним истоком, где разрешены отрицательные длины ребер.

19.5.4. NP-трудность бесцикловых кратчайших путей

В *задаче о кратчайшем пути с одним истоком* входные данные состоят из ориентированного графа $G = (V, E)$, вещественнозначной длины ℓ_e для каждого ребра $e \in E$ и стартовой вершины $s \in V$. *Длина* пути — это сумма длин его ребер. Цель — вычислить для каждой возможной финальной вершины $v \in V$ минимальную длину $dist(s, v)$ ориентированного пути в G из s в v. (Если такого пути не существует, то $dist(s, v)$ определяется как $+\infty$.) Важно отме-

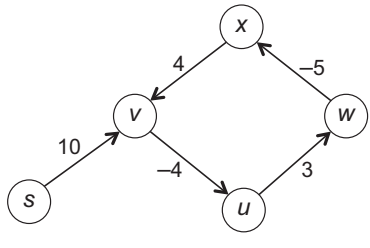
тить, что допускаются отрицательные длины ребер.[1, 2] Например, расстояния кратчайшего пути из *s* в графе



таковы: $dist(s, s) = 0$, $dist(s, v) = 1$ и $dist(s, t) = -4$.

Отрицательные циклы

Как определить расстояния кратчайшего пути в графе, подобном приведенному ниже?



Этот граф имеет *отрицательный цикл* — ориентированный цикл с отрицательной суммой длин ребер. В нем есть *s*-*v*-вершинный путь длиной 10 с одним переходом. Прикрепив циклический обход в конец, мы получим *s*-*v*-вершинный путь суммарной длины 8 с пятью переходами. Добавление второго обхода уменьшит совокупную длину до 6 и так далее. Допустив пути с циклами, мы получим граф, не имеющий кратчайшего *s*-*v*-вершинного пути.

---

[1] Помните, что пути в графе могут представлять собой абстрактные последовательности решений, а не что-то физически реализуемое. Например, чтобы вычислить прибыльную последовательность финансовых операций, включающих как покупку, так и продажу, нужен кратчайший путь в графе с длинами ребер, которые являются как положительными, так и отрицательными.

[2] В графах только с неотрицательными длинами ребер задача о кратчайшем пути с одним истоком может быть решена ослепительно быстро алгоритмом Дейкстры (см. главу 9 *Второй части*).

Задача о бесцикловом кратчайшем пути

Очевидная альтернатива заключается в запрете путей с циклами, чтобы каждая вершина посещалась не более одного раза.

ЗАДАЧА: БЕСЦИКЛОВЫЕ КРАТЧАЙШИЕ ПУТИ

Вход: ориентированный граф $G = (V, E)$, стартовая вершина $s \in V$ и вещественнозначная длина ℓ_e для каждого ребра $e \in E$.

Выход: для каждого $v \in V$ минимальная длина бесциклового s-v-вершинного пути в G (либо $+\infty$, если в G s-v-вершинного пути не существует).

К сожалению, эта версия задачи является NP-трудной.[1]

Теорема 19.1 (NP-трудность бесцикловых кратчайших путей). *Задача о бесцикловом кратчайшем пути является NP-трудной.*

О ЛЕММАХ, ТЕОРЕМАХ И ПРОЧЕМ

В математической записи самые важные высказывания имеют статус *теорем*. *Лемма* — это высказывание, которое помогает доказать теорему (так же подпрограмма помогает реализовать программу). *Следствие* — это вывод, который непосредственно вытекает из уже доказанного результата, например частного случая теоремы. Термин *утверждение* (в иных источниках он именуется пропозицией или высказыванием) мы используем для автономных технических высказываний, которые сами по себе не особенно важны.

[1] Это объясняет, почему алгоритм Беллмана — Форда (см. главу 18 *Третьей части*) наряду с любым другим полиномиально-временным алгоритмом поиска кратчайшего пути решает только частный случай задачи: при входных графах без отрицательных циклов, в которых кратчайшие пути автоматически являются бесцикловыми. Теорема 19.1 показывает, что при $P \neq NP$ ни один такой алгоритм не способен вычислить правильные расстояния бесцикловых кратчайших путей.

Задача об ориентированном гамильтоновом пути

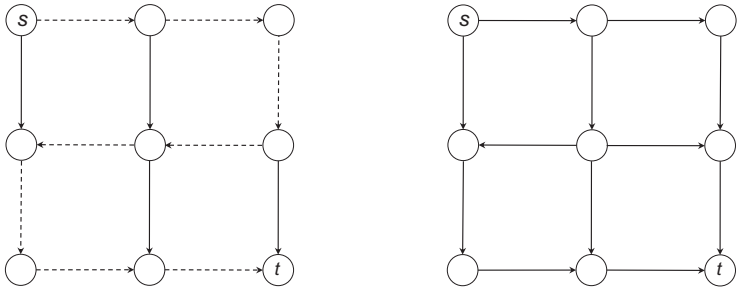
Докажем теорему 19.1, следуя рецепту из раздела 19.5.3. Для первого шага возьмем известную NP-трудную задачу об *ориентированном гамильтоновом пути* (DHP, directed Hamiltonian path).

ЗАДАЧА: ОРИЕНТИРОВАННЫЙ ГАМИЛЬТОНОВ ПУТЬ

Вход: ориентированный граф $G = (V, E)$, стартовая вершина $s \in V$ и финальная вершина $t \in V$.

Вывод: «да», если G содержит s-t-вершинный путь, посещающий каждую вершину $v \in V$ ровно один раз (именуемый *гамильтоновым* s-t-вершинным путем), и «нет» — в противном случае.

Например, из двух ориентированных графов



первый имеет гамильтонов s-t-вершинный путь (пунктирные ребра), а второй — нет.

Доказательство теоремы 19.1

Раздел 22.6 доказывает, что задача об ориентированном гамильтоновом пути является NP-трудной (с помощью рецепта из раздела 19.5.3). Пока же мы просто допустим ее NP-трудность и перейдем ко второму шагу рецепта,

в котором сведем NP-трудную задачу об ориентированном гамильтоновом пути к интересующей нас задаче о бесцикловых кратчайших путях.

Лемма 19.2 (упрощение задачи об ориентированном гамильтоновом пути к задаче о бесцикловых кратчайших путях). *Задача об ориентированном гамильтоновом пути сводится к задаче о бесцикловых кратчайших путях.*

Доказательство: как использовать подпрограмму для задачи о бесцикловых кратчайших путях для решения задачи об ориентированном гамильтоновом пути (вспомните рис. 19.1)? Представим экземпляр последней задачи, заданный ориентированным графом $G = (V, E)$, стартовой вершиной $s \in V$ и финальной вершиной $t \in V$. Предполагаемая подпрограмма бесциклового кратчайшего пути ожидает граф (мы предложим ей наш входной граф G) и стартовую вершину s (ту же самую). Она не готова к финальной вершине, но мы можем промолчать о t. Подпрограмма ожидает получить вещественнозначные длины ребер, поэтому мы ее обманем: дадим каждому ребру отрицательную длину, будто длинные пути (подобные гамильтонову s-t-вершинному пути) на самом деле короткие. Итак, редукция такова (рис. 19.4):

1. Назначить каждому ребру $e \in E$ длину $\ell_e = -1$.
2. Вычислить бесцикловые кратчайшие пути, используя предполагаемую подпрограмму, повторно взяв входной граф G и стартовую вершину s.
3. Если длина кратчайшего бесциклового пути из s в t равна $-(|V| - 1)$, то вернуть «да». В противном случае — вернуть «нет».

Чтобы доказать правильность этой редукции, покажем, что она возвращает «да» всякий раз, когда входной граф G содержит гамильтонов s-t-вершинный путь, и «нет» — в противном случае. В построенном образце бесцикловых кратчайших путей минимальная длина бесциклового s-t-вершинного пути равна произведению -1 и максимального числа переходов в бесцикловом s-t-вершинном пути исходного графа G. Бесцикловый s-t-вершинный путь использует $|V| - 1$ переходов, будучи гамильтоновым s-t-вершинным путем (с посещением всех $|V|$ вершин), и меньше переходов — в противном случае. Таким образом, если G имеет гамильтонов

s-*t*-вершинный путь, то расстояние бесциклового кратчайшего пути из *s* в *t* в построенном экземпляре равно $-(|V| - 1)$. В противном случае расстояние больше (но все еще отрицательное). В любом случае редукция возвращает правильный ответ. *Ч. Т. Д.*[1]

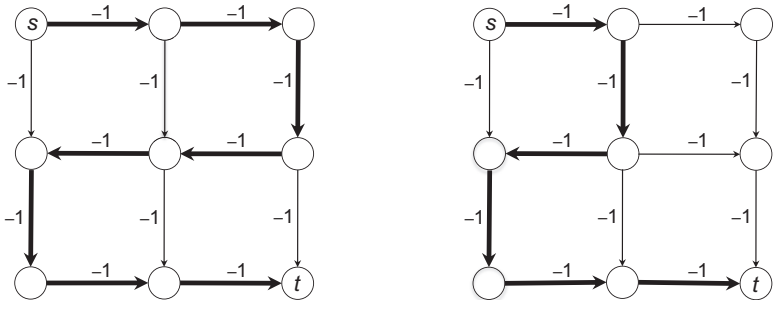


Рис. 19.4. Пример редукции в доказательстве леммы 19.2. Гамильтонов *s*-*t*-вершинный путь в первом графе приводит к бесцикловому *s*-*t*-вершинному пути длиной –8. Второй граф не имеет гамильтонова *s*-*t*-вершинного пути, и минимальная длина бесциклового *s*-*t*-вершинного пути равна –6

В силу двухшагового рецепта лемма 19.2 и NP-трудность задачи об ориентированном гамильтоновом пути доказывают теорему 19.1. В главе 22 приводится еще много примеров этого рецепта в действии.

19.5.5. Решение к тестовому заданию 19.3

Правильный ответ: (г). На первый взгляд кажется, что ответ — (в): каждый из не более $T_1(n)$ вызовов подпрограммы выполняет не более $T_3(n)$ операций, причем помимо этого алгоритм выполняет не более $T_2(n)$ операций с совокупным временем выполнения не более $T_1(n) \times T_3(n) + T_2(n)$.

Это рассуждение верно для большинства естественных редукций между задачами, включая три примера из раздела 19.5.2. Но технически редукция может, получив *n*-размерные входные данные, вызывать подпрограмму для *B* на

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

входах *больше n*. Представьте редукцию, которая в качестве входных данных берет граф и добавляет к нему дополнительные вершины или ребра, перед тем как вызвать подпрограмму для *B*. Чем это опасно? Поскольку редукция выполняет не более $T_2(n)$ операций вне вызовов подпрограммы, она успевает записать входные данные задачи *B* только размером не более $T_2(n)$. Таким образом, каждый вызов $T_1(n)$ задачи *B* требует не более $T_3(T_2(n))$ операций для совокупного времени выполнения $T_1(n) \times T_3(T_2(n)) + T_2(n)$.

19.6. Ошибки новичков и допустимые неточности

NP-трудность — это теоретическая тема, актуальная для практикующих дизайнеров алгоритмов и программистов. Помимо учебников и научных работ, ученые-компьютерщики часто позволяют себе вольности с точными математическими определениями для упрощения коммуникации. Некоторые типы неточностей характерны для невежественного новичка, в то время как другие — культурно приемлемы. Как их отличать? Сейчас расскажу.

ОШИБКА НОВИЧКА № 1

Думать, что «NP» означает «не полиномиальный» (not polynomial).

От вас не требуется помнить, что на самом деле означает «NP», коль скоро вы избегаете этой ошибки новичка.[1]

ОШИБКА НОВИЧКА № 2

Говорить, что задача является «NP-задачей» или «выполняется за NP» вместо «NP-трудной».

[1] И что же эта аббревиатура означает? Раздел 23.3 предоставляет исторический контекст, но на случай, если неизвестность вас терзает... — «недетерминированное полиномиальное время».

В разделе 23.3 я отметил, что выражения «NP-задача» или «выполняется за NP» на самом деле не так уж плохи.[1] Но не забывайте о слове «трудная» после «NP».

ОШИБКА НОВИЧКА № 3

Недооценивать NP-трудность, потому что NP-трудные задачи решаются на практике.

NP-трудность — не смертный приговор. Во многих практических приложениях NP-трудные задачи укрощены благодаря человеческим и вычислительным вложениям (глава 24). Но есть масса приложений, в которых вычислительные задачи были изменены или оставлены за бортом из-за проблем NP-трудности. (Естественно, люди сообщают о своих успехах в решении NP-трудных задач гораздо охотнее, чем о неудачах!) Если трудных на практике задач нет, почему так распространены эвристические алгоритмы? И как вообще работает электронная коммерция?[2]

ОШИБКА НОВИЧКА № 4

Думать, что достижения computer science спасут нас от NP-трудности.

Закон Мура и, соответственно, более крупные размеры входных данных только усугубляют проблему, а пропасть между полиномиальным и неполиномиальным временем выполнения становится огромной (раздел 19.3.2). Квантовые компьютеры улучшают исчерпывающий поиск, но не решают ни одну NP-трудную задачу за полиномиальное время (раздел 19.3.8).

---

[1] В частности, это означает, что если кто-то вручит вам решение на серебряном блюдечке с голубой каемочкой (например, завершенную головоломку судоку), то вы сможете проверить его валидность за полиномиальное время.

[2] Электронная коммерция опирается на криптосистемы, такие как RSA, безопасность которых зависит от труднорешаемости факторизации больших целых чисел. Полиномиально-временной алгоритм для любой NP-трудной задачи через редукцию наградит таким же алгоритмом факторизацию.

ОШИБКА НОВИЧКА № 5

Разрабатывать редукцию в неправильном направлении.

Сведение задачи A к задаче B распространяет NP-трудность от A к B, а не наоборот (сравните рис. 19.2 и 19.3). Всякий раз, когда вы считаете, что доказали NP-трудность задачи, вернитесь назад и трижды проверьте, что ваша редукция идет в направлении распространения труднорешаемости.

Допустимые неточности

Далее следуют три культурно приемлемых утверждения, которые не доказаны или технически неверны. Но смело можете их использовать.

ДОПУСТИМАЯ НЕТОЧНОСТЬ № 1

Исходить из истинности предположения о том, что $P \neq NP$.

Статус предположения, что $P \neq NP$, остается открытым, хотя большинство экспертов в него верят. Пока мы ждем, когда наше математическое понимание догонит нашу интуицию, многие рассматривают это предположение как закон природы.

ДОПУСТИМАЯ НЕТОЧНОСТЬ № 2

Использовать термины «NP-трудный» и «NP-полный» взаимозаменяемо.

NP-полнота — это специфический тип NP-трудности (раздел 23.3). Их алгоритмические следствия одинаковы.

ДОПУСТИМАЯ НЕТОЧНОСТЬ № 3

Объединять NP-трудность с требованием экспоненциального времени в худшем случае.

Вспомните сверхупрощенную интерпретацию NP-трудности из начала раздела 19.3. Технически она неточна (раздел 19.3.9), но соответствует мнению большинства экспертов.

ВЫВОДЫ

- Полиномиально-временной алгоритм — это алгоритм с временем выполнения $O(n^d)$ в худшем случае, где n обозначает размер входных данных, и d — это константа.
- Вычислительная задача поддается решению за полиномиальное время, если существует полиномиально-временной алгоритм, который решает ее правильно для каждого элемента входных данных.
- Сверхупрощенная теория NP-трудности уравнивает «легкость» задачи с возможностью ее решения за полиномиальное время. Для решения «трудной» задачи в худшем случае требуется экспоненциальное время.
- Неформально предположение, что $P \neq NP$, утверждает, что легче проверить готовое решение задачи, чем придумать свое решение с нуля.
- Предварительно вычислительная задача является NP-трудной, если полиномиально-временной алгоритм, который ее решает, опровергнет предположение, что $P \neq NP$.
- Полиномиально-временной алгоритм для любой NP-трудной задачи автоматически решит тысячи задач, которые сопротивлялись усилиям бесчисленных умов на протяжении десятилетий.
- NP-трудные задачи повсеместны.
- Чтобы приблизиться к решению NP-трудной задачи, пойдите на компромисс в отношении универсальности, правильности или скорости.
- Быстродействующие эвристические алгоритмы выполняются быстро, но не всегда правильно. Для их разработки особенно полезны парадигмы жадности и локального поиска.
- Динамическое программирование может улучшить исчерпывающий поиск решений нескольких NP-трудных задач.

- ✶ Решатели задач MIP и SAT являются передовой технологией для решения NP-трудных задач на практике.
- ✶ Задача *A* сводится к задаче *B*, если *A* может быть решена с помощью полиномиального числа вызовов подпрограммы, решающей *B*, и полиномиального объема дополнительной работы.
- ✶ Редукции распространяют легкорешаемость: если задача *A* сводится к задаче *B* и *B* может быть решена полиномиально-временным алгоритмом, то *A* тоже может быть решена полиномиально-временным алгоритмом.
- ✶ Редукции распространяют труднорешаемость в противоположном направлении: если задача *A* сводится к задаче *B* и *A* является NP-трудной, то *B* тоже является NP-трудной.
- ✶ Чтобы доказать, что задача *B* является NP-трудной, нужно: (1) выбрать NP-трудную задачу *A*; (2) доказать, что *A* сводится к *B*.

Задачи на закрепление материала

Задача 19.1 *(S)*. Предположим, что вычислительная задача *B*, которая вас интересует, является NP-трудной. Что из перечисленного является истинным? (Правильных ответов может быть несколько.)

а) NP-трудность — это смертный приговор. Не стоит пытаться решить экземпляры задачи *B*, релевантные для приложения.

б) Если босс недоволен тем, что вы не нашли полиномиально-временной алгоритм для *B*, можно ответить, что тысячи умов тоже не смогли решить *B*.

в) Не стоит пытаться разработать алгоритм, который гарантированно решит *B* правильно и за полиномиальное время для каждого возможного экземпляра задачи (если только вы явно не опровергаете предположение, что $P \neq NP$).

г) Поскольку парадигма динамического программирования полезна только для разработки точных алгоритмов, нет смысла пытаться применять ее к задаче *B*.

Задача 19.2 *(S)*. Какое из следующих утверждений истинно? (Правильных ответов может быть несколько.)

а) Задача о минимальном остовном дереве — легкорешаемая, поскольку число остовных деревьев графа является полиномиальным по числу n вершин и числу m ребер.

б) Задача о минимальном остовном дереве — легкорешаемая, поскольку существует не более m возможностей для суммарной стоимости остовного дерева графа.

в) Исчерпывающий поиск не решает задачу коммивояжера за полиномиальное время, потому что граф имеет экспоненциальное число туров.

г) Задача коммивояжера вычислительно трудно решается, потому что граф имеет экспоненциальное число туров.

Задача 19.3 *(S)*. Какое из следующих утверждений истинно? (Правильных ответов может быть несколько.)

а) Если предположение, что $P \neq NP$, является истинным, то NP-трудные задачи никогда не получится решить на практике.

б) Если предположение, что $P \neq NP$, является истинным, то никакую NP-трудную задачу не получится решить алгоритмом, который всегда является правильным и всегда выполняется за полиномиальное время.

в) Если предположение, что $P \neq NP$, является ложным, то NP-трудные задачи всегда можно решить на практике.

г) Если предположение, что $P \neq NP$, является ложным, то некоторые NP-трудные задачи поддаются решению за полиномиальное время.

Задача 19.4 *(S)*. Какие из следующих утверждений вытекают из предположения о том, что $P \neq NP$? (Правильных ответов может быть несколько.)

а) Каждый алгоритм, решающий NP-трудную задачу, выполняется за сверхполиномиальное время в худшем случае.

б) Каждый алгоритм, решающий NP-трудную задачу, работает за экспоненциальное время в худшем случае.

в) Каждый алгоритм, решающий NP-трудную задачу, всегда выполняется за сверхполиномиальное время.

г) Каждый алгоритм, решающий NP-трудную задачу, всегда выполняется за экспоненциальное время.

Задача 19.5 *(S)*. Если задача A сводится к задаче B, какое из следующих утверждений всегда истинно? (Правильных ответов может быть несколько.)

а) Если A поддается решению за полиномиальное время, то B тоже поддается решению за полиномиальное время.

б) Если B является NP-трудной, то A тоже является NP-трудной.

в) B также сводится к A.

г) B не может быть сведена к A.

д) Если задача B сводится к задаче C, то A тоже сводится к C.

Задача 19.6 *(S)*. Если предположение, что $P \neq NP$, истинно, какое из следующих утверждений о задаче о рюкзаке (раздел 19.4.2) является верным? (Правильных ответов может быть несколько.)

а) Частный случай, в котором все размеры предметов являются целыми положительными числами не более n^5, где n — это число предметов, может быть решен за полиномиальное время.

б) Частный случай, в котором все значения предметов являются целыми положительными числами не более n^5, где n — это число предметов, может быть решен за полиномиальное время.

в) Частный случай, в котором все значения и размеры предметов, а также вместимость рюкзака являются целыми положительными числами, может быть решен за полиномиальное время.

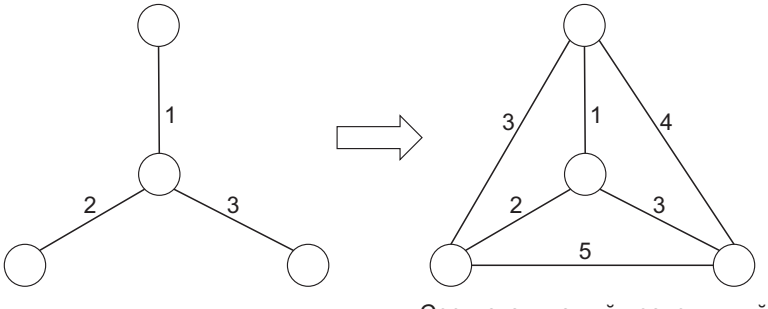
г) Для задачи о рюкзаке не существует полиномиально-временного алгоритма.

Задачи повышенной сложности

Задача 19.7 *(H)*. Входные данные в задаче о пути коммивояжера (TSPP, traveling salesman path problem) являются такими же, как и в задаче коммивояжера (TSP). Цель состоит в вычислении самого дешевого бесциклового пути, который посещает каждую вершину (то есть тур без его финального

ребра). Докажите, что задача о пути коммивояжера сводится к задаче коммивояжера и наоборот.

Задача 19.8 *(H)*. Эта задача описывает легкорешаемый частный случай задачи коммивояжера. Рассмотрим связный и ациклический граф $T = (V, F)$, в котором каждое ребро $e \in F$ имеет неотрицательную длину $a_e \geq 0$. Определите соответствующий древовидный экземпляр $G = (V, E)$ задачи коммивояжера, установив стоимость c_{vw} каждого ребра $(v, w) \in E$ равной длине $\sum_{e \in P_{vw}} a_e$ (уникального) v-w-вершинного пути P_{vw} в T. Например,



Связный ациклический граф

Соответствующий древовидный экземпляр задачи коммивояжера

Разработайте линейно-временной алгоритм, который, получив связный ациклический граф с неотрицательными длинами ребер, выводит самый дешевый тур коммивояжера соответствующего древовидного экземпляра. Докажите, что ваш алгоритм верен.

Задача по программированию

Задача 19.9. Реализуйте на своем любимом языке программирования алгоритм исчерпывающего поиска для задачи коммивояжера (см. тестовое задание 19.2). Дайте вашей реализации спин на экземплярах, в которых реберные стоимости выбираются независимо и равномерно случайным образом из множества $\{1, 2, \ldots, 100\}$. Насколько большим является размер входных данных (количество вершин), который ваша программа способна надежно обрабатывать менее чем за минуту? А что будет через час? (Тестовые случаи и наборы данных для сложных задач см. на веб-сайте www.algorithmsilluminated.org.)

20 Компромисс в отношении правильности: эффективные неточные алгоритмы

Алгоритм для NP-трудной задачи требует компромисса в отношении универсальности, правильности или скорости. Когда универсальность и скорость имеют решающее значение, особенно эффективны эвристические алгоритмы. Они универсальны и быстры, причем доказуемо или по меньшей мере эмпирически являются «приближенно правильными». Эта глава иллюстрирует на примерах использование разных методов: как новых (локальный поиск), так и старых (жадные алгоритмы). Тематические исследования касаются планирования (раздел 20.1), отбора персонала (раздел 20.2), анализа социальных сетей (раздел 20.3) и задачи коммивояжера (раздел 20.4).

20.1. Минимизация производственной продолжительности

Первое тематическое исследование касается задачи *планирования* и цели, состоящей в оптимизации некоторого целевого критерия при назначении операций совместным ресурсам. Ресурсом может быть компьютерный про-

цессор (где операции соответствуют работам), учебная аудитория (где операции соответствуют лекциям) или рабочий день (где операции соответствуют собраниям).

20.1.1. Определение задачи

В задачах планирования выполняемые операции обычно называются *работами*, а ресурсы — *машинами*. *План* служит для отнесения каждой работы к одной машине, которая будет его обрабатывать. Возможных задач планирования может быть много. Какую выбрать?

Предположим, что каждая работа j имеет известную *длину* ℓ_j — количество времени, необходимое для его обработки (например, продолжительность лекции или собрания). Распространенным целевым критерием в приложениях является такая расстановка работ, при которой все они завершаются как можно быстрее. Следующая *целевая функция* формализует эту идею путем назначения числового балла каждому плану и количественного выражения наших ожиданий:

ПРОДОЛЖИТЕЛЬНОСТЬ ПЛАНА

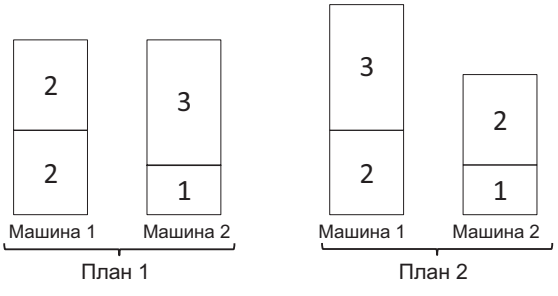
1. *Загрузка* машины в плане — это сумма длин закрепленных за ней работ.
2. *Продолжительность*[1] плана — это максимальное число загрузок машин.

Машинные загрузки и продолжительность — это одно и то же, независимо от того как упорядочены работы на каждой машине, поэтому в планах указывается не упорядочение работ, а только их закрепление за машинами.

[1] Продолжительность (makespan), или продолжительность изготовления, — это суммарная длина плана (до завершения всех работ). — *Примеч. пер.*

ТЕСТОВОЕ ЗАДАНИЕ 20.1

Какова производственная продолжительность следующих задач планирования? (Работы помечены их длинами.)



а) 4 и 3

б) 4 и 4

в) 4 и 5

г) 8 и 8

(Ответ и анализ решения см. в разделе 20.1.9.)

Таким образом, «оптимальный» план имеет минимальную производственную продолжительность. В тестовом задании 20.1 только первый план минимизирует производственную продолжительность.

ЗАДАЧА: МИНИМИЗАЦИЯ ПРОИЗВОДСТВЕННОЙ ПРОДОЛЖИТЕЛЬНОСТИ

Вход: множество n работ с положительными длинами ℓ_1, ℓ_2, …, ℓ_n и m идентичных машин.

Выход: закрепление работ за машинами, которое минимизирует производственную продолжительность.

Если работы — это части вычислительной операции, подлежащие параллельной обработке (как в средах MapReduce или Hadoop), то производственная продолжительность плана управляет завершением всего вычисления.

Минимизирование производственной продолжительности является NP-трудной задачей (см. задачу 22.10). Может ли существовать алгоритм, который является универсальным, быстрым и «почти правильным»?

20.1.2. Жадные алгоритмы

В решении многих вычислительных задач (как простых, так и сложных) жадные алгоритмы являются отличным стартом для мозгового штурма. Парадигма проектирования жадных алгоритмов такова (глава 13 *Третьей части*):

> **ЖАДНАЯ ПАРАДИГМА**
>
> Строить решение итеративно, посредством последовательности близоруких решений, и надеяться, что в конце концов все получится.

Две самых выигрышных черты жадных алгоритмов — их легко придумать и можно быстро реализовать. Недостаток же состоит в том, что они могут вернуть неверное решение. Но для NP-трудной задачи этот недостаток выглядит общим для всех быстрых алгоритмов: ни один полиномиально-временной алгоритм не будет правильным на всех входах (если $P \neq NP$)! Поэтому жадная парадигма особенно подходит для разработки эвристических алгоритмов для NP-трудных задач и играет главную роль в этой главе.

20.1.3. Алгоритм Грэма

Как выглядит жадный алгоритм для задачи о минимизации производственной продолжительности? Возможно, это однопроходный алгоритм, который безвозвратно назначает машинам по одной работе. Каким машинам? Оптимально

было бы закреплять работу за машиной с *наименьшей* текущей загрузкой. Это позволяет сделать алгоритм Грэма, или `Graham`.[1]

GRAHAM

Вход: множество $\{1, 2, \ldots, m\}$ машин и множество $\{1, 2, \ldots, n\}$ работ с положительными длинами $\ell_1, \ell_2, \ldots, \ell_n$.

Выход: закрепление работ за машинами.

```
// Инициализация
for i = 1 to m do
    J_i := Ø                    // работы, закрепленные за машиной i
    L_i := 0                    // текущая машинная загрузка i
// Главный цикл
for j = 1 to n do
    k := arg min_{i=1}^{m} L_i  // наименее загруженная машина[2]
    J_k := J_k ∪ {j}            // назначить текущую работу
    L_k := L_k + ℓ_j            // обновить загрузки
return J_1, J_2, ..., J_m
```

О ПСЕВДОКОДЕ

В этой книге алгоритмы описываются с использованием комбинации высокоуровневого псевдокода и обычного человеческого языка (как в этом разделе).

Я исхожу из того, что у вас есть навыки трансляции таких высокоуровневых (общих) описаний в рабочий код на вашем любимом языке програм-

[1] Предложен Рональдом Грэмом в статье «Границы многопроцессорных временных аномалий» («*Bounds on Multiprocessing Time Anomalies», Ronald L. Graham, SIAM Journal on Applied Mathematics*, 1969).

[2] Для ряда $a_1, a_2, \ldots, a_n$ вещественных чисел $\arg\min_{i=1}^{n} a_i$ обозначает индекс наименьшего числа. (Если несколько чисел претендуют на то, чтобы быть наименьшим, то $\arg\min_{i=1}^{n} a_i$ следует интерпретировать как произвольный разрыв связей между ними.) Функция $\arg\max_{i=1}^{n} a_i$ определяется схожим образом.

мирования. Несколько других книг и ресурсов в интернете предлагают конкретные реализации различных алгоритмов на определенных языках программирования.

Во-первых, преимущество в предпочтении высокоуровневых описаний над реализациями, специфичными для конкретного языка программирования, заключается в их гибкости: хотя я исхожу из вашей осведомленности в *каком-то* языке программирования, меня не интересует, что это за язык. Во-вторых, этот подход способствует пониманию алгоритмов на глубоком и концептуальном уровне, не обремененном деталями низкого уровня. Опытные программисты и специалисты computer science обычно мыслят и обмениваются информацией об алгоритмах на столь же высоком уровне.

Тем не менее ничто не может заменить детального понимания алгоритма, которое вытекает из разработки своей собственной рабочей программы. Настоятельно рекомендую вам реализовать на любимом языке программирования столько алгоритмов из этой книги, насколько у вас хватит времени. (Это также будет отличным поводом улучшить свои навыки программирования!)

Для развития ваших навыков программирования воспользуйтесь задачами по программированию в конце главы и соответствующими тестовыми примерами.

20.1.4. Время работы

Так ли хорош `Graham`? Как это обычно бывает с жадными алгоритмами, его время работы легко анализируется. Если вычисление argmin в строке 5 реализовано исчерпывающим поиском среди m возможностей, то каждая из n итераций главного цикла выполняется за $O(m)$-*е* время (например, если J_i реализовано в виде связанных списков). Поскольку вне главного цикла выполняется только $O(m)$-я работа, эта прямая реализация приводит к времени выполнения $O(mn)$.

Читатели, имеющие опыт работы со структурами данных, распознают возможность для улучшения. Алгоритм повторяет вычисление минимума, поэтому у вас в голове должна зажечься лампочка: он взывает к кучевой структуре

данных![1] Поскольку куча сокращает время выполнения вычисления минимума с линейного до логарифмического, ее использование здесь приводит к ослепительно быстрой $O(n \log m)$-временной реализации алгоритма. Задача 20.6 просит вас заполнить детали.

20.1.5. Приближенная правильность

А что насчет построения производственной продолжительности плана алгоритмом `Graham`?

ТЕСТОВОЕ ЗАДАНИЕ 20.2

Есть пять машин и список из двадцати работ длиной 1 каждая, с последующей особой работой длиной 5. Какова производственная продолжительность плана, выводимая `Graham`, и какова наименьшая возможная производственная продолжительность плана выполнения этих работ?

а) 5 и 4

б) 6 и 5

в) 9 и 5

г) 10 и 5

(Ответ и анализ решения см. в разделе 20.1.9.)

Как демонстрирует тестовое задание 20.2, `Graham` не всегда выдает оптимальный план. И это неудивительно, учитывая, что задача является NP-трудной и алгоритм является полиномиально-временным. (Если бы алгоритм всегда был правильным, то мы бы опровергли предположение, что $P \neq NP$!) Тем не менее пример должен заставить вас задуматься. Есть ли более сложные входные данные, для которых `Graham` работает еще хуже? К счастью, пример из тестового задания 20.2 настолько плох, насколько это возможно.

---

[1] Например, см. главу 10 *Второй части*.

Теорема 20.1 (`Graham`: приближенная правильность). *Производственная продолжительность плана, выводимая* `Graham`*, никогда не превышает в 2 – 1/m раз минимально возможную производственную продолжительность, где m обозначает число машин.*[1, 2]

Таким образом, `Graham` является «приближенно правильным» алгоритмом для задачи о минимизации производственной продолжительности. Рассуждайте о теореме 20.1 как о страховом полисе. Даже в сценарии Судного дня с надуманными входными данными, как в тестовом задании 20.2, производственная продолжительность плана алгоритма не более чем вдвое превышает то, что вы получили бы при исчерпывающем поиске. В случае более реалистичных входных данных следует ожидать, что `Graham` превысит расчетную величину и достигнет производственной продолжительности, гораздо более близкой к минимально возможной (см. задачу 20.1).

В следующем разделе приводится полное доказательство теоремы 20.1. Ограниченный по времени или математикобоязненный читатель может предпочесть краткое, но точное пояснение:

ПОЯСНЕНИЕ ДЛЯ ТЕОРЕМЫ 20.1

1. *Наименьшая* загрузка машины — это машинная загрузка, не превышающая загрузку в идеально сбалансированном плане, который, в свою очередь, не увеличивает минимально возможную производственную продолжительность (поскольку сценарием для наилучшего случая является идеально сбалансированный план).
2. В силу жадного критерия алгоритма `Graham` наибольшие и наименьшие машинные загрузки различаются не более чем на длину одной работы, которая, в свою очередь, не превышает минимально возможную производственную про-

---

[1] Для того чтобы применить плохой пример из тестового задания 20.2 для произвольного числа машин m, используйте $m(m – 1)$ работ длиной 1 с последующей одной работой длиной m.

[2] Множитель $2 – 1/m$ иногда называют *аппроксимационным коэффициентом* алгоритма, который в свою очередь называют *алгоритмом* $(2 – 1/m)$*-й аппроксимации*.

должительность (поскольку каждая работа должна все-таки когда-то закончиться).

3. Отсюда наибольшая машинная загрузка на выходе из алгоритма не более чем в два раза больше минимально возможной производственной продолжительности.

20.1.6. Доказательство теоремы 20.1

Перед нами работы с длинами $\ell_1, \ell_2, \ldots, \ell_n$ и числом m машин. Прямое сравнение минимально возможной производственной продолжительности M^* и производственной продолжительности M плана, выводимой `Graham`, будет запутанным. Вместо этого обратимся к двум легко вычисляемым нижним границам по M^* — максимальной длине работы и средней машинной загрузке, — они соотносятся с M и в итоге показывают, что $M \leq (2 - 1/m)M^*$.

Первая нижняя граница по M^* проста: каждая работа должна закончиться, поэтому невозможно достичь производственной продолжительности меньше длины работы.

Лемма 20.2 (нижняя граница № 1 по оптимальной производственной продолжительности). *Если M^* обозначает минимальную производственную продолжительность любого плана и j — работу, то*

$$M^* \geq \ell_j. \tag{20.1}$$

В общем случае в каждом плане каждая работа j закрепляется ровно за одной машиной i и вносит свой вклад ℓ_j в ее загрузку L_i. Таким образом, в каждом плане сумма машинных загрузок равна сумме длин работ: $\sum_{i=1}^{m} L_i = \sum_{j=1}^{n} \ell_j$. В совершенном плане каждая машина имеет идеальную загрузку — точную долю $1/m$ от общей суммы (то есть $\frac{1}{m}\sum_{j=1}^{n} \ell_j$). В любом другом плане некоторые машины имеют более чем идеальную загрузку, другие же — менее идеальную. Например, в тестовом задании 20.1 в первом плане обе машины имеют идеальные загрузки, тогда как во втором плане одна перегружена, а другая недогружена.

Вторая нижняя граница по M^* вычисляется на основании факта, что каждый план имеет машину с загрузкой, равной идеальной загрузке или превышающей ее.

Лемма 20.3 (нижняя граница № 2 по оптимальной производственной продолжительности). *Если M^* обозначает минимальную производственную продолжительность любого плана, то*

$$M^* \geq \underbrace{\frac{1}{m}\sum_{j=1}^{n} \ell_j}_{\text{идеальная загрузка}}. \tag{20.2}$$

Напоследок ограничим сверху производственную продолжительность M плана алгоритма `Graham` с точки зрения двух нижних границ, введенных в леммах 20.2 и 20.3. Обозначим через i машину с самой большой загрузкой в этом плане (то есть с загрузкой $L_i = M$) и j — закрепленную за ней заключительную работу (рис. 20.1(а)). Перемотаем алгоритм назад, к моменту непосредственно перед закреплением работы j, и обозначим через $\hat{L}_i$ загрузку машины i в то время. Новая и заключительная загрузка L_i машины (и, следовательно, производственная продолжительность M) составляет $\ell_j + \hat{L}_i$.

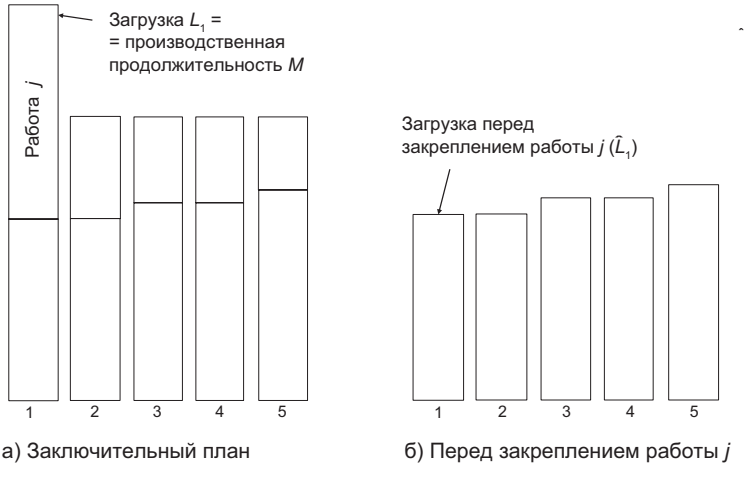


Рис. 20.1. Наиболее загруженная машина была наименее загруженной непосредственно перед закреплением за ней заключительной работы

Насколько большой могла бы быть $\hat{L}_i$? В силу жадного критерия алгоритма `Graham` машина i была в то время самой недозагруженной (рис. 20.1(б)). Если бы работы $\{1, 2, \ldots, j-1\}$ перед j были бы идеально сбалансированы по всем машинам, то все машинные загрузки в то время были бы равны $\frac{1}{m}\sum_{h=1}^{j-1}\ell_h$. В противном случае самая легкая загрузка $\hat{L}_i$ была бы еще меньше. В любом случае заключительная производственная продолжительность $M = \ell_j + \hat{L}_i$ не превышает

$$\ell_j + \frac{1}{m}\sum_{h=1}^{j-1}\ell_h \le \ell_j + \frac{1}{m}\sum_{h=j}^{n}\ell_h,$$

где в правой части мы для удобства внесли недостающие (положительные) члены ℓ_{j+1}/m, ℓ_{j+2}/m, ..., ℓ_n/m. Перенеся ℓ_j/m из первого члена во второй, мы можем записать:

$$M \le \underbrace{\left(1-\frac{1}{m}\right)\times\ell_j}_{\le\left(1-\frac{1}{m}\right)M^* \text{ в силу (20.1)}} + \underbrace{\frac{1}{m}\sum_{h=1}^{n}\ell_h}_{\le M^* \text{ в силу (20.2)}} \le \left(2-\frac{1}{m}\right)\times M^*, \qquad (20.3)$$

причем второе неравенство следует из леммы 20.2 (чтобы ограничить первый член) и леммы 20.3 (чтобы ограничить второй член). *Ч. Т. Д.*

20.1.7. Сначала самое длительное время обработки

Страховой полис, такой как гарантия приближенной правильности в теореме 20.1, обнадеживает, но мы по-прежнему должны думать: можно ли добиться лучшего и разработать другой быстрый эвристический алгоритм, который будет «еще менее неправильным», предложив страховой полис с более низким отчислением? Попробуем использовать уже знакомый бесплатный примитив.

БЕСПЛАТНЫЕ ПРИМИТИВЫ

Алгоритм с линейным или почти линейным временем выполнения можно рассматривать как примитив, который «бесплатно» доступен, потому

что объем его вычислений едва превышает объем, необходимый только для чтения входных данных. Если такой примитив соответствует вашей задаче, почему бы им не воспользоваться? Например, чтобы отсортировать данные на шаге предварительной обработки. Одна из целей этой серии книг — снабдить ваш алгоритмический инструментарий как можно большим числом бесплатных примитивов, готовых к применению в любой момент.

Что не так с `Graham` в тестовом задании 20.2? Он идеально уравновешивает работы длиной 1, не оставляя ни одного подходящего места для работы длиной 5. Если бы алгоритм сначала рассматривал работу длиной 5, то все остальные работы встали бы на свои места. В общем случае вторая часть интуиции теоремы 20.1 (с. 70) и заключительный шаг ее доказательства (неравенство (20.3)) выступают за то, чтобы последняя работа, закрепляемая за наиболее загруженной машиной (работа j в (20.3)), выполнялась как можно раньше. Это предполагает алгоритм сначала самого длительного времени обработки (`LPT`, longest processing time first), также предложенный Грэмом.

LPT

Вход / выход: как в `Graham` (с. 67).

Отсортировать работы от самой длинной до самой короткой;

выполнить `Graham` на отсортированных работах.

Первый шаг можно реализовать со временем $O(n \log n)$ для n работ, используя, например, алгоритм `MergeSort`. Если `Graham` реализован с привлечением куч (задача 20.6), то оба этих шага выполнятся почти за линейное время.[1]

[1] `Graham` является примером онлайн-алгоритма: его можно использовать даже в том случае, если работы материализуются одна за другой и немедленно вставляются в план. `LPT` не является онлайн-алгоритмом: он требует предварительного знания о всех работах, чтобы сортировать их по длине.

ТЕСТОВОЕ ЗАДАНИЕ 20.3

Есть пять машин, три работы длиной 5 и по две работы с длинами 6, 7, 8 и 9. Какова производственная продолжительность плана, выводимая `LPT`, и какова наименьшая возможная производственная продолжительность плана выполнения этих работ?

а) 16 и 15

б) 17 и 15

в) 18 и 15

г) 19 и 15

(Ответ и анализ решения см. в разделе 20.1.9.)

Опять же, поскольку задача о минимизации производственной продолжительности является NP-трудной, а `LPT` выполняется за полиномиальное время, примеры демонстрируют неоптимальность алгоритма. Но обеспечивает ли он более качественный страховой полис, чем `Graham`?

Теорема 20.4 (`LPT`: приближенная правильность). *Производственная продолжительность плана, выводимая* `LPT`*, всегда не более чем в 3/2 – 1/2m раза больше минимально возможной производственной продолжительности, где m — это число машин.*

Интуитивно сортировка работ уменьшает возможный ущерб, наносимый одиночной работой (разность между наибольшей и наименьшей машинными загрузками) с M^* (минимально возможной производственной продолжительности) до $M^*/2$.

Внимательный читатель, возможно, заметил расхождение между плохим примером в тестовом задании 20.3 (с раздутием производственной продолжительности до $19/15 \approx 1{,}267$) и подтверждением теоремы 20.4 (которая для $m = 5$ обещает раздутие не выше $14/10 = 1{,}4$). С помощью нескольких дополнительных аргументов (изложенных в задаче 20.7) гарантия в теореме 20.4 может быть уточнена с $3/2 – 1/2m$ до $4/3 – 1/3m$. Следовательно, примеры,

предложенные тестовым заданием 20.3, так же плохи, как и использованные для `LPT`. И, как и в случае с `Graham`, вы должны ожидать, что `LPT` превысит расчетную величину для более реалистичных входных данных.[1]

20.1.8. Доказательство теоремы 20.4

Доказательство теоремы 20.4 следует за доказательством теоремы 20.1, причем улучшение обеспечивается вариантом леммы 20.2, полезным при сортировке работ от самой длинной до самой короткой.

Лемма 20.5 (вариант нижней границы № 1). *Если M^* обозначает минимальную производственную продолжительность любого плана и j — работу, которая не входит в число m самых длинных (произвольно разрывая связи), то*

$$M^* \geq 2\ell_j. \tag{20.4}$$

Доказательство: в силу принципа голубиных ячеек каждый план должен закреплять две из $m + 1$ самых длинных работ за одной машиной.[2] Следовательно, минимально возможная производственная продолжительность как минимум вдвое больше длины самой длинной работы из $(m + 1)$ и составляет по меньшей мере $2\ell_j$. *Ч. Т. Д.*

Теперь перейдем к доказательству теоремы.

Доказательство теоремы 20.4: как и в доказательстве теоремы 20.1, обозначим i машину с наибольшей загрузкой в плане алгоритма `LPT` и j — закрепленную за ней заключительную работу (рис. 20.1(а)). Предположим, что

---

[1] Существуют более сложные алгоритмы с улучшенными гарантиями приближенной правильности. Технически они выполняются за полиномиальное время, но на практике неоправданно медленны. Если вам нужно минимизировать производственную продолжительности, то алгоритм `LPT` станет отличной отправной точкой.

[2] *Принцип голубиных ячеек* (pigeonhole principle) — это очевидный факт, что независимо от того, как вы размещаете $n + 1$ голубей в n ячеек, всегда будет иметься ячейка как минимум с двумя голубями.

хотя бы одна другая работа закреплена за машиной i (перед j) — иначе будет нечего доказывать.[1]

Алгоритм закрепляет каждую из первых m работ за другой машиной (каждая из них в этот момент пуста). Следовательно, работа j не входит в m первых работ. В силу жадного критерия алгоритма `LPT` работа j также не входит в m самых длинных работ. Лемма 20.5 говорит, что $\ell_j \le M^*/2$, где M^* — это минимально возможная производственная продолжительность. Подключение этой улучшенной границы (относительно леммы 20.2) в неравенство (20.3) показывает, что производственная продолжительность M, достигаемая `LPT`, удовлетворяет выражению

$$M \le \underbrace{\left(1-\frac{1}{m}\right)\times \ell_j}_{\le\left(1-\frac{1}{m}\right)\times\left(M^*/2\right)\text{ в силу (20.4)}} + \underbrace{\frac{1}{m}\sum_{k=1}^{n}\ell_k}_{\le M^*\text{ в силу (20.2)}} \le \left(\frac{3}{2}-\frac{1}{2m}\right)\times M^*.$$

Ч. Т. Д.

20.1.9. Решения к тестовым заданиям 20.1–20.3

Решение к тестовому заданию 20.1

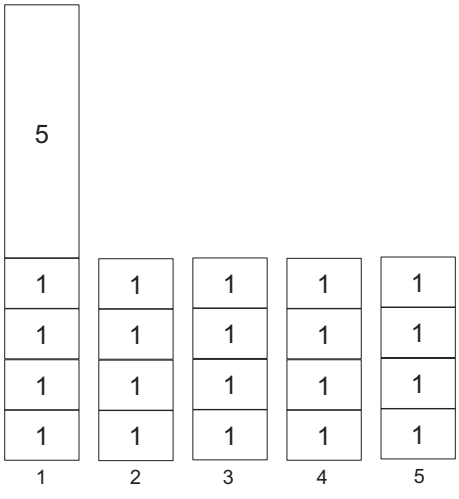
Правильный ответ: (в). Машинные загрузки составляют 2 + 2 = 4 и 1 + 3 = 4 в первом плане и 2 + 3 = 5 и 1 + 2 = 3 во втором. Поскольку производственная продолжительность представляет собой наибольшую машинную загрузку, эти планы имеют производственные продолжительности соответственно 4 и 5.

Решение к тестовому заданию 20.2

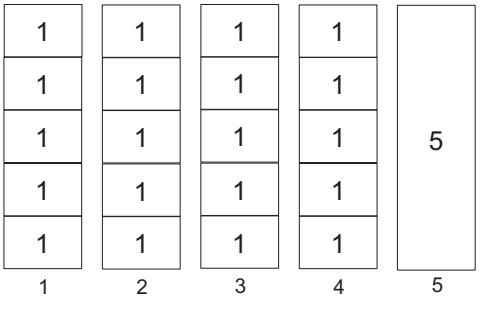
Правильный ответ: (в). `Graham` расставляет первые двадцать работ равномерно по всем машинам (по четыре работы длиной 1 на каждой). Независимо от того, как он планирует заключительную работу длиной 5, он застрял с производственной продолжительностью 9:

---

[1] Если j является единственной закрепленной за машиной i работой, то план алгоритма имеет производственную продолжительность ℓ_j, и никакой другой план не может быть лучше (в силу леммы 20.2).

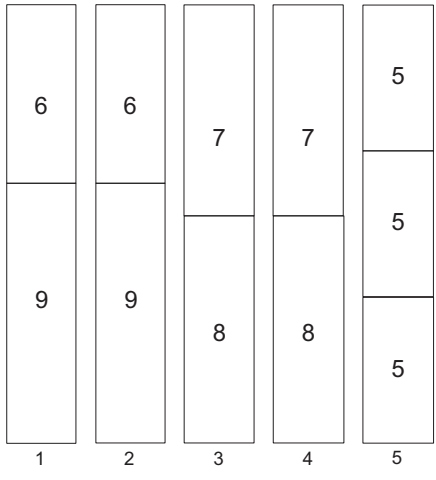


Между тем, зарезервировав одну машину для большой работы и разбив двадцать малых работ равномерно между оставшимися четырьмя машинами, он создает идеально сбалансированный план с производственной продолжительностью 5:

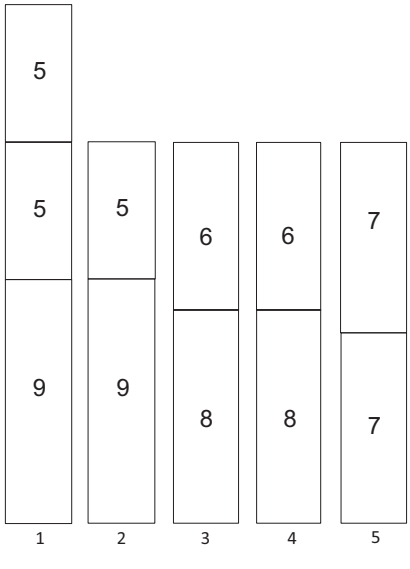


Решение к тестовому заданию 20.3

Правильный ответ: (г). Оптимальный план идеально сбалансирован, причем три работы длиной 5 закреплены за общей машиной, а каждая машина поочередно получает работу длиной то 9 или 6, то 8 или 7:



Производственная продолжительность плана — 15. Но у всех машин уже есть загрузка 14, когда наступает время с помощью `LPT` закрепить последнюю работу длиной 5. Поэтому план застревает с заключительной производственной продолжительностью 19:



20.2. Максимальный охват

Представьте, что вам поручили собрать коллектив: может, в компанию, чтобы завершить проект, или в спортивную лигу — отыграть сезон. Вы можете нанять только ограниченное число людей, каждый из которых обладает навыками — знает языки программирования или занимает определенные позиции на игровом поле. Вам нужна разнообразная команда, обладающая как можно бóльшим количеством разных навыков. Кого выбрать?

20.2.1. Определение задачи

В задаче *максимального охвата* на входе есть m подмножеств $A_1, A_2, \ldots, A_m$ универсального множества U и бюджет k. Универсальное множество U соответствует всем возможным навыкам членов команды, а каждое подмножество A_i соответствует одному кандидату, причем элементы подмножества указывают на навыки. Нужно выбрать k подмножеств так, чтобы максимизировать их *охват* — число отдельных элементов универсального множества, которые они содержат. В задаче о наборе кадров охват соответствует числу навыков, которыми обладает команда.

ЗАДАЧА: МАКСИМАЛЬНЫЙ ОХВАТ

Вход: коллекция $A_1, A_2, \ldots, A_m$ подмножеств универсального множества U и положительное целое число k.

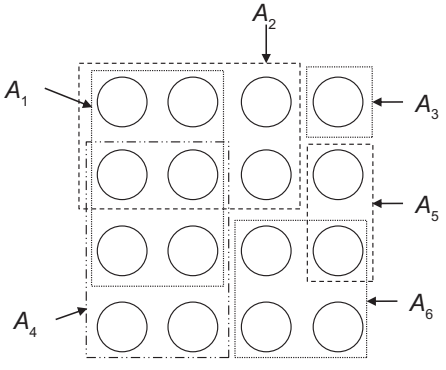
Выход: выборка $K \subseteq \{1, 2, \ldots, m\}$ из k индексов с целью максимизировать охват $f_{\text{cov}}(K)$ соответствующих подмножеств, где:

$$f_{\text{cov}}(K) = \left| \cup_{i \in K} A_i \right| \tag{20.5}$$

Например:

ТЕСТОВОЕ ЗАДАНИЕ 20.4

Рассмотрим универсальное множество U с 16 элементами и 6 подмножествами в нем:



Каков самый большой охват, достигнутый четырьмя из подмножеств?

а) 13

б) 14

в) 15

г) 16

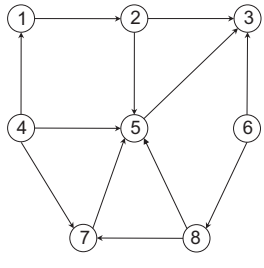
(Ответ и анализ решения см. в разделе 20.2.8.)

Задачи максимального охвата коварны тем, что имеют наложения между подмножествами. Например, некоторые навыки могут встречаться часто (охватываться многими подмножествами), а другие — редко (охватываться лишь несколькими подмножествами). Идеальное подмножество является большим с малым числом избыточных элементов — член команды, одаренный разными навыками.

20.2.2. Дальнейшие применения

Задачи о максимальном охвате повсеместны. Представьте, что нужно расположить k новых пожарных частей на расстоянии полутора километров от как можно большего числа жителей. Здесь элементы универсального множества соответствуют жителям, каждое подмножество соответствует возможному местоположению пожарной части, а элементы подмножества соответствуют жителям, которые живут в пределах полутора километров от этого местоположения.

Сложнее представить сбор людей на концерт. Вы приглашаете k друзей, но они приводят еще *своих* друзей, а те — своих и так далее. Эту задачу можно визуализировать с помощью ориентированного графа, в котором вершины соответствуют людям, а ребро, ориентированное из v в w, обозначает, что w придет на концерт, только если туда явится v. Например, в графе



приглашение друга 1 вызовет присутствие четырех человек (1, 2, 3 и 5). Друг 6 появится только, если будет вызван напрямую, и в этом случае четыре других (3, 5, 7 и 8) последуют его примеру.

Это задача о максимальном охвате. Элементы универсального множества соответствуют людям — вершинам графа. На одного человека приходится одно подмножество с указанием, кто последует за ним на концерт, — вершины, достижимые по ориентированному пути из данной вершины. Так суммарная посещаемость, активируемая за счет вызова k человек, точно соответствует охвату, достигаемому k подмножествами.

20.2.3. Жадный алгоритм

Задача о максимальном охвате является NP-трудной (см. задачу 22.8). Чтобы не отказываться от скорости, рассмотрим эвристические алгоритмы. Жадные

алгоритмы, которые близоруко отбирают подмножества одно за другим, подходят для старта.

Задача легко решается, когда можно взять только одно самое большое подмножество ($k = 1$). Предположим, что $k = 2$, и вы взяли самое большое из подмножеств — A. Каким будет второе подмножество? Сейчас важны элементы в подмножестве, *еще не охваченном A*, поэтому разумным жадным критерием будет максимизация числа новых охватываемых элементов. Распространение этой идеи на произвольный бюджет k приводит к известному жадному алгоритму для задачи о максимальном охвате, в котором функция охвата f_{cov} определяется как в (20.5).[1]

GREEDYCOVERAGE

Вход: подмножества $A_1, A_2, \ldots, A_m$ универсального множества U и положительное целое число k.

Выход: множество $K \subseteq \{1, 2, \ldots, m\}$ из k индексов.

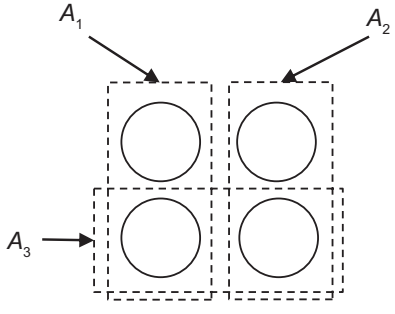
```
1 K := Ø                    // индексы выбранных множеств
2 for j = 1 to k do         // выбирать множества одно за другим
    // жадно увеличивать охват
    // (произвольно разрывать связи)
3   i* := arg max_{i=1}^{m} [ f_cov(K ∪ {i}) − f_cov(K) ]
4   K := K ∪ {i*}
5 return K
```

Для простоты `argmax` в строке 3 рассматривает все подмножества. Также он мог бы ограничить свое внимание подмножествами, еще не выбранными в предыдущих итерациях.

[1] Впервые проанализирована Жераром П. Корнежолем, Маршаллом Л. Фишером и Джорджем Л. Немхаузером в работе «Расположение банковских счетов для оптимизации потока: аналитическое исследование точных и приближенных алгоритмов» *(«Location of Bank Accounts to Optimize Float: An Analytic Study of Exact and Approximate Algorithms», Gérard P. Cornuéjols, Marshall L. Fisher, and George L. Nemhauser, Management Science,* 1977).

20.2.4. Плохие примеры для GreedyCoverage

Алгоритм жадного охвата, или `GreedyCoverage`, легко реализуется за полиномиальное время.[1] Поскольку задача о максимальном охвате является NP-трудной, будьте готовы к примерам, для которых он выводит неоптимальное решение (иначе он опровергнет предположение, что P ≠ NP). Вот один из них:



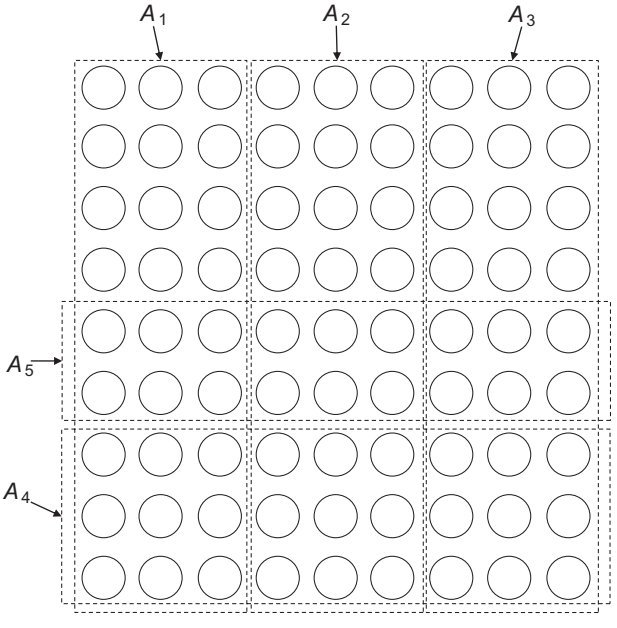
Предположим, что $k = 2$. Оптимальным решением будет взять подмножества A_1 и A_2 для охвата всех четырех элементов. Но жадный алгоритм (с произвольным разрывом связей) вполне может выбрать подмножество A_3 на первой итерации и застрять, выбрав A_1 или A_2 на второй итерации, охватив только три из четырех элементов.

Есть ли более худшие примеры для `GreedyCoverage`? Да, для больших бюджетов k.

ТЕСТОВОЕ ЗАДАНИЕ 20.5

Рассмотрим универсальное множество из 81 элемента и его 5 подмножеств:

[1] Попробуйте вычислить argmax в строке 3 путем исчерпывающего поиска в m подмножествах, вычисляя дополнительный охват $f_{\text{cov}}(K \cup \{i\}) - f_{\text{cov}}(K)$, предоставляемый подмножеством A_i, и используя один проход над элементами A_i. Прямая реализация приведет ко времени выполнения $O(kms)$, где s обозначает максимальный размер подмножества (не более $|U|$).



Определите для $k = 3$: (1) максимально возможный охват; (2) наименьший возможный охват выходных данных `GreedyCoverage` (с произвольным разрывом связей):

а) 72 и 60

б) 81 и 57

в) 81 и 60

г) 81 и 64

(Ответ и анализ решения см. в разделе 20.2.8.)

При $k = 2$ `GreedyCoverage` способен уловить 75 % охватываемых элементов, а при $k = 3$ может дойти до $57/81 = 19/27 \approx 70{,}4$ %. Насколько все может ухудшиться? Задача 20.8 (а) просит вас расширить шаблон на все положительные целые числа k, показав:[1]

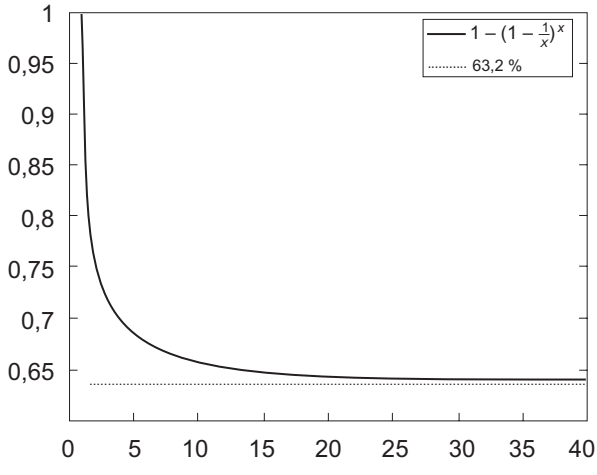
[1] Опора на произвольный разрыв связей для этих примеров удобна, но не существенна (см. задачу 20.8 (б)).

Утверждение 20.6 (плохие примеры для `GreedyCoverage`). *Для каждого положительного целого числа k существует пример задачи о максимальном охвате, в котором:*

(a) существует k подмножеств, которые охватывают все универсальное множество;

(b) при произвольном разрыве связей `GreedyCoverage` *может охватить только* $1 - (1 - 1/k)^k$ *долю элементов.*[1]

Чтобы понять выражение с одной переменной, постройте график. Вы увидите, как функция $1 - (1 - 1/x)^x$ уменьшается, но приближается к асимптоте примерно на 63,2 %:



Дело в том, что $1 - x$ очень хорошо аппроксимируется с помощью e^{-x} при x, близком к 0 (наглядно это демонстрируют график и тейлорово разложение e^{-x}). Отсюда выражение $1 - (1 - 1/k)^k$ стремится к $1 - (e^{-1/k})^k = 1 - 1/e \approx 0{,}632$ по мере того, как k стремится к бесконечности.[2]

---

[1] Обратите внимание, что $1 - (1 - 1/k)^k = 1 - (1/2)^2 = 3/4$ при $k = 2$ и $1 - (2/3)^3 = 19/27$ при $k = 3$.

[2] Здесь $e = 2{,}718...$ обозначает число Эйлера.

20.2.5. Приближенная правильность

Что делает странное число наподобие $1 - 1/e$ в непосредственной близости от сверхпростого алгоритма `GreedyCoverage`? Может быть, это артефакт примеров из тестового задания 20.5 и утверждения 20.6? Совсем наоборот: следующая гарантия приближенной правильности доказывает, что эти примеры являются худшими для `GreedyCoverage`, поскольку он неразрывно связан с числами $1 - (1 - 1/k)^k$ и $1 - 1/e$.[1, 2]

Теорема 20.7 (`GreedyCoverage`: приближенная правильность). *Охват решения, выдаваемого* `GreedyCoverage`*, всегда составляет по меньшей мере $1 - (1 - 1/k)^k$-ю долю максимально возможного охвата, где k — это число выбранных подмножеств.*

`GreedyCoverage` гарантированно охватывает элементы в размере хотя бы 75 % от оптимального решения при $k = 2$, хотя бы 70,4 % при $k = 3$ и хотя бы 63,2 % независимо от величины k. Как и в случае с теоремами 20.1 и 20.4, теорема 20.7 является страховым полисом, который ограничивает ущерб в худшем случае. Для более реалистичных входных данных указанный алгоритм, скорее всего, превысит расчетную величину и достигнет более высоких процентов.

20.2.6. Ключевая лемма

Чтобы развить вашу интуицию в отношении теоремы 20.7, вернемся к тестовому заданию 20.5. Почему `GreedyCoverage` не нашел оптимального решения? На первой итерации он мог выбрать любое из трех подмножеств в оптимальном решении (A_1, A_2 или A_3). К сожалению, алгоритм был обманут четвертым, столь же большим подмножеством A_4, охватывающим 27 элементов. Во

---

[1] Если P ≠ NP, никакой полиномиально-временной алгоритм (жадный или иной) не гарантирует охват больше $1 - 1/e$-й доли максимально возможного решения по мере увеличения k. См. статью Уриэла Фейга «Порог ln n для аппроксимирования покрытия множества» (*«A Threshold of ln n for Approximating Set Cover»*, *Uriel Feige*, *Journal of the ACM*, 1998). Указанный факт дает сильное теоретическое обоснование принимать `GreedyCoverage` за отправную точку в решении задачи максимального охвата. Из этого также следует, что число $1 - 1/e$ присуще самой задаче, а не алгоритму.

[2] В этой книге вы больше не увидите число $1 - 1/e$, но оно таинственным образом будет повторяться в анализе алгоритмов.

второй итерации алгоритм снова мог выбрать A_1, A_2 или A_3, но был обманут подмножеством A_5, которое охватывало столько же новых элементов (18).

В общем случае каждая ошибка `GreedyCoverage` может быть отнесена к подмножеству, которое охватывает по меньшей мере столько же новых элементов, сколько каждое из K подмножеств в оптимальном решении. Но разве это не означает, что `GreedyCoverage` прогрессирует на каждой итерации? Эта идея формализуется в следующей лемме, которая ограничивает снизу число вновь охватываемых элементов на каждой итерации в зависимости от текущего дефицита охвата.

Лемма 20.8 (`GreedyCoverage` прогрессирует). *Для каждого $j \in \{1, 2, ..., k\}$ обозначим через C_j охват, достигаемый первыми j подмножествами, выбранными* `GreedyCoverage`. *Для каждого такого j выбранное j-е подмножество охватывает по меньшей мере $(1/k)(C^k - C_{j-1})$ новых элементов, где C обозначает максимально возможный охват k подмножествами:*

$$C_j - C_{j-1} \geq \frac{1}{k}\left(C^* - C_{j-1}\right). \tag{20.6}$$

Доказательство: обозначим через K_{j-1} индексы первых $j - 1$ подмножеств, выбранных `GreedyCoverage`, и $C_{j-1} = f_{\text{cov}}(K_{j-1})$ — их охват. Рассмотрим любое конкурирующее множество $\hat{K}$ из k индексов с соответствующим охватом $\hat{C} = f_{\text{cov}}(\hat{K})$.

Самое важное неравенство в доказательстве таково:

$$\sum_{i \in \hat{K}} \underbrace{\left[f_{\text{cov}}\left(K_{j-1} \cup \{i\}\right) - C_{j-1}\right]}_{\text{увеличение охвата с } A_i} \geq \underbrace{\hat{C} - C_{j-1}}_{\text{текущий разрыв в охвате}} \tag{20.7}$$

Почему оно является истинным? Элементы универсального множества W охватываются подмножествами, соответствующими $\hat{K}$, но не K_{j-1} (рис. 20.2). С одной стороны, размер W равен по меньшей мере $\hat{C} - C_{j-1}$ — правой части неравенства (20.7). С другой — он не превышает левую часть (20.7): каждый элемент W вносит свой вклад в сумму, хотя бы один раз влияя на подмножество с индексом в $\hat{K}$, которое его содержит. Следовательно, левая сторона неравенства (20.7) равна по меньшей мере правой стороне, причем размер W находится в промежутке между ними.

Далее, если бы k чисел, суммируемых в левой части (20.7), были эквивалентны, то каждое было бы равно $\frac{1}{k}\sum_{i\in\hat{K}}\left[f_{\text{cov}}\left(K_{j-1}\cup\{i\}\right)-C_{j-1}\right]$. В противном случае, самое крупное из них было бы еще больше:

$$\underbrace{\max_{i\in\hat{K}}\left[f_{\text{cov}}\left(K_{j-1}\cup\{i\}\right)-C_{j-1}\right]}_{\text{наибольшее значение}}\geq\underbrace{\frac{1}{k}\sum_{i\in\hat{K}}\left[f_{\text{cov}}\left(K_{j-1}\cup\{i\}\right)-C_{j-1}\right]}_{\text{среднее значение}}. \qquad (20.8)$$

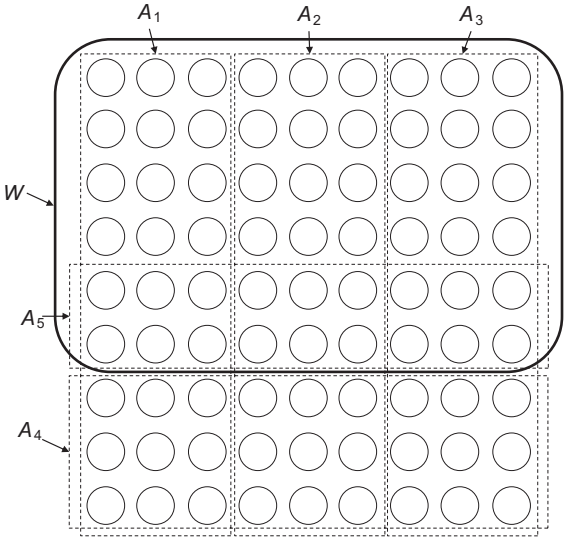


Рис. 20.2. Иллюстрация доказательства леммы 20.8 на примере из тестового задания 20.5, где $j = 2$, $K_{j-1} = \{4\}$, $C_{j-1} = 27$, $\hat{K} = \{1, 2, 3\}$ и $\hat{C} = 81$. Множество W равно 81 – 27 = 54 элементам, охватываемым некоторым подмножеством с индексом в K и никаким подмножеством с индексом в K_{j-1}

Теперь инстанцируем $\hat{K}$ в виде индексов K^* оптимального решения с охватом $f_{\text{cov}}(K^*) = C^*$. Выстроенные в цепь неравенства (20.7) и (20.8) показывают, что `GreedyCoverage` имеет по меньшей мере один хороший вариант (наилучший из индексов в оптимальном решении K^*):

$$\underbrace{\max_{i\in K^*}\left[f_{\text{cov}}\left(K_{j-1}\cup\{i\}\right)-C_{j-1}\right]}_{\text{наилучший из оптимальных индексов}}\geq\underbrace{\frac{1}{k}\left(C^*-C_{j-1}\right)}_{\text{гарантированный прогресс}}$$

`GreedyCoverage` благодаря жадному критерию выбирает индекс по меньшей мере не худший, тем самым увеличивая охват своего решения как минимум на $(1/k)(C^* - C_{j-1})$. *Ч. Т. Д.*

20.2.7. Доказательство теоремы 20.7

Теперь докажем теорему 20.7 путем многократного выполнения рекуррентного уравнения (20.6) из леммы 20.8, которое ограничивает снизу прогресс, достигаемый `GreedyCoverage` на каждой итерации. Продолжая те же обозначения, цель состоит в том, чтобы сравнить охват C_k, достигнутый решением алгоритма, с максимально возможным охватом C^*.

Предвосхищенный член $1 - 1/k$ входит в игру сразу после применения леммы 20.8 (сначала с $j = k$):

$$C_k \geq C_{k-1} + \frac{1}{k}\left(C^* - C_{k-1}\right) = \frac{C^*}{k} + \left(1 - \frac{1}{k}\right)C_{k-1}.$$

Применим его снова (теперь с $j = k - 1$):

$$C_{k-1} \geq \frac{C^*}{k} + \left(1 - \frac{1}{k}\right)C_{k-2}.$$

Объединим два неравенства:

$$C_k \geq \frac{C^*}{k}\left(1 + \left(1 - \frac{1}{k}\right)\right) + \left(1 - \frac{1}{k}\right)^2 C_{k-2}.$$

Применим лемму 20.8 третий раз с $j = k - 2$ и затем подставим C_{k-2}:

$$C_k \geq \frac{C^*}{k}\left(1 + \left(1 - \frac{1}{k}\right) + \left(1 - \frac{1}{k}\right)^2\right) + \left(1 - \frac{1}{k}\right)^3 C_{k-3}.$$

Указанный шаблон продолжается и после k применений леммы (и при $C_0 = 0$), получаем в итоге:

$$C_k \geq \frac{C^*}{k}\underbrace{\left(1 + \left(1 - \frac{1}{k}\right) + \left(1 - \frac{1}{k}\right)^2 + \ldots + \left(1 - \frac{1}{k}\right)^{k-1}\right)}_{\text{геометрический ряд}}.$$

Внутри скобок находится наш старый друг — геометрический ряд. В общем случае для $r \neq 1$ существует полезная формула в замкнутой форме для геометрического ряда:[1]

$$1 + r + r^2 + \ldots + r^{\ell} = \frac{1 - r^{\ell+1}}{1 - r}. \tag{20.9}$$

При вызове этой формулы с $r = 1 - 1/k$ и $\ell = k - 1$ нижняя граница по отношению к C_k преобразовывается в

$$C_k \geq \frac{C^*}{k}\left(\frac{1 - \left(1 - \frac{1}{k}\right)^k}{1 - \left(1 - \frac{1}{k}\right)}\right) = C^*\left(1 - \left(1 - \frac{1}{k}\right)^k\right).$$

исполнив обещание, данное теоремой 20.7. *Ч. Т. Д.*

20.2.8. Решения к тестовым заданиям 20.4–20.5

Решение к тестовому заданию 20.4

Правильный ответ: (в). Есть два способа охватить 15 из 16 элементов: выбрать A_2, A_4 и A_6 или либо A_3, либо A_5. Большое подмножество A_1 не участвует ни в одном оптимальном решении, поскольку оно в значительной степени является избыточным по отношению к другим подмножествам.

Решение к тестовому заданию 20.5

Правильный ответ: (б). Оптимальное решение выбирает подмножества A_1, A_2 и A_3 и охватывает весь 81 элемент. Одно из возможных выполнений `GreedyCoverage` выбирает A_4 на первой итерации, разрывая четырехстороннюю связь с A_1, A_2, A_3; A_5 на второй итерации, снова разрывая четырехстороннюю связь с A_1, A_2, A_3; и, наконец, A_1. Это решение имеет охват 27 + 18 + 12 = 57.

---

[1] Для того чтобы проверить это тождество, обе стороны следует умножить на $1 - r$: $(1 - r)(1 + r + r^2 + \cdots + r^{\ell}) = 1 - r + r - r^2 + r^2 - r^3 + r^3 - \cdots - r^{\ell+1} = 1 - r^{\ell+1}$.

*20.3. Максимизация влияния

`GreedyCoverage` (раздел 20.2) изначально задумывался для классических применений, таких как выбор местоположений для новых заводов. В XIX веке обобщения этого алгоритма нашли новое применение в computer science. Этот раздел описывает репрезентативный пример в анализе социальных сетей.[1]

20.3.1. Каскады в социальных сетях

Для наших целей *социальная сеть* — это ориентированный граф $G = (V, E)$, в котором вершины соответствуют пользователям, а ориентированное ребро (v, w) означает, что v «влияет на» w. Например, w следит за новостями v.

ПРОСТАЯ КАСКАДНАЯ МОДЕЛЬ

Изначально каждая затравочная вершина является «активной», а все остальные вершины «неактивными». Все ребра изначально находятся «не в подброшенном» состоянии.

Пока есть активная вершина v и не подброшенное исходящее ребро (v, w), надо:

- подбросить несмещенную монету: орел выпадает с вероятностью p;
- нсли выпадает орел, обновить статус ребра (v, w) значением «активное». Если вершина w не активна, обновить статус ребра значением «активное»;
- если выпадает решка, обновить статус ребра (v, w) значением «неактивное».

Каскадная модель показывает, как информация (например, новостная статья или мем) перемещается по социальной сети. Вот простой пример сети,

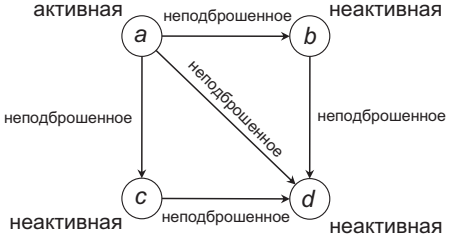
[1] Разделы, помеченные звездочкой, подобные этому, являются более трудными, и при первом прочтении их можно пропустить.

параметризованной ориентированным графом $G = (V, E)$, активационной вероятностью $p \in [0, 1]$ и подмножеством $S \subseteq V$ *затравочных* (seed) вершин.[1]

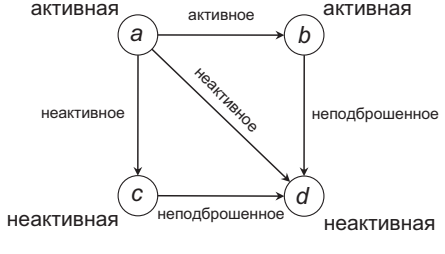
После того как вершина активирована (скажем, вследствие чтения статьи или просмотра фильма), она больше не станет неактивной. Вершина может иметь несколько активационных возможностей — одну для каждого ее активизированного влиятельного объекта. Например, первые две рекомендации от друзей относительно нового фильма не регистрируются, но третья побуждает вас его посмотреть.

20.3.2. Пример

В графе

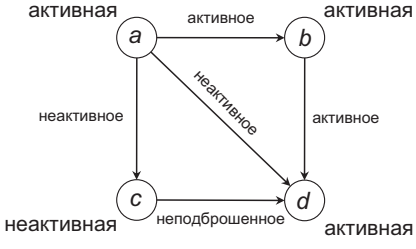


вершина a является затравочной и изначально активной. Остальные вершины изначально неактивны. Каждое из исходящих ребер (a, b), (a, c) и (a, d) имеет вероятность p активирования другой конечной точки ребра. Предположим, что монета, связанная с ребром (a, b), выпадает орлом, а две другие — решкой. Новая картина такова:

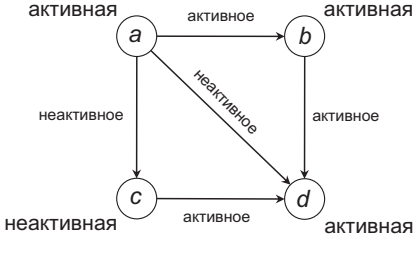


[1] Для того чтобы освежить информацию о базовой дискретной вероятности, см. Приложение Б *Первой части* либо ресурсы на веб-сайте www.algorithmsilluminated.org.

В этот момент нет никакой надежды активировать вершину *c*. Остается вероятность *p*, что вершина *d* активируется через неподброшенное ребро (*b*, *d*). Если это событие произойдет, то финальным состоянием будет:



Для замыкания и удобства можно факультативно добавить шаг постобработки, который подбрасывает монеты любых оставшихся неподброшенных ребер и соответствующим образом обновляет их статусы (оставляя статусы всех вершин неизменными). В нашем рабочем примере окончательный результат может быть таким:



В общем случае, с шагом постобработки или без него, вершины, которые в итоге активируются в конце процесса, достигаются из затравочной вершины по ориентированному пути, состоящему из активированных ребер.

20.3.3. Задача о максимизации влияния

Цель задачи о *максимизации влияния* состоит в выборке ограниченного числа затравочных вершин в социальной сети для максимизации разброса информации (при числе вершин, которые в конечном итоге активируются в соответствии

с каскадной моделью).[1] Это число является случайной величиной, зависящей от исходов подбрасываний монеты в каскадной модели, и мы сосредоточимся на его математическом ожидании.[2] Формально обозначим через $X(S)$ (случайное) множество вершин, которые в конечном итоге активируются, когда вершины S выбираются в качестве затравочных, и определим *влияние* S как

$$f_{\text{inf}}(S) = \mathbf{E}[|X(S)|], \tag{20.10}$$

где математическое ожидание строится над случайными подбрасываниями монеты в каскадной модели. Влияние множества зависит как от графа, так и от активационной вероятности, причем большее число ребер или более высокая вероятность приводят к большему влиянию.

ЗАДАЧА: МАКСИМИЗАЦИЯ ВЛИЯНИЯ

Вход: ориентированный граф $G = (V, E)$, вероятность p и положительное целое число k.

Выход: выборка $S \subseteq V$ из k вершин с максимально возможным влиянием $f_{\text{inf}}(S)$ в каскадной модели с активационной вероятностью p.

Например, если вы раздаете k рекламных копий продукта и хотите выбрать получателей, чтобы максимизировать его принятие, то вы сталкиваетесь с задачей о максимизации влияния.

Задача 20.9 просит вас показать, что задача о максимальном охвате может рассматриваться как частный случай задачи о максимизации влияния. Поскольку частный случай является NP-трудным (задача 22.8), то и более общая

---

[1] Подробнее о задаче о максимизации влияния и ее многочисленных вариациях читайте в статье Дэвида Кемпе, Джона Клейнберга и Эвы Тардос «Максимизирование распространения влияния через социальную сеть» («*Maximizing the Spread of Influence Through a Social Network», David Kempe, Jon Kleinberg, Éva Tardos, Theory of Computing*, 2015).

[2] Математическое ожидание $\mathbf{E}[Y]$ случайной величины Y — это ее среднее значение, взвешенное соответствующими вероятностями. Например, если Y может принимать значения $\{0, 1, 2, \ldots, n\}$, то $\mathbf{E}[Y] = \sum_{i=0}^{n} i \times \mathbf{Pr}[Y = i]$.

задача тоже. Может ли существовать быстрый и приближенно правильный эвристический алгоритм для задачи о максимизации влияния?

20.3.4. Жадный алгоритм

Задача о максимизации влияния напоминает задачу о максимальном охвате, где вершины играют роль подмножеств, а влияние (20.10) играет роль охвата (20.5). `GreedyCoverage` для последней задачи подходит для первой, если подставить новое определение (20.10) целевой функции.

GREEDYINFLUENCE

Вход: ориентированный граф $G = (V, E)$, вероятность $p \in [0, 1]$ и положительное целое число k.

Выход: множество $S \subseteq V$ из k вершин.

```
1 S := ∅                                          // выбранные вершины
2 for j = 1 to k do          // выбирать вершины одну за другой
    // жадно увеличивать влияние
    // (произвольно разрывать связи)
3     v* := argmax_{v∈V}[f_inf(S ∪ {v}) − f_inf(S)]
4     S := S ∪ {v*}
5 return S
```

ТЕСТОВОЕ ЗАДАНИЕ 20.6

Каково время выполнения простой реализации `GreedyInfluence` на графах с n вершинами и m ребрами? (Выбрать самый точный ответ.)

а) $O(knm)$

б) $O(knm^2)$

в) $O(knm^{2m})$

г) неясно

(Ответ и анализ решения см. в разделе 20.3.8.)

20.3.5. Приближенная правильность

Поскольку задача о максимальном охвате является частным случаем задачи о максимизации влияния (задача 20.9), известная нам гарантия приближенной правильности жадного эвристического алгоритма будет столь же сильна для более общей задачи.

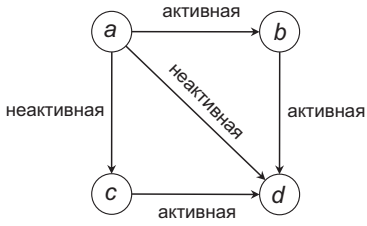
Теорема 20.9 (`GreedyInfluence`: приближенная правильность). *Влияние решения на выходе из алгоритма жадного влияния* `GreedyInfluence` *всегда составляет по меньшей мере* $1 - (1 - 1/k)^k$*-ю долю максимально возможного влияния, где* k *— это число выбранных вершин.*

Ключевым моментом в доказательстве теоремы 20.9 является признание функции влияния (20.10) как средневзвешенного значения функций охвата (20.5). Каждая из этих функций охвата соответствует применению, предметом которого была посещаемость концерта (раздел 20.2.2), с подграфом социальной сети (содержащим активированные ребра). Теперь мы можем проверить, что доказательство теоремы 20.7 в разделах 20.2.6 и 20.2.7 для функций охвата может быть распространено на средневзвешенные значения функций охвата.

20.3.6. Влияние есть взвешенная сумма функций охвата

Выражаясь формальнее, зададим направленный граф $G = (V, E)$, активационную вероятность $p \in [0, 1]$ и положительное целое число k. Для удобства включим в каскадную модель шаг постобработки (раздел 20.3.2), чтобы каждое ребро в итоге стало либо активным, либо неактивным. Вершины $X(S)$, активируемые затравочными вершинами в S, будут достижимы из вершины в S по ориентированному пути активированных ребер.

Если мы заранее угадали бы ребра $H \subseteq E$, которые будут активированы, бросая монеты всех ребер загодя, задача о максимизации влияния сводилась бы к задаче о максимальном охвате. Универсальным множеством стали бы вершины V, и на каждую вершину пришлось бы одно подмножество, причем подмножество $A_{v,H}$ содержало бы вершины, доступные из v по ориентированному пути в подграфе (V, H) активированных ребер. Например, если граф и реберные статусы таковы:



тогда $A_{a,H} = \{a, b, d\}$, $A_{b,H} = \{b, d\}$, $A_{c,H} = \{c, d\}$ и $A_{d,H} = \{d\}$. Влияние множества S затравочных вершин (учитывая активированные ребра H) стало бы охватом соответствующих подмножеств:

$$f_H(S) := |\cup_{v \in S} A_{v,H}|. \tag{20.11}$$

Конечно, у нас нет предварительных знаний о подмножестве активированных ребер, и каждое подмножество $H \subseteq E$ случается с некоторой положительной вероятностью p_H.[1] Но поскольку влияние определяется как математическое ожидание, можно выразить его как средневзвешенное значение функций охвата с весами, равными вероятностям.[2]

Лемма 20.10 (влияние равно среднему значению функций охвата). *Для каждого подмножества $H \subseteq E$ ребер обозначим функцию охвата f_H, определенную в (20.11), и вероятность p_H того, что подмножество ребер, активированных в каскадной модели, точно равно H. Для каждого подмножества $S \subseteq V$ вершин*

$$f_{\inf}(S) = \mathbf{E}_H\left[f_H(S)\right] = \sum_{H \subseteq E} p_H \times f_H(S) \tag{20.12}$$

20.3.7. Доказательство теоремы 20.9

Мы объявим победу, если докажем аналог (20.8), показав, что `GreedyInfluence` прогрессирует на каждой итерации. Теорема 20.9 следует из той же самой

---

[1] Не то чтобы она нам понадобилась, но формула такова: $p_H = p^{|H|}(1-p)^{|E|-|H|}$.

[2] Для одержимых строгостью: мы используем правило полного математического ожидания, чтобы записать математическое ожидание в (20.10) как средневзвешенное по вероятности условных ожиданий, строящихся на активированных ребрах H.

алгебры, которую мы использовали для доказательства теоремы 20.7 в разделе 20.2.7.

Лемма 20.11 (`GreedyInfluence` прогрессирует). *Для каждого $j \in \{1, 2, \ldots, k\}$ обозначим влияние I_j, достигаемое первыми j вершинами, выбранными* `GreedyInfluence`. *Для каждого такого j выбранная j-я вершина увеличивает влияние по меньшей мере на $1/k(I^* - I_{j-1})$, где I^* обозначает максимально возможное влияние k вершин:*

$$I_j - I_{j-1} \geq \frac{1}{k}(I^* - I_{j-1}).$$

Доказательство: обозначим через S_{j-1} первые $j - 1$ вершины, выбранные алгоритмом `GreedyInfluence`, через $I_{j-1} = f_{inf}(S_{j-1})$ — их влияние, а через S^* — множество k вершин с максимально возможным влиянием I^*.

Рассмотрим произвольное подмножество $H \subseteq E$ ребер и функцию охвата f_H, определенную в (20.11). Опираясь на свои знания о функциях охвата, переведем ключевое неравенство (20.7) в анализе алгоритма `GreedyCoverage` в неравенство

$$\sum_{v \in S^*} \underbrace{\left[f_H\left(S_{j-1} \cup \{v\}\right) - f_H\left(S_{j-1}\right) \right]}_{\text{увеличение охвата с } f_H \text{ (при } f_H)} \geq \underbrace{f_H\left(S^*\right) - f_H\left(S_{j-1}\right)}_{\text{разрыв охвата (при } f_H)}. \qquad (20.13)$$

Здесь S_{j-1} и S^* играют роли K_{j-1} и $\hat{K}$, а $f_H(S_{j-1})$ и $f_H(S^*)$ соответствуют C_{j-1} и $\hat{C}$.

Согласно лемме 20.10, влияние (f_{inf}) — это средневзвешенное значение функций охвата (f_H). Возьмем неравенство формы (20.13) для каждого подмножества $H \subseteq E$ ребер и обозначим его левую и правую стороны как L_H и R_H, чтобы рассмотреть аналогичное средневзвешенное значение этих 2^m неравенств (где m обозначает $|E|$).

Поскольку умножение обеих сторон неравенства на одно и то же неотрицательное число (например, вероятность p_H) сохраняет его,

$$p_H \times L_H \geq p_H \times R_H$$

для каждого $H \subseteq E$. Поскольку все 2^m неравенства идут в одном направлении, они складываются в комбинированное неравенство:

$$\sum_{H \subseteq E} p_H \times L_H \geq \sum_{H \subseteq E} p_H \times R_H. \tag{20.14}$$

Распаковав правую часть (20.14) и использовав расширенную формулу для f_{inf} в (20.12), мы получим

$$\sum_{H \subseteq E} p_H \left(f_H\left(S^*\right) - f_H\left(S_{j-1}\right)\right) = \sum_{H \subseteq E} p_H \times f_H\left(S_{j-1}\right)$$
$$= f_{\text{inf}}\left(S^*\right) - f_{\text{inf}}\left(S_{j-1}\right).$$

Левая сторона неравенства (20.14) после тех же маневров становится

$$\underbrace{\sum_{v \in S^*} \left[f_{\text{inf}}\left(S_{j-1} \cup \{v\}\right) - f_{\text{inf}}\left(S_{j-1}\right)\right]}_{\text{левая сторона неравенства (20.14)}}.$$

Таким образом, неравенство (20.14) переводится в аналог ключевого неравенства (20.7) в доказательстве леммы 20.8:

$$\sum_{v \in S^*} \left[f_{\text{inf}}\left(S_{j-1} \cup \{v\}\right) - \underbrace{f_{\text{inf}}\left(S_{j-1}\right)}_{=I_{j-1}} \right] \geq \underbrace{f_{\text{inf}}\left(S^*\right)}_{=I^*} - \underbrace{f_{\text{inf}}\left(S_{j-1}\right)}_{=I_{j-1}}. \tag{20.15}$$

Самый большой из k членов в сумме слева представляет собой по меньшей мере среднее значение (как в (20.8)), поэтому `GreedyInfluence` всегда имеет хотя бы один хороший вариант (лучшую из вершин в оптимальном решении S^*):

$$\max_{v \in S^*} \left[f_{\text{inf}}\left(S_{j-1} \cup \{v\}\right) - I_{j-1} \right] \geq \frac{1}{k}\left(I^* - I_{j-1}\right).$$

`GreedyInfluence` из-за жадного критерия выбирает по крайней мере не худшую вершину, увеличивая влияние своего решения на $(1/k)(I^* - I_{j-1})$. *Ч. Т. Д.*

20.3.8. Решение к тестовому заданию 20.6

Правильные ответы: (в), (г). Каждая из k итераций главного цикла предусматривает вычисление argmax над n вершинами. Поэтому время выполнения простой реализации в $O(kn)$ раз превышает вычисление влияния $f_{\text{inf}}(S)$

подмножества S. Сколько операций оно требует? Ответ на этот вопрос не очевиден из-за математического ожидания в (20.10). (В этом смысле верен ответ (г).) Наивное вычисление $|X(S)|$ посредством поиска сначала в ширину или сначала в глубину за $O(m)$ время для каждого из 2^m возможных исходов подбрасывания монеты и усреднение результатов приводят к границе времени выполнения в (в).

Полезен ли `GreedyInfluence` на практике? Конечно. Возможно, вычислить влияние $f_{\text{inf}}(S)$ подмножества S с произвольной прецизионностью трудно, но его можно точно оценить через случайный отбор — подбросить все монеты в каскадной модели и посмотреть, сколько вершин активируется с помощью затравочного множества S. После многократного повтора этого эксперимента среднее число активированных вершин почти всегда будет хорошей оценкой влияния $f_{\text{inf}}(S)$.

20.4. Эвристический алгоритм двукратной замены для задачи коммивояжера

NP-трудность всегда тормозит решение, но для таких задач, как в разделах 20.1–20.3 (о минимизации производственной продолжительности, максимальном охвате и максимизации влияния), существуют быстрые алгоритмы с хорошими гарантиями приближенной правильности. Увы, для многих других NP-трудных задач, включая задачу коммивояжера, такой алгоритм опроверг бы предположение, что $\text{P} \neq \text{NP}$ (см. задачу 22.12). Для них в лучшем случае подойдет эвристический алгоритм, который, несмотря на отсутствие страхового полиса, хорошо работает на образцах задач вашего приложения. *Локальный поиск* и его многочисленные варианты являются одной из наиболее мощных и гибких парадигм для разработки алгоритмов такого типа.

20.4.1. Решение задачи коммивояжера

Пока рано говорить о локальном поиске в целом. Сейчас мы разработаем с нуля эвристический алгоритм для задачи коммивояжера, чтобы развить несколько новых идей. Затем в разделе 20.5 мы уменьшим масштаб и определим элементы решения, иллюстрирующие общие принципы локального

поиска. Вооружившись заготовкой для разработки алгоритмов локального поиска и примером создания экземпляра, вы сможете обогатить свою работу.

В задаче коммивояжера (раздел 19.1.2) на вход подается полный граф $G = (V, E)$ с вещественнозначными реберными стоимостями, чтобы вычислить тур — цикл, посещающий каждую вершину ровно один раз, — с минимально возможной суммой реберных стоимостей. Задача коммивояжера является NP-трудной (раздел 22.7). Если скорость имеет для нее критическую важность, для ее решения необходим эвристический алгоритм (исходя из истинности предположения о том, что $P \neq NP$).

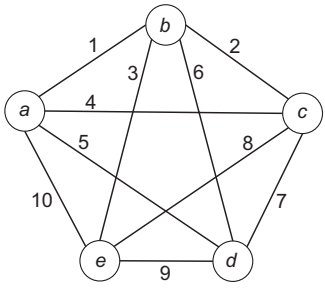
Чтобы понять задачу коммивояжера, начнем с первого жадного алгоритма, о котором вы, возможно, уже подумали, — алгоритма поиска минимального остовного дерева.

ТЕСТОВОЕ ЗАДАНИЕ 20.7

Эвристический алгоритм *ближайшего соседа* — это жадный алгоритм для задачи коммивояжера, который при наличии полного графа с вещественнозначными реберными стоимостями работает следующим образом:

1. Начать тур с произвольной вершины a.
2. Повторять до тех пор, пока не будут посещены все вершины:
   а) если текущая вершина равна v, то перейти к ближайшей непосещенной вершине w, минимизирующей c_{vw}.
3. Вернуться к стартовой вершине.

Какова стоимость тура, построенного алгоритмом ближайшего соседа, и какова минимальная стоимость тура в следующем примере?



а) 23 и 29

б) 24 и 29

в) 25 и 29

г) 24 и 30

(Ответ и анализ решения см. в разделе 20.4.6.)

Тестовое задание 20.7 показывает, что алгоритм ближайшего соседа не всегда строит самый дешевый тур. И это неудивительно, ведь задача коммивояжера является NP-трудной, а алгоритм выполняется за полиномиальное время. Что еще тревожнее, жадно построенный тур остается без изменений, даже если поменять стоимость финального перехода (a, e) на огромное число. В отличие от жадных эвристических алгоритмов из разделов 20.1–20.3, алгоритм ближайшего соседа может производить решения бесконечно хуже оптимального. Более сложные жадные алгоритмы способны преодолевать этот конкретный плохой пример, но все они в конечном счете страдают одинаково в сложных экземплярах задачи коммивояжера.

20.4.2. Улучшение тура двукратной заменой

Кто сказал, что можно сдаваться после построения первого тура? Если есть способ улучшить тур жадным способом, почему бы это не сделать? Каковы минимальные изменения, способные повысить качество тура?

ТЕСТОВОЕ ЗАДАНИЕ 20.8

Каково максимальное число ребер, которые могут совместно использоваться двумя отдельными турами в экземпляре задачи коммивояжера с n вершинами?

а) $\log_2 n$

б) $n/2$

в) $n - 2$

г) $n - 1$

(Ответ и анализ решения см. в разделе 20.4.6.)

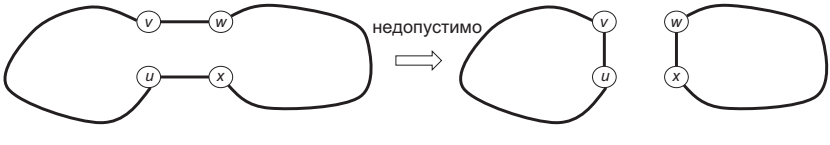
Тестовое задание 20.8 предлагает разведать ландшафт туров, обменивая одну пару ребер на другую:

Этот тип обмена называется *двукратной заменой* (2-change).

ДВУКРАТНАЯ ЗАМЕНА

1. Получив тур T, удалить два ребра (v, w), (u, x) из T, не имеющие общей конечной точки.
2. Добавить ребра (v, x) и (u, w) либо ребра (u, v) и (w, x), в зависимости от того, какая пара приведет к новому туру T'.

На первом шаге выбираем два ребра с четырьмя разными конечными точками.[1] Существует три разных варианта соединения четырех вершин в пары, и только один из них создает новый тур (как в предыдущем рисунке). Помимо этого и их первоначального соединения в пару, третье соединение в пару создает два непересекающихся цикла вместо допустимого тура:



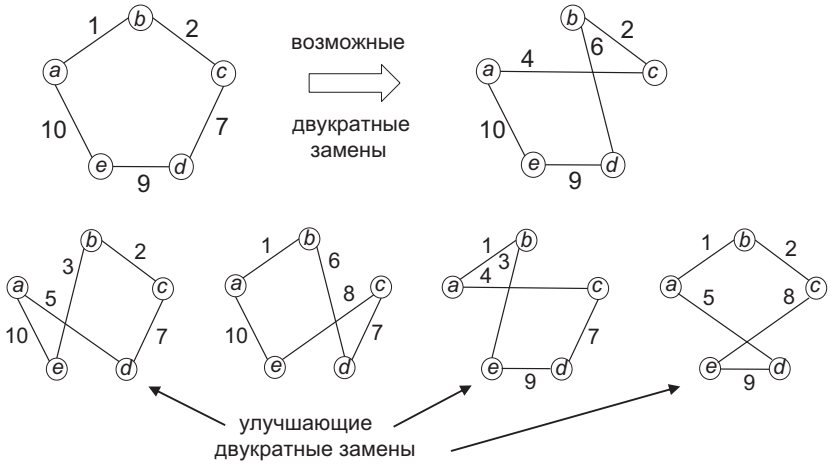
Двукратная замена может создать тур, который будет лучше или хуже оригинального. Если только что обменянные ребра равны (u, w) и (v, x),

$$\text{уменьшение в стоимости тура} = \underbrace{\left(c_{vu} + c_{wx}\right)}_{\text{удаленные ребра}} - \underbrace{\left(c_{uw} + c_{vx}\right)}_{\text{добавленные ребра}} \qquad (20.16)$$

[1] Удалять два ребра с совместной конечной точкой бессмысленно. Единственный способ вернуть допустимый тур — поместить ребра назад на свое место.

Если уменьшение в (20.16) положительное — выгода $c_{vw} + c_{ux}$ от удаления старых ребер перевешивает стоимость $c_{uw} + c_{vx}$ добавления новых, — то двукратная замена производит более дешевый тур и называется *улучшающей*.

Например, вместе с жадно построенным туром из тестового задания 20.7 есть пять возможных двукратных замен, три из которых улучшающие:[1]



20.4.3. Алгоритм двукратной замены 2-OPT

Алгоритм двукратной замены `2-OPT` для задачи коммивояжера строит начальный тур (например, с помощью алгоритма ближайшего соседа) и выполняет все возможные улучшающие двукратные замены, пока они не кончатся. В следующем псевдокоде подпрограмма `2Change` принимает на входе тур и два его ребра (с разными конечными точками) и возвращает на выходе тур, созданный двукратной заменой (с. 104).

2OPT

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

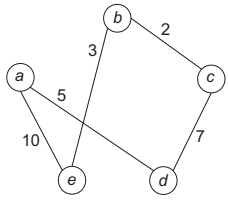
---

[1] В общем случае с $n \geq 4$ вершинами всегда существует $n(n-3)/2$ возможных двукратных замен: выбирая одно из n ребер тура, за которым следует одно из $n-3$ ребер тура с другими конечными точками, мы подсчитываем каждую двукратную замену двумя способами.

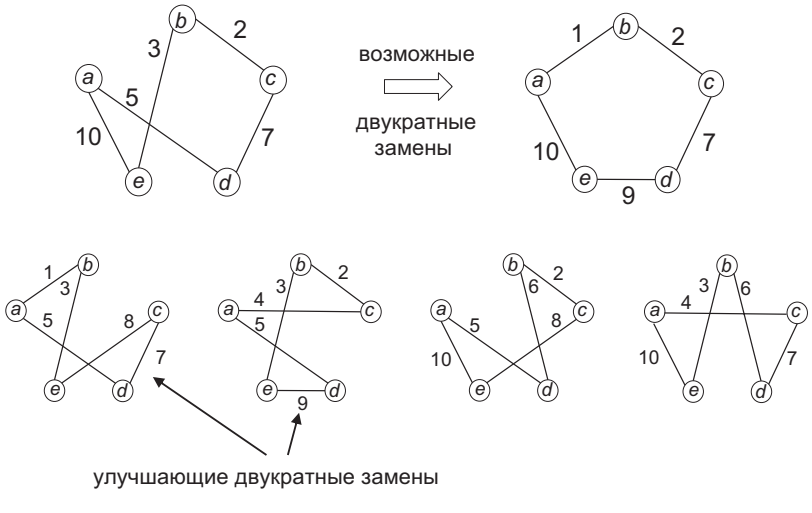
Выход: тур коммивояжера.

```
1 T := первый тур // возможно, жадно построенный
2 while улучшающая двукратная замена (v, w), (u, x) ∈ T
  существует do
3     T := 2Change(T, (v, w), (u, x))
4 return T
```

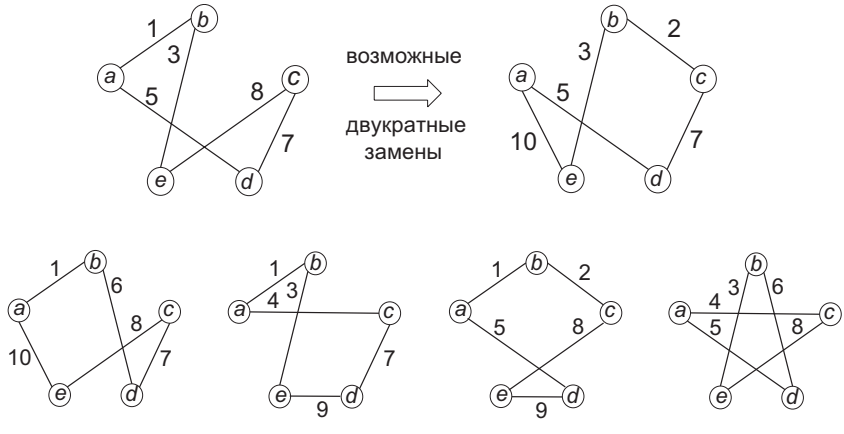
Например, начиная с тура, построенного алгоритмом ближайшего соседа в тестовом задании 20.7, первая итерация алгоритма `2OPT` может заменить ребра (a, b) и (d, e) на ребра (a, d) и (b, e):



тем самым снизив стоимость тура с 29 до 27. Отсюда следуют еще пять двукратных замен:



Если на второй итерации алгоритма выполняется первая из двух улучшающих двукратных замен (ребер (*a*, *e*) и (*b*, *c*) на ребра (*a*, *b*) и (*c*, *e*)), то стоимость тура уменьшается еще больше, с 27 до 24. На данный момент улучшающих двукратных замен больше нет (одна оставляет стоимость тура неизменной, а четыре ее увеличивают), и алгоритм останавливается:



не улучшающие двукратные замены

20.4.4. Время работы

Останавливается ли алгоритм `2OPT` вообще или же он зацикливается? Число туров коммивояжера огромно (тестовое задание 19.1), но все же оно конечно. Каждая итерация алгоритма `2OPT` производит тур со стоимостью строго меньшей, чем предыдущая, поэтому один и тот же тур точно не появится в двух разных итерациях. Даже если в сценарии Судного дня алгоритм выполняет поиск по всем возможным турам, он в итоге остановится.

Время выполнения `2OPT` регулируется числом итераций главного цикла `while`, умноженным на число операций, выполняемых за одну итерацию. При n вершинах необходимо проверить $O(n^2)$ разных двукратных замен на каждой итерации, что приведет к ограничению времени итерации, равному $O(n^2)$ (задача 20.13). А как быть с числом итераций?

Плохая новость: иногда перед остановкой алгоритм `2OPT` может выполнять экспоненциальное (в n) число итераций. Хорошие же новости имеют двоякий

характер. Во-первых, на более реалистичных входных данных алгоритм 2OPT почти всегда останавливается за разумное число итераций (обычно субквадратичное в n). Во-вторых, поскольку указанный алгоритм поддерживает допустимый тур на протяжении всего его выполнения, он может быть прерван в любое время.[1] Длительность выполнения алгоритма можно определить заранее (одна минута, один час, один день и т. д.) и, когда время истечет, использовать последнее (и наилучшее) найденное решение.

20.4.5. Качество решения

Алгоритм двукратной замены 2OPT может улучшить свой первый тур, не найдя оптимального решения. Пример в тестовом задании 20.7 из раздела 20.4.3 уже показал, как алгоритм вернул тур со стоимостью 24 вместо оптимального тура (который стоил 23). Могут ли быть примеры еще хуже? И насколько хуже?

Плохая новость: более сложные и надуманные примеры показывают, что тур, возвращаемый 2OPT, может стоить бесконечно больше оптимального тура. Другими словами, алгоритм не дает гарантий приближенной правильности сродни тем, что приведены в разделах 20.1–20.3. Хорошая новость: для задачи коммивояжера на практике 2OPT обычно находит туры с суммарной стоимостью, ненамного выше минимально возможной. Для решения такой задачи на больших входных данных (где n исчисляется в тысячах или более) алгоритм 2OPT с дополнениями, описанными в разделе 20.5, будет отличной отправной точкой.

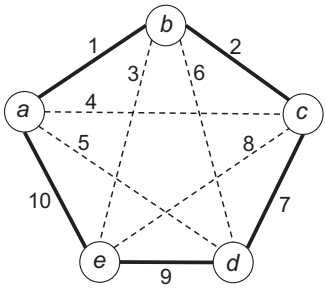
20.4.6. Решения к тестовым заданиям 20.7–20.8

Решение к тестовому заданию 20.7

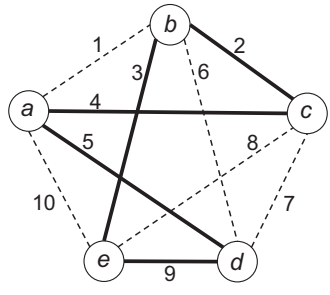
Правильный ответ: (a). Алгоритм ближайшего соседа начинается в a и жадно перемещается в b, и затем в c. В этот момент тур должен перейти либо в d, либо в e (так как ни одна вершина не может быть посещена дважды), причем d предпочтительнее (стоимость 7 вместо 8). Находясь в d, остальные переходы являются вынужденными: нет другого варианта, кроме как отправиться в e (со

[1] Такие алгоритмы называются *алгоритмами с остановкой в любое время* (anytime algorithm) или алгоритмами с отсечением по времени.

стоимостью 9) и затем вернуться в *a* (со стоимостью 10). Суммарная стоимость этого тура составляет 29 (см. ниже левый рисунок). Между тем минимальная суммарная стоимость тура составляет 23 (см. ниже правый рисунок).



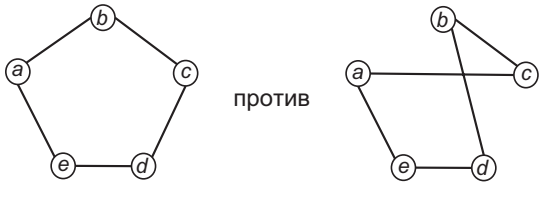
а) Жадный тур



б) Оптимальный тур

Решение к тестовому заданию 20.8

Правильный ответ: (в). Поскольку любые $n - 1$ ребер тура уникально определяют его финальное ребро, разные туры не могут совместно использовать $n - 1$ ребер. Однако разные туры могут совместно использовать $n - 2$ ребер:



20.5. Принципы локального поиска

Алгоритм локального поиска исследует пространство допустимых решений с помощью локальных ходов, которые один за другим улучшают целевую функцию. Каноническим примером такого алгоритма является `2OPT` для задачи коммивояжера, в котором локальные ходы соответствуют двукратным заменам. В этом разделе выделяются и изолируются существенные аспекты парадигмы проектирования алгоритмов локального поиска, а также ключевые модельные и алгоритмические решения, необходимые для ее применения.

20.5.1. Метаграф допустимых решений

Для графа $G = (V, E)$ экземпляра задачи коммивояжера с вещественнозначными реберными стоимостями `2OPT` визуализируется как жадная прогулка по метаграфу $H = (X, F)$ допустимых решений (рис. 20.3, на примере тестового задания 20.7). Метаграф H имеет одну вершину $x \in X$ для каждого тура в G, помечаемого суммарной стоимостью тура. Он также имеет одно ребро $(x, y) \in F$ для каждой пары x у туров, которые отличаются ровно двумя ребрами графа G. Другими словами, ребра метаграфа соответствуют возможным двукратным заменам.

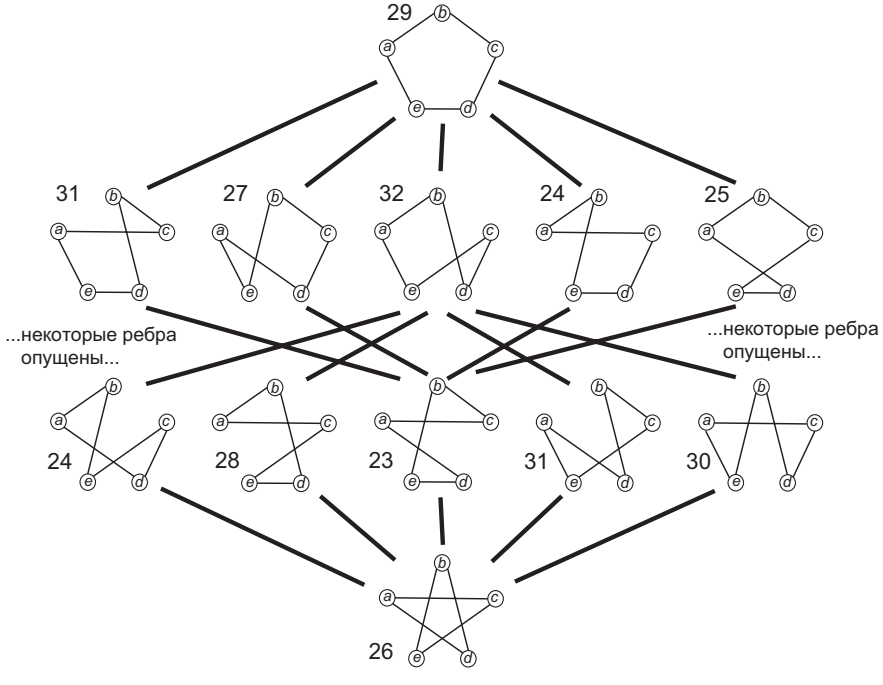


Рис. 20.3. Метаграф допустимых решений для экземпляра задачи коммивояжера из тестового задания 20.7. Вершины метаграфа соответствуют турам, а два тура соединяются ребром метаграфа, если и только если они отличаются ровно двумя ребрами. Верхний тур примыкает в метаграфе к пяти турам во втором ряду, а нижний тур — к турам в третьем ряду. Каждый тур во втором ряду примыкает к каждому туру в третьем ряду, за исключением тура в своем собственном столбце. (Во избежание беспорядка некоторые из этих ребер метаграфа на рисунке опущены.) Каждый тур помечен своей суммарной стоимостью

ТЕСТОВОЕ ЗАДАНИЕ 20.9

Сколько вершин и ребер имеет метаграф, соответствующий экземпляру задачи коммивояжера с $n \geq 4$ вершинами?[1]

а) $1/2(n-1)!$ и $n!(n-3)/8$

б) $1/2(n-1)!$ и $n!(n-3)/4$

в) $(n-1)!$ и $n!(n-3)/4$

г) $(n-1)!$ и $n!(n-3)/2$

(Ответ и анализ решения см. в разделе 20.5.8.)

Представим, что `2OPT` начинает поиск в некоторой вершине метаграфа (например, на выходе алгоритма ближайшего соседа) и многократно пересекает ребра метаграфа, чтобы поочередно посетить последовательность туров с меньшими стоимостями. Алгоритм останавливается, когда достигает вершины метаграфа со стоимостью не больше, чем у любого из его соседей в метаграфе. Пример траектории алгоритма `2OPT` в разделе 20.4.3 начинается с верхнего тура (рис. 20.3), перед тем как перейти ко второму туру во втором ряду, а затем останавливается в первом туре в третьем ряду.

20.5.2. Парадигма проектирования алгоритма локального поиска

Большинство алгоритмов локального поиска тоже можно визуализировать как жадную прогулку в метаграфе допустимых решений.[2] Такие алгоритмы отличаются только выбором метаграфа и деталями стратегии его исследования.[3]

---

[1] Не беспокойтесь о том, что метаграф ТАКОЙ ОГРОМНЫЙ. Он существует только в нашем сознании и никогда не будет записан явным образом.

[2] Можно даже добавить в визуализацию третью размерность, причем «высота» вершины метаграфа будет значением его целевой функции. Это объясняет, почему локальный поиск иногда называют *восхождением на вершину холма*.

[3] Один из вариантов — *градиентный спуск*. Это древний алгоритм локального поиска для непрерывной (не дискретной) оптимизации, который занимает центральное место в современном машинном обучении. Простой пример градиентного спуска —

ПАРАДИГМА ЛОКАЛЬНОГО ПОИСКА

1. Определить допустимые решения (вершины метаграфа).
2. Определить целевую функцию (числовые метки вершин метаграфа) и цель: ее максимизация или минимизация.
3. Определить разрешенные локальные ходы (ребра метаграфа).
4. Решить, как выбирать первое допустимое решение (стартовую вершину метаграфа).
5. Решить, как выбирать один улучшающий локальный ход из нескольких (возможные следующие шаги в метаграфе).
6. Выполнить локальный поиск: начиная с первого допустимого решения итеративно улучшать значение целевой функции посредством локальных ходов до достижения *локального оптимума* — допустимого решения, из которого невозможно улучшить локальный ход.

Псевдокод генерического алгоритма локального поиска очень похож на псевдокод алгоритма `2OPT`. (Подпрограмма `MakeMove` принимает на входе допустимое решение и описание локального хода и возвращает соответствующее соседнее решение.)

GENERICLOCALSEARCH

```
S := первое решение        // шаг 4
while улучшающий локальный ход L существует do
   S := MakeMove(S, L)     // шаг 5
return S // вернуть найденный локальный оптимум
```

эвристический алгоритм для минимизации дифференцируемой целевой функции по всем точкам евклидова пространства с улучшением локальных ходов, которые соответствуют малым шагам в направлении наиболее крутого спуска (то есть отрицательного градиента) из текущей точки.

Первые три шага в парадигме локального поиска представляют собой модельные решения, а следующие два — алгоритмические решения. Рассмотрим шаги подробнее.

20.5.3. Три модельных решения

Первые два шага парадигмы локального поиска определяют задачу.

Шаг 1. Определить допустимые решения. В задаче коммивояжера допустимыми решениями n-вершинного экземпляра являются $1/2\ (n-1)!$ туров. В задаче о минимизации производственной продолжительности (раздел 20.1.1) допустимыми решениями экземпляра с m машинами и n работами являются m^n разных способов закрепления работ за машинами. В задаче о максимальном охвате (раздел 20.2.1), в экземпляре с m подмножествами и параметром k, допустимыми решениями являются $\binom{m}{k}$ разных способов выборки k из подмножеств.

Шаг 2. Определить целевую функцию. В задаче коммивояжера целевой функцией (которую можно минимизировать) является суммарная стоимость тура. В задачах о минимизации производственной продолжительности и о максимальном охвате целевыми функциями (соответственно, минимизируемой и максимизируемой) являются, естественно, производственная продолжительность плана и охват коллекции из k подмножеств.

Шаги 1 и 2 определяют *глобальные оптимумы* экземпляра — допустимые решения с наилучшим значением целевой функции (как тур со стоимостью 23 на рис. 20.3). Шаг 3 завершает определение метаграфа, определяя его ребра — разрешимые локальные ходы из одного допустимого решения к другому.

Шаг 3. Определить разрешимые локальные ходы. В алгоритме `2OPT` для задачи коммивояжера локальные ходы соответствуют двукратным заменам. В экземпляре с n вершинами *размер окрестности* $n(n-3)/2$ — это число локальных ходов, доступных из каждого решения. Как применить парадигму локального поиска к задачам о минимизации производственной продолжительности и о максимальном охвате? В первой задаче самое простое определение локального хода — это переназначение одной работы другой машине, при котором размер окрестности равен $n(m-1)$, где m и n обозначают соответственно число

машин и работ. Для задачи о максимальном охвате локальный ход меняет одно из k подмножеств текущего решения на другое, причем при m подмножествах размер окрестности равен $k(m - k)$ с k вариантами для исключения и $m - k$ — для подстановки.

После шагов 1–3 полностью задается метаграф допустимых решений, среди которых *локальные оптимумы* — это допустимые решения, из которых нет улучшающего локального хода, то есть вершины метаграфа со значениями целевой функции не хуже, чем у соседей. Например, на рис. 20.3 два локальных оптимума — это первый и третий туры в третьем ряду. (Последний также является глобальным оптимумом, в то время как первый — нет.) Для задачи о минимизации производственной продолжительности, где локальные ходы соответствуют переназначениям одной-единственной работы, план, вырабатываемый алгоритмом `LPT` (тестовое задание 20.3), является локальным минимумом, в то время как план, вырабатываемый алгоритмом `Graham` (тестовое задание 20.2), таковым не является. Для задачи о максимальном охвате, где локальные ходы соответствуют перестановкам подмножеств, выходные данные алгоритма `GreedyCoverage` не являются локальным максимумом ни в одном из примеров раздела 20.2.4.[1] Убедитесь сами.

Пример: окрестность с троекратными заменами для задачи коммивояжера

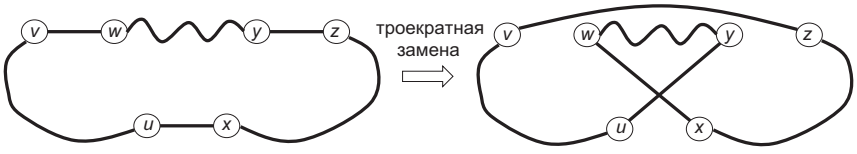
Шаги 1 и 2 не уникально определяют решение на шаге 3, поскольку для данной вычислительной задачи могут иметь смысл несколько определений локального хода. Например, кто говорит, что в задаче коммивояжера мы можем вынуть и положить обратно только два ребра за один раз? Почему не три, а то и больше?

Троекратная замена — это операция, которая заменяет три ребра тура коммивояжера тремя другими ребрами, чтобы получился новый тур:[2]

---

[1] Когда у вас мало времени, пропустите данные на выходе из эвристического алгоритма через этап постобработки локальным поиском. В конце концов решение от этого может стать только лучше!

[2] В операции двукратной замены удаленная пара ребер уникально определяет добавляемую пару (раздел 20.4.2). Это больше не относится к операции троекратной замены. Например, если удалить три ребра без совместных конечных точек, то



Алгоритм `3OPT` является обобщением алгоритма `2OPT` (раздел 20.4.3), выполняющим в каждой итерации главного цикла `while` операции двукратной *или троекратной замены*, которые производят менее дорогой тур.

ТЕСТОВОЕ ЗАДАНИЕ 20.10

В экземпляре задачи коммивояжера обозначить через H_2 и H_3 метаграфы, соответствующие алгоритмам `2OPT` и `3OPT`. Какое из следующих утверждений является истинным? (Правильных ответов может быть несколько.)

а) Каждое ребро метаграфа H_2 также является ребром метаграфа H_3.

б) Каждое ребро метаграфа H_3 также является ребром метаграфа H_2.

в) Каждый локальный минимум метаграфа H_2 также является локальным минимумом метаграфа H_3.

г) Каждый локальный минимум метаграфа H_3 также является локальным минимумом метаграфа H_2.

(Ответ и анализ решения см. в разделе 20.5.8.)

Выбор размера окрестности

Какое из конкурирующих определений локальных ходов нужно использовать? Лучше попробовать несколько вариантов и выбрать подходящий. Но тестовое задание 20.10 иллюстрирует общее преимущество крупных размеров окрестностей: больше локальных ходов означает меньше негодных

получится семь способов сделать пары из их шести конечных точек, которые приводят к новым допустимым турам (убедитесь сами).

локальных оптимумов, в которых застревает поиск. Однако недостатком более крупных размеров окрестностей является замедление проверок наличия улучшающих локальных ходов. Например, в задаче коммивояжера проверка на улучшающую двукратную замену занимает квадратичное время (по числу вершин), в то время как проверка на улучшающую троекратную замену занимает кубическое время. Таким образом, оптимальнее всего использовать самую крупную окрестность, возможную при данном целевом времени выполнения в расчете на каждую итерацию (например, одна секунда или десять секунд).

20.5.4. Два решения по проектированию алгоритма

Шаги 4 и 5 парадигмы локального поиска предоставляют сведения, которых не было в генерическом алгоритме локального поиска из раздела 20.5.2.

Шаг 4. Решить, как выбрать первое допустимое решение. Два простых способа выбрать первое решение — жадно и наугад. В задаче коммивояжера первый тур может быть построен алгоритмом ближайшего соседа (тестовое задание 20.7) или путем равномерного случайного порядка посещения вершин. В задаче о минимизации производственной продолжительности первый план может быть построен алгоритмами `Graham` или `LPT` либо путем независимого закрепления работ за равномерно случайно распределенными машинами. Для задачи о максимальном охвате первым решением может быть результат алгоритма `GreedyCoverage` или случайный выбор равномерно распределенных k из заданных подмножеств.

Зачем отбрасывать хороший жадный эвристический алгоритм в пользу случайного решения? Потому что старт локального поиска из более качественного первого решения не обязательно приведет к более качественному (или такому же хорошему) локальному оптимуму. Идеальная процедура инициализации должна быстро создать не слишком плохое решение, оставив много возможностей для локального улучшения. Случайная инициализация часто отвечает этому критерию.

Шаг 5. Решить, как выбрать один из нескольких улучшающих локальных ходов. Самый простой подход состоит в прокрутке локальных ходов один

за другим, до тех пор пока не будет найден улучшающий.[1] Альтернативой с более медленным временем выполнения за итерацию, но более крупным улучшением целевой функции за итерацию является выполнение этой прокрутки с жадным выполнением локального хода, предлагающего наибольшее улучшение. Третий вариант — стимулирование более широкого исследования пространства решений для выбора одного из улучшающих ходов наугад.

20.5.5. Время выполнения и качество решения

Шаги 1–5 полностью определяют алгоритм локального поиска, который начинается с первого решения (выбранного с помощью процедуры из шага 4) и повторно выполняет улучшающие локальные ходы (выбранные с помощью процедуры из шага 5), до тех пор пока не будет достигнут локальный оптимум и дальнейшие улучшающие ходы не станут невозможными. Какую результативность можно ждать от такого алгоритма?

Все уроки, усвоенные об алгоритме `2OPT` для задачи коммивояжера в разделах 20.4.4–20.4.5, применимы к большинству других алгоритмов локального поиска:

ОБЩИЕ ХАРАКТЕРИСТИКИ ЛОКАЛЬНОГО ПОИСКА

1. Остановка гарантирована. (Исходя из того что существует только конечное число допустимых решений.)
2. Не гарантируется остановка при полиномиальном (по входному размеру) количестве итераций.
3. На реалистичных входных данных почти всегда происходит остановка в пределах допустимого числа итераций.
4. Поиск может быть прерван в любое время, возвращая последнее (и наилучшее) найденное решение.
5. Возврат локального оптимума со значением целевой функции, близким к наилучшему из возможных, не гарантирован.

[1] Алгоритм `2OPT` следовал этому подходу в примере раздела 20.4.3 (исходя из того что на рисунках он всегда сканировал двукратные замены слева направо).

6. На реалистичных входных данных поиск производит низкокачественные локальные оптимумы намного реже, чем высококачественные.

20.5.6. Избегание плохих локальных оптимумов

Низкокачественные локальные оптимумы мешают локальному поиску. Как заставить алгоритм их избегать? Можно увеличить размер окрестности (раздел 20.5.3), но еще проще выбирать на каждой итерации случайное первое решение или случайные улучшающие ходы. Так вы сможете выполнить столько независимых испытаний алгоритма, сколько вам позволяет время, возвращая наилучший локальный оптимум каждого испытания.

Более радикальный подход к избеганию плохих локальных оптимумов — иногда допускать неулучшающие локальные ходы. Например, на каждой итерации:

1) из текущего решения выбрать локальный ход случайным образом;
2) если выбранный локальный ход улучшается, то выполнить его;
3) если выбранный локальный ход ухудшает целевую функцию на $\Delta \geq 0$, то выполнить его с некоторой вероятностью $p(\Delta)$, которая уменьшается на Δ, или не делать ничего. (Одним из популярных вариантов для функции $p(\Delta)$ является экспоненциальная функция $e^{-\lambda\Delta}$, где $\lambda > 0$ — это настраиваемый параметр.)[1]

Алгоритмы локального поиска, которые разрешают неулучшающие ходы, обычно не останавливаются и должны прерываться по истечении заданного времени вычисления.

Число возможных дополнений к базовому алгоритму локального поиска бесконечно.[2] Вот два из них:

---

[1] Если вы слышали об алгоритме Метрополиса или симулированном закаливании (имитации отжига), то оба они основаны на этой идее.

[2] Для глубокого погружения в тему ознакомьтесь с книгой «Локальный поиск в комбинаторной оптимизации» под редакцией Эмиля Аартса и Яна Карела Ленстры («*Local Search in Combinatorial Optimization», ed. Emile Aarts and Jan Karel Lenstra, Princeton University Press*, 2003).

- *Окрестности, зависящие от истории.* Вместо того чтобы фиксировать разрешенные локальные ходы раз и навсегда, их можно сделать зависимыми от текущей траектории алгоритма локального поиска. Вы можете запретить локальные ходы, которые частично обращают вспять предыдущий ход, например двукратную замену с использованием таких же конечных точек, как и в предыдущей двукратной замене[1]. Это также поможет избежать появления циклов, которые разрешают неулучшающие ходы.
- *Поддержание популяции решений.* Алгоритм может поддерживать $k \geq 2$ допустимых решений не только за один раз. Каждая итерация алгоритма будет генерировать k новых допустимых решений из k старых, например, сохраняя только k наилучших соседей текущих k решений или комбинируя пары текущих решений для создания новых.[2]

20.5.7. Когда использовать локальный поиск?

Вы знаете много парадигм проектирования алгоритмов. Так когда локальный поиск будет первым лучшим решением? Если в вашем проекте стоят галочки напротив нескольких следующих пунктов, то локальный поиск, вероятно, стоит попробовать.

КОГДА ИСПОЛЬЗОВАТЬ ЛОКАЛЬНЫЙ ПОИСК

1. Нет времени вычислять точное решение.
2. Можно отказаться от приблизительных гарантий времени выполнения и правильности.
3. Нужен алгоритм, который относительно легко реализовать.
4. Надо улучшить результат эвристического алгоритма на шаге постобработки.
5. Нужен алгоритм, который можно прервать в любое время.

---

[1] Если вы слышали о поиске с запретами (табу-поиске) или эвристике переменной глубины Лина — Кернигана, то оба указанных алгоритма связаны с этой идеей.

[2] Если вы слышали о лучевом поиске (beam search) или генетических алгоритмах, то оба они являются вариантами реализации этой идеи.

6. Не подошли решатели задач MIP и SAT (разделы 21.4–21.5). Возможно, размеры входных данных слишком велики или задачу трудно перевести в нужный формат.

Помните, для того чтобы получить максимальную отдачу от локального поиска, надо экспериментировать — с разными окрестностями, стратегиями инициализации, стратегиями выбора локальных ходов, дополнениями и так далее.

20.5.8. Решения к тестовым заданиям 20.9–20.10

Решение к тестовому заданию 20.9

Правильный ответ: (а). Метаграф имеет одну вершину на тур, в общей сложности $1/2\,(n-1)!$ (см. тестовое задание 19.1). Каждая вершина метаграфа примыкает к $n\,(n-3)/2$ другим вершинам (см. сноску на с. 105). Следовательно, суммарное число ребер равно

$$\frac{1}{2}\times \underbrace{\frac{(n-1)!}{2}}_{\text{число вершин}} \times \underbrace{\frac{n(n-3)}{2}}_{\text{число инцидентных ребер}} = \frac{n!(n-3)}{8}.$$

Ведущий член 1/2 вносит поправку на двойной учет каждого ребра метаграфа (один раз посредством каждой конечной точки).

Решение к тестовому заданию 20.10

Правильные ответы: (а), (г). Алгоритм `3OPT` на каждой итерации может делать любую двукратную (либо троекратную) замену. С каждым локальным ходом, доступным для алгоритма `2OPT`, а значит и для `3OPT`, ответ (а) верен. И (г) тоже верен: если у вершины есть сосед в H_2 с более качественным значением целевой функции (показывающим, что она не является локальным минимумом в H_2), то этот сосед показывает, что эта вершина также не является локальным минимумом в H_3.

В примере на рис. 20.3 первый тур в третьем ряду не может быть улучшен двукратной заменой, но может быть улучшен троекратной заменой (убедитесь сами). Это демонстрирует, что и (б), и (в) являются неправильными ответами.

ВЫВОДЫ

- Цель задачи о минимизации производственной продолжительности состоит в закреплении работы за машинами для уменьшения максимальной машинной загрузки.
- Выполнение одного прохода по работам и планирование прикрепления каждой из них к наименее загруженной в данный момент машине создает план с производственной продолжительностью, не более чем вдвое превышающей минимально возможную.
- Предварительная сортировка работ (от самых длинных до самых коротких) улучшает гарантию с 2 до 4/3.
- Цель задачи о максимальном охвате состоит в выборке k из m подмножеств для максимизации размера их объединения.
- Жадный отбор подмножеств, максимально увеличивающих охват, позволяет получить охват не менее 63,2 % от максимально возможного.
- Влияние множества первых активных вершин в ориентированном графе представляет собой ожидаемое число в конечном счете активированных вершин, исходя из того что активированная вершина активирует каждого из своих внешних соседей с некоторой вероятностью p.
- Цель задачи о максимизации влияния состоит в выборке k вершин ориентированного графа для максимизации их влияния.
- Поскольку влияние является средневзвешенным значением функций охвата, гарантия 63,2 % переносится на жадный алгоритм, который итеративно отбирает вершины, увеличивающие влияние.
- В задаче коммивояжера на вход подается полный граф с вещественнозначными реберными стоимостями для вычисления тура (цикла, посещающего каждую вершину ровно один раз) с минимально возможной суммой реберных стоимостей.
- Операция двукратной замены создает новый тур из старого, меняя одну пару ребер на другую.

- ✶ Алгоритм `2OPT` для задачи коммивояжера многократно улучшает первый тур посредством двукратной замены, до тех пор пока такие улучшения не станут невозможными.
- ✶ Алгоритм локального поиска проходит через метаграф, в котором вершины соответствуют допустимым решениям (помеченным значением целевой функции), а ребра — локальным ходам.
- ✶ Алгоритм локального поиска задается метаграфом, первым решением и правилом для отбора среди улучшающих локальных ходов.
- ✶ Алгоритмы локального поиска часто дают высококачественные решения в разумные сроки, несмотря на отсутствие доказуемого времени работы и приблизительных гарантий правильности.
- ✶ Алгоритмы локального поиска можно настроить так, чтобы они лучше избегали низкокачественных локальных оптимумов, например через рандомизацию и разрешение неулучшающих локальных ходов.

Задачи на закрепление материала

Задача 20.1 *(S)*. В задаче о минимизации производственной продолжительности (раздел 20.1.1) предположим, что работы имеют одинаковую длину (где $\ell_j \leq 2\ell_h$ для всех работ j, h) и что существует нормальное число работ (по меньшей мере в 10 раз больше числа машин). Что можно сказать о продолжительности плана, производимой алгоритмом `Graham` из раздела 20.1.3? (Выбрать самый точный ответ.)

а) Она не более чем на 10 % крупнее минимально возможной производственной продолжительности.

б) Она не более чем на 20 % крупнее минимально возможной производственной продолжительности.

в) Она не более чем 50 % крупнее минимально возможной производственной продолжительности.

г) Она не более чем на 100 % крупнее минимально возможной производственной продолжительности.

Задача 20.2 *(S)*. В задаче о максимальном охвате (раздел 20.2.1) требуется охватить как можно больше элементов с помощью фиксированного числа подмножеств. В тесно связанной с ней задаче о покрытии множества[1] нужно охватить все элементы с помощью как можно меньшего числа подмножеств (например, нанять команду со всеми необходимыми навыками и минимально возможной стоимостью).[2] Жадный алгоритм для задачи о максимальном охвате (раздел 20.2.3) легко распространяется на задачу о покрытии множества (заданную в виде входных m подмножеств $A_1, A_2, \ldots, A_m$ универсального множества U, где $\cup_{i=1}^{m} A_i = U$):

ЖАДНЫЙ ЭВРИСТИЧЕСКИЙ АЛГОРИТМ ДЛЯ ПОКРЫТИЯ МНОЖЕСТВА

$K := \emptyset$ `// индексы выбранных множеств`
while $f_{\text{cov}}(K) < |U|$ **do** `// непокрытая часть U`
    $i^* := \arg\max_{i=1}^{m} \left[f_{\text{cov}}\left(K \cup \{i\}\right) - f_{\text{cov}}\left(K\right) \right]$
    $K := K \cup \{i^*\}$
return K

Обозначим через k минимальное число подмножеств, необходимых для покрытия всего U. Какая из следующих гарантий приближенной правильности соблюдается для этого алгоритма? (Выбрать самый точный ответ.)

а) Его решение состоит не более чем из $2k$ подмножеств.

б) Его решение состоит из $O(k \log |U|)$ подмножеств.

в) Его решение состоит из $O(k \times \sqrt{|U|}$ подмножеств.

г) Его решение состоит из $O(k \times |U|)$ подмножеств.

Задача 20.3 *(S)*. Эта задача рассматривает три жадные эвристики для задачи о рюкзаке (раздел 19.4.2). Входные данные состоят из n предметов со значениями $v_1, v_2, \ldots, v_n$ и размерами $s_1, s_2, \ldots, s_n$, а также вместимости рюкзака C.

---

[1] При наличии множества элементов {1, 2, ..., n} (именуемого универсумом) и коллекции S из m множеств, объединение которых равно универсуму, задача о покрытии множества заключается в выявлении наименьших подмножеств множества S, объединение которых равняется универсуму. — *Примеч. пер.*

[2] Эта задача является NP-трудной; см. задачу 22.6.

ЖАДНЫЙ ЭВРИСТИЧЕСКИЙ АЛГОРИТМ № 1 ДЛЯ РЮКЗАКА

$I := \emptyset$, $S := 0$ // выбранные предметы и их размер
отсортировать и переиндексировать работы так,
    чтобы $v_1 \geq v_2 \geq \cdots \geq v_n$
for i = 1 to n **do**
    if $S + s_i \leq C$ **then** // выбрать предмет, если он допустим
        $I := I \cup \{i\}$, $S := S + s_i$
return I

ЖАДНЫЙ ЭВРИСТИЧЕСКИЙ АЛГОРИТМ № 2 ДЛЯ РЮКЗАКА

$I := \emptyset$, $S := 0$ // выбранные предметы и их размер
отсортировать и переиндексировать работы так,
    чтобы $v_1/s_1 \geq v_2/s_2 \geq \cdots \geq v_n/s_n$
for i = 1 to n **do**
    if $S + s_i \leq C$ **then** // выбрать предмет, если он допустим
        $I := I \cup \{i\}$, $S := S + s_i$
return I

ЖАДНЫЙ ЭВРИСТИЧЕСКИЙ АЛГОРИТМ № 3 ДЛЯ РЮКЗАКА

I_1 := результат жадного эвристического алгоритма № 1
I_2 := результат жадного эвристического алгоритма № 2
вернуть любой результат из I_1 или I_2, который имеет
    большее суммарное значение

Какое из следующих утверждений истинно? (Правильных ответов может быть несколько.)

а) Суммарное значение решения, возвращаемое первым жадным алгоритмом, всегда составляет по меньшей мере 50 % от максимально возможного.

б) Суммарное значение решения, возвращаемое вторым жадным алгоритмом, всегда составляет по меньшей мере 50 % от максимально возможного.

в) Суммарное значение решения, возвращаемое третьим жадным алгоритмом, всегда составляет по меньшей мере 50 % от максимально возможного.

г) Если размер каждого элемента составляет не более 10 % вместимости рюкзака (то есть $\max_{i=1}^{n} s_i \leq C / 10$), то суммарное значение решения, возвращаемое первым жадным алгоритмом, составляет по меньшей мере 90 % от максимально возможного.

д) Если размер каждого элемента составляет не более 10 % вместимости рюкзака, то суммарное значение решения, возвращаемое вторым жадным алгоритмом, составляет по меньшей мере 90 % от максимально возможного.

е) Если размер каждого элемента составляет не более 10 % вместимости рюкзака, то суммарное значение решения, возвращаемое третьим жадным алгоритмом, составляет по меньшей мере 90 % от максимально возможного.

Задача 20.4 *(S)*. В задаче о покрытии множества на вход подается неориентированный граф $G = (V, E)$. Нужно определить подмножество вершин $S \subseteq V$ минимального размера, которое включает хотя бы одну конечную точку каждого ребра из E.[1] (Если ребра представляют собой дороги, а вершины — перекрестки, то нужно контролировать все дороги, устанавливая камеры безопасности на как можно меньшем числе перекрестков.) Один простой эвристический алгоритм многократно выбирает еще не покрытое ребро и добавляет обе его конечные точки в свое текущее решение:

ЭВРИСТИЧЕСКИЙ АЛГОРИТМ ДЛЯ ПОКРЫТИЯ МНОЖЕСТВА

$S := \emptyset$ `// выбранные вершины`

while имеется ребро $(v, w) \in E$, где $v, w \notin S$ **do**

    $S := S \cup \{v, w\}$ `// добавить обе конечные точки ребра`

return S

[1] Эта задача является NP-трудной (см. задачу 22.5).

Обозначим через k минимальное число вершин, необходимое для охвата по меньшей мере одной конечной точки каждого ребра. Какая из следующих гарантий приближенной правильности соблюдается для этого алгоритма? (Выбрать самый точный ответ.)

а) Его решение состоит не более чем из $2k$ вершин.

б) Его решение состоит из $O(k \log |E|)$ вершин.

в) Его решение состоит из $O(k \times \sqrt{|E|})$ вершин.

г) Его решение состоит из $O(k \times |E|)$ вершин.

Задача 20.5 *(S)*. Какое из следующих утверждений о генерическом алгоритме локального поиска из раздела 20.5.2 не является истинным?

а) Его результат в целом зависит от выбора первого допустимого решения.

б) Его результат обычно зависит от метода выбора одного улучшающего локального хода из многих.

в) Он всегда, в конечном счете, будет останавливаться на оптимальном решении.

г) В некоторых случаях он выполняет экспоненциальное (по размеру входных данных) число итераций перед остановкой.

Задачи повышенной сложности

Задача 20.6 *(H)*. Предложите реализацию алгоритма `Graham` (раздел 20.1.3), которая выполняется на кучевой структуре данных за время $O(n \log m)$, где n — это число работ, и m — число машин.[1]

Задача 20.7 *(H)*. Эта задача совершенствует теорему 20.4 и расширяет пример из тестового задания 20.3, идентифицируя наилучшую возможную гарантию приближенной правильности для алгоритма `LPT` (раздел 20.1.7).

---

[1] Технически время выполнения будет равно $O(m + n \log m)$. Однако задача неинтересна, когда $n \leq m$, так как в этом случае каждой работе может быть предоставлена выделенная машина.

а) Пусть j будет последней работой, закрепленной за наиболее загруженной машиной в плане, возвращаемом алгоритмом `LPT`. Докажите, что если $\ell_j > M^*/3$, где M^* обозначает минимально возможную производственную продолжительность, то этот план является оптимальным (то есть имеет производственную продолжительность M^*).

б) Докажите, что алгоритм `LPT` всегда выводит план с производственной продолжительностью не более чем в $4/3 - 1/3m$ раз больше минимально возможной, где m — это число машин.

в) Обобщите пример из тестового задания 20.3, чтобы показать, что для каждого $m \geq 1$ существует пример с m машинами, в котором план, производимый алгоритмом `LPT`, имеет продолжительность в $4/3 - 1/3m$ раз больше возможной.

Задача 20.8 *(H)*. Вспомните плохой пример для алгоритма `GreedyCoverage` из тестового задания 20.5.

а) Докажите утверждение 20.6.

б) Расширьте свои примеры в (а), чтобы показать, что с разрывом связей даже в лучшем случае для каждой константы $\epsilon > 0$ алгоритм `GreedyCoverage` не гарантирует долю $1 - (1 - 1/k)^k + \epsilon$ максимально возможного охвата (где k обозначает число выбранных подмножеств).

Задача 20.9 *(H)*. Покажите, что каждый экземпляр задачи о максимальном охвате может быть закодирован как экземпляр задачи о максимизации влияния, причем: (1) оба экземпляра будут иметь одинаковое оптимальное значение целевой функции F^*; (2) любое решение для последнего экземпляра с влиянием F может быть легко конвертировано в решение для первого экземпляра с охватом по меньшей мере F.

Задача 20.10 *(H)*. Цель задачи о максимальном охвате состоит в выборке k подмножеств для максимизации охвата f_{cov}. Цель задачи о максимизации влияния состоит в выборке k вершин для максимизации влияния f_{inf}. В обобщенной версии этих задач требуется: получив множество O объектов и вещественнозначную функцию множеств f (задающую число $f(S)$ для каждого подмножества $S \subseteq O$), выбрать k объектов из O, чтобы максимизировать f. Алгоритмы `GreedyCoverage` и `GreedyInfluence` распространяются естественным образом на эту общую задачу.

ЖАДНЫЙ АЛГОРИТМ МАКСИМИЗАЦИИ ФУНКЦИИ МНОЖЕСТВА

$S := \emptyset$ // выбранные объекты
for $j = 1$ to k **do** // выбрать объекты один за другим
    // жадно увеличивать целевую функцию
    $o^* := \text{argmax}_{o \notin S}[f(S \cup \{o\}) - f(S)]$
    $S := S \cup \{o^*\}$
return S

Для каких целевых функций f этот жадный алгоритм обладает гарантией приближенной правильности (как в теоремах 20.7 и 20.9)? Вот их ключевые свойства:

1. *Неотрицательность*: $f(S) \geq 0$ для всех $S \subseteq O$.
2. *Монотонность*: $f(S) \geq f(T)$ всякий раз, когда $S \supseteq T$.
3. *Субмодульность*: $f(S \cup \{o\}) - f(S) \leq f(T \cup \{o\}) - f(T)$ всякий раз, когда $S \supseteq T$ и $o \notin S$.[1]

а) Доказать, что функции охвата и влияния f_{cov} и f_{inf} обладают всеми тремя свойствами.

б) Доказать, что всякий раз, когда f является неотрицательной, монотонной и субмодульной, общий жадный эвристический алгоритм гарантированно возвращает множество S объектов, удовлетворяющих

$$f(S) \geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \times f\left(S^*\right),$$

где S^* максимизирует f над всеми k-размерными подмножествами множества O.

Задача 20.11 *(H)*. Задача 20.3 расследовала гарантии приближенной правильности жадных эвристических алгоритмов для задачи о рюкзаке. Она описы-

[1] Субмодульность (submodularity) отстаивает свойство убывающей возвратности (или убывающей отдачи): предельная ценность нового объекта o может убывать только, когда приобретаются другие объекты.

вала алгоритм динамического программирования с сильной гарантией: для заданного пользователем параметра ошибки $\epsilon > 0$ (наподобие 0,1 или 0,01) алгоритм выдавал решение с суммарным значением не менее чем в $1 - \epsilon$ раз больше максимально возможного. (Для NP-трудной задачи это прозвучит слишком хорошо, чтобы быть правдой, но я открою вам секрет: время выполнения алгоритма претерпевает взрывной рост по мере приближения ϵ к 0.)

а) В разделе 19.4.2 упоминалось, что задача о рюкзаке может быть решена за время $O(nC)$ с помощью динамического программирования, где n обозначает число предметов, а C — вместимость рюкзака (см. также главу 16 *Третьей части*). (Все значения и размеры предметов, а также вместимость рюкзака являются целыми положительными числами.) Приведите другой алгоритм динамического программирования для задачи, которая выполняется за время $O(n^2 \times v_{\max})$, где $v_{\max}$ обозначает наибольшую стоимость любого предмета.

б) Для того чтобы уменьшить стоимости предметов до приемлемой величины, разделите каждое из них на $m := (\epsilon \times v_{\max})/n$ и округлите каждый результат до ближайшего целого числа (где ϵ — это заданный пользователем параметр ошибки). Докажите, что суммарное значение каждого возможного решения уменьшается как минимум в m раз, а суммарное значение оптимального решения снижается как минимум в $m/(1 - \epsilon)$ раз. (Можете исходить из того, что размер каждого предмета не превышает C и, следовательно, помещается в рюкзаке.)

в) Предложите $O(n^3/\epsilon)$-временной алгоритм, который гарантированно возвращает допустимое решение с суммарным значением как минимум в $1 - \epsilon$ раз больше максимально возможного.[1]

Задача 20.12 *(H)*. В часто встречающемся метрическом экземпляре задачи коммивояжера все реберные стоимости c_e в графе $G = (V, E)$ являются неотрицательными, и кратчайший путь между любыми двумя вершинами является прямым путем с одним переходом (условие называется неравенством треугольника):

$$c_{uv} \leq \sum_{e \in P} c_e$$

[1] Эвристический алгоритм с таким типом гарантии называется схемой полностью полиномиально-временной аппроксимации (FPTA, fully polynomial-time approximation scheme).

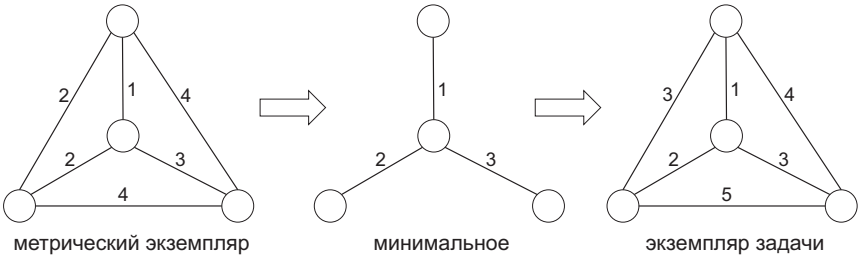
для каждой пары $v, w \in V$ вершин и v-w-вершинного пути P. (Пример в тестовом задании 19.2 является метрическим экземпляром, а пример в тестовом задании 20.7 — нет.) Неравенство треугольника, как правило, соблюдается в приложениях, где реберные стоимости соответствуют физическим расстояниям. Задача коммивояжера остается NP-трудной в частном случае метрических экземпляров (см. задачу 22.12(а)).

Отправной точкой для быстрого эвристического алгоритма станет линейновременной алгоритм для древовидных экземпляров (из задачи 19.8). Попробуйте свести общий метрический экземпляр к древовидному, вычислив минимальное остовное дерево.

ЭВРИСТИКА МИНИМИЗАЦИИ ОСТОВНОГО ДЕРЕВА ДЛЯ МЕТРИЧЕСКОЙ ЗАДАЧИ КОММИВОЯЖЕРА

- T := минимальное остовное дерево входного графа G;
- вернуть оптимальный обход экземпляра дерева, определенный деревом T.

Первый шаг может быть реализован за почти линейное время алгоритмом Прима или Краскала. Второй шаг может быть реализован за линейное время решением задачи 19.8. В древовидном экземпляре задачи коммивояжера, построенном на втором шаге, длина a_e ребра e дерева T устанавливается равной стоимости c_e этого ребра в данном метрическом экземпляре задачи коммивояжера (при этом стоимость каждого ребра (v, w) в древовидном экземпляре тогда определяется как суммарная длина $\sum_{e \in P_{vw}} a_e$ уникального v-w-вершинного пути P_{vw} в T):



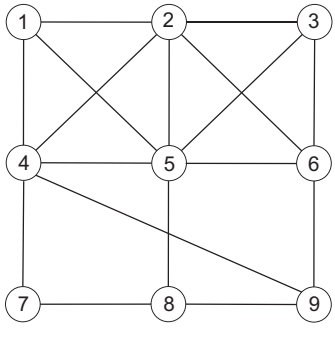
метрический экземпляр задачи коммивояжера | минимальное остовное дерево | экземпляр задачи коммивояжера

а) Докажите, что минимальная суммарная стоимость обхода коммивояжера равна по меньшей мере стоимости минимального остовного дерева. (Этот шаг не требует неравенства треугольника.)

б) Докажите, что для каждого метрического экземпляра задачи коммивояжера суммарная стоимость обхода, вычисленная эвристикой минимального остовного дерева, не более чем в два раза больше минимально возможной.

Задача 20.13 *(H)*. Предложите реализацию алгоритма `2OPT` (раздел 20.4.3), в котором каждая итерация главного цикла `while` выполняется за время $O(n^2)$, где n — это число вершин.

Задача 20.14 *(S)*. Большинство алгоритмов локального поиска не гарантируют выполнения за полиномиальное время и приближенную правильность. Эта задача описывает редкое исключение. Для целого числа $k \geq 2$ в задаче о максимальном k-членном разрезе на вход подается неориентированный граф $G = (V, E)$. Допустимыми решениями являются k-членные разрезы графа — разбиения множества вершин V на k непустых групп $S_1, S_2, ..., S_k$. Цель — максимизировать число ребер с конечными точками в разных группах. Например, в графе



шестнадцать из семнадцати ребер имеют конечные точки в разных группах трехчленного разреза ({1, 6, 7}, {2, 5, 9}, {3, 4, 8}).[1]

[1] Сделать лучше невозможно, так как две вершины в {1, 2, 4, 5} должны принадлежать к общей группе.

Для k-членного разреза (S_1, S_2, ..., S_k) каждый локальный ход соответствует переназначению одной вершины из одной группы в другую с учетом ограничения, что ни одна из k групп не может стать пустой.

а) Докажите, что для каждого первого k-членного разреза и отборочного правила для выбора улучшающих локальных ходов генерический алгоритм локального поиска останавливается в пределах $|E|$ итераций.

б) Докажите, что для каждого первого k-членного разреза и отборочного правила для выбора улучшающих локальных ходов генерический алгоритм локального поиска останавливается на k-членном разрезе, имеющем значение целевой функции как минимум в $(k - 1)/k$ раз больше максимально возможного.

Задачи по программированию

Задача 20.15. Реализуйте на своем любимом языке программирования алгоритм ближайшего соседа для задачи коммивояжера (как в тестовом задании 20.7). Испытайте свою реализацию со стоимостями ребер, выбираемыми независимо и равномерно случайным образом из множества {1, 2, …, 100}, или стоимостями вершин, соответствующих точкам, выбираемым независимо и равномерно случайным образом из единичного квадрата.[1] Насколько большим будет размер входных данных (то есть число вершин), который ваша программа способна надежно обрабатывать менее чем за минуту? А что будет через час? (Тестовые случаи и наборы данных для сложных задач см. на веб-сайте www.algorithmsilluminated.org.)

Задача 20.16. Реализуйте на своем любимом языке программирования алгоритм `2OPT` (из раздела 20.4.3). Используйте свою реализацию алгоритма ближайшего соседа (из задачи 20.15) для вычисления первого тура. Реа-

[1] То есть координаты x и y каждой точки являются независимыми и равномерно распределенными случайными числами в [0, 1]. А стоимость ребра, соединяющего две точки (x_1, y_1) и (x_2, y_2), определяется как евклидово (то есть прямолинейное) расстояние между ними, то есть $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. (Для алгоритма ближайшего соседа вы тоже можете работать с квадратичными евклидовыми расстояниями.)

лизуйте каждую итерацию главного цикла так, чтобы она выполнялась за квадратичное время (см. задачу 20.13), и поэкспериментируйте с разными способами выбора улучшающего локального хода. Испытайте свою реализацию на экземплярах, которые вы использовали для задачи 20.15.[1] Насколько локальный поиск улучшает суммарную стоимость первого тура? Какое из ваших отборочных правил приводит к наиболее значительному улучшению? (Тестовые случаи и наборы данных для сложных задач см. на веб-сайте www.algorithmsilluminated.org.)

[1] Для точек в единичном квадрате есть ли разница, что вы используете: евклидовы или квадратичные евклидовы расстояния?

21 Компромисс в отношении скорости: точные неэффективные алгоритмы

Рассмотрим еще один вынужденный компромисс в выборе алгоритма для NP-трудных задач. Когда правильность решения не должна ставиться под угрозу и эвристические алгоритмы не подходят, самое время рассмотреть правильные алгоритмы, которые не всегда выполняются за полиномиальное время. Нацелимся на разработку универсального и правильного алгоритма, который будет выполняться максимально быстро (безусловно, быстрее исчерпывающего поиска) на как можно большем объеме входных данных. Разделы 21.1 и 21.2 используют динамическое программирование для разработки алгоритмов, которые всегда работают быстрее исчерпывающего поиска в задаче коммивояжера и задаче о поиске пути указанной длины в графе. Разделы 21.3–21.5 вводят решатели задач MIP и SAT, не обещающие обогнать исчерпывающий поиск, но полезные при решении экземпляров NP-трудных задач на практике.

21.1. Алгоритм Беллмана — Хелда — Карпа для задачи коммивояжера

21.1.1. Базовый уровень: исчерпывающий поиск

В задаче коммивояжера (раздел 19.1.2) на вход подается полный граф $G = (V, E)$ с вещественнозначными реберными стоимостями для вычисления

тура — цикла, посещающего каждую вершину ровно один раз с минимально возможной суммой реберных стоимостей. Задача коммивояжера является NP-трудной (раздел 22.7). Если правильность не может быть поставлена под угрозу, мы используем алгоритм, который в худшем случае работает со сверхполиномиальным (и предположительно экспоненциальным) временем (исходя из истинности предположения, что $P \neq NP$). Сможет ли алгоритмическая изобретательность превзойти исчерпывающий поиск? На какое ускорение можно надеяться?

Решение задачи коммивояжера путем исчерпывающего поиска среди $1/2\,(n-1)!$ возможных обходов (тестовое задание 19.1) в результате приводит к $O(n!)$-временному алгоритму. Факториальная функция $n! = n \times (n-1)\,(n-2) \times \times 2 \times 1$, безусловно, растет быстрее, чем простая экспоненциальная функция, такая как 2^n, ведь последняя является произведением n двоек, а первая — произведением n членов, которые в основном намного больше 2. Насколько она увеличивается? Существует удивительно точный ответ на этот вопрос, именуемый *аппроксимацией Стирлинга.*[1] (В ней $e = 2{,}718\ldots$ обозначает число Эйлера и, конечно же, $\pi = 3{,}14\ldots$)

АППРОКСИМАЦИЯ СТИРЛИНГА

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \qquad (21.1)$$

Аппроксимация Стирлинга показывает, что факториальная функция (с ее зависимостью n^n-типа) растет *намного* быстрее, чем 2^n. Например, на современных компьютерах до полного завершения можно выполнять $n!$-временной алгоритм только для $n \leq 15$, в то время как 2^n-временной алгоритм может обрабатывать размеры входных данных примерно до $n = 40$. (Да, не впечатляет, но для NP-трудной задачи — неплохо!) Таким образом, заниматься решением задачи коммивояжера за время ближе к 2^n, чем $n!$, — достойная цель.

[1] Запоминайте название формулы, а не ее запись или доказательство. Вы всегда сможете найти ее в «Википедии» или в другом месте, когда потребуется.

21.1.2. Динамическое программирование

Многие крутые приложения динамического программирования подходят задачам, поддающимся решению за полиномиальное время. Но и в решении NP-трудных задач они быстрее исчерпывающего поиска, включая задачу о рюкзаке (раздел 19.4.2), задачу коммивояжера (этот раздел) и другие (раздел 21.2). Напомню (глава 16 *Третьей части*), парадигма динамического программирования такова:

ПАРАДИГМА ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ

1. Выявить относительно малую коллекцию подзадач.
2. Показать, как быстро и правильно решать «бóльшие» подзадачи, получив решения «меньших» подзадач.
3. Показать, как быстро и правильно логически выводить окончательное решение из решений всех подзадач.

После реализации этих трех шагов алгоритм динамического программирования разворачивается сам собой: систематически решает все подзадачи одну за другой, двигаясь от «меньшей» к «большей», и извлекает окончательное решение.

Предположим, что алгоритм динамического программирования решает не более $f(n)$ разных подзадач (двигаясь от «меньших» к «большим»), используя время не более $g(n)$ для каждой, и выполняет объем постобработки не более $h(n)$ по извлечению окончательного решения (где n — это размер входных данных). Тогда время работы алгоритма составляет не более

$$\underbrace{f(n)}_{\text{число подзадач}} \times \underbrace{g(n)}_{\substack{\text{время на подзадачу} \\ \text{(при наличии предыдущих решений)}}} + \underbrace{h(n)}_{\text{постобработка}} \tag{21.2}$$

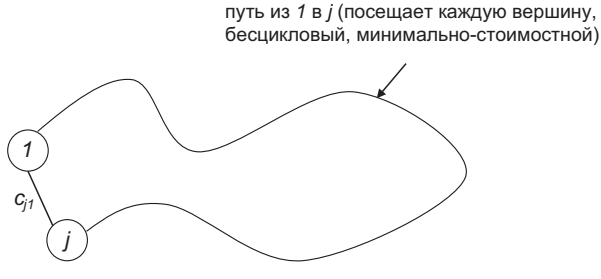
Применяя динамическое программирование к NP-трудной задаче, мы должны ожидать, что по меньшей мере одна из функций $f(n)$, $g(n)$ или $h(n)$ будет экспоненциальной в n. Оглянувшись на некоторые канонические алгоритмы динамического программирования, мы увидим, что функции $g(n)$ и $h(n)$ почти всегда имеют сложность $O(1)$ или $O(n)$, в то время как число $f(n)$ подзадач

широко варьируется от алгоритма к алгоритму.[1] Поэтому будьте готовы к тому, что алгоритм динамического программирования для задачи коммивояжера будет использовать экспоненциальное число подзадач.

21.1.3. Оптимальная подструктура

Динамическое программирование хорошо выявляет правильную коллекцию подзадач. Лучший способ сфокусироваться на них — это продумать разные способы построения оптимального решения из оптимальных решений «меньших» подзадач.

Представьте, что вам достался самый дешевый тур коммивояжера T из вершин $V = \{1, 2, ..., n\}$, с $n \geq 3$. Как он выглядит? Сколькими разными способами можно его построить из оптимальных решений «меньших» подзадач? Подумайте о туре T, который начинается и заканчивается в вершине 1, и сконцентрируйтесь на его последнем решении — финальном ребре, ведущем из некоторой вершины j назад в стартовую точку 1. Идентифицировав j, мы узнаем, как выглядит тур — самый дешевый бесцикловый путь из 1 в j, который посещает каждую вершину с последующим ребром из j назад в 1:[2]



[1] Например, алгоритм динамического программирования для задачи о взвешенном независимом множестве в путевых графах решает $O(n)$ подзадач (где n обозначает число вершин), в то время как для задачи о рюкзаке он решает $O(nC)$ подзадач (где n обозначает число предметов и C — вместимость рюкзака). Алгоритмы кратчайшего пути Беллмана — Форда и Флойда — Уоршелла используют соответственно $O(n^2)$ и $O(n^3)$ подзадач (где n — это число вершин).

[2] Почему $T - \{(j,1)\}$ должен быть самым дешевым путем? Потому что, если бы существовал еще более дешевый бесцикловый 1-j-путь, посещающий каждую вершину, мы подключили бы ребро (j, 1) обратно, чтобы восстановить более дешевый тур (что противоречит оптимальности пути T).

Таким образом, существует только $n - 1$ претендентов на звание оптимального тура (по одному для каждого варианта $j \in \{2, 3, \ldots, n\}$ финальной вершины), и лучшим из них должен быть самый дешевый тур:[1]

$$\text{стоимость оптимального тура} = \min_{j=2}^{n} \left(\begin{array}{c} \text{минимально-стоимостной} \\ \text{бесцикловый путь,} \\ \text{посещающий каждую вершину} \end{array} + c_{j1} \right) \tag{21.3}$$

Пока все хорошо. Но дальше аргументация станет сложнее. Рассмотрим оптимальное решение одной из $n - 1$ подзадач — самый дешевый путь из 1 в j, который посещает каждую вершину ровно один раз (то есть является бесцикловым). Как он выглядит?

ТЕСТОВОЕ ЗАДАНИЕ 21.1

Обозначим через P самый дешевый бесцикловый путь из 1 в j, который посещает каждую вершину с финальным переходом (k, j). Пусть P' обозначает P, из которого удален финальный переход (k, j). Какое из следующих утверждений истинно? (Правильных ответов может быть несколько.)

а) P' — это бесцикловый путь из 1 в k, который посещает каждую вершину $V - \{j\}$.

б) P' — это самый дешевый путь формы (а).

в) P' — это бесцикловый путь из 1 в k, который посещает каждую вершину $V - \{j\}$ и не посещает вершину j.

г) P' — это самый дешевый путь формы (в).

(Ответ и анализ решения см. в разделе 21.1.7.)

Решение задачи 21.1 доказывает следующую лемму.

[1] Рассуждая рекурсивно, самый дешевый тур может быть вычислен путем перебора $n - 1$ вариантов для вершины j, и на каждой итерации рекурсивного вычисления самого дешевого бесциклового 1-j-пути, который посещает каждую вершину.

Лемма 21.1 (оптимальная для задачи коммивояжера подструктура). *Допустим, что $n \geq 3$, P — самый дешевый бесцикловый путь из вершины 1 в вершину j, который посещает каждую вершину $V = \{1, 2, ..., n\}$, и его финальный переход равен (k, j). 1-k-вершинный подпуть P' — это самый дешевый бесцикловый 1-k-вершинный путь, который посещает исключительно вершины $V - \{j\}$.*

Другими словами, как только вы узнаете последний переход оптимального пути, вы узнаете, как должна выглядеть остальная его часть.

Подзадача, решаемая с помощью подпути P' в лемме 21.1, указывает точное подмножество посещаемых вершин. Плохая новость: это побудит алгоритм динамического программирования использовать подзадачи, индексируемые подмножествами вершин (которых, к сожалению, существует экспоненциальное число). Хорошая новость: эти подзадачи не будут описывать порядок посещения вершин, и их число будет масштабироваться с 2^n, а не с $n!$.[1]

21.1.4. Рекуррентное уравнение

Лемма 21.1 сужает возможности оптимального пути из вершины 1 в вершину j до $n - 2$ (и только $n - 2$) кандидатов — по одному для каждого варианта предпоследней вершины k. Лучший из этих $n - 2$ кандидатов станет оптимальным путем.

Следствие 21.2 (рекуррентное уравнение задачи коммивояжера). *С допущениями леммы 21.1 обозначим через $C_{S,j}$ минимальную стоимость бесциклового пути, который начинается в вершине 1, заканчивается в вершине $j \in S$ и посещает исключительно вершины в подмножестве $S \subseteq V$. Тогда для каждой $j \in V - \{1\}$*

$$C_{S,j} = \min_{\substack{k \in S \\ k \neq 1, j}} (C_{S-\{j\},k} + c_{kj}). \tag{21.4}$$

---

[1] По той же причине требуемая алгоритмом память также будет масштабироваться с 2^n (в отличие от исчерпывающего поиска, который использует минимальную память).

В более общем случае для каждого подмножества $S \subseteq V$, содержащего 1 и по меньшей мере две другие вершины, и для каждой вершины $j \in S - \{1\}$

$$C_{S,j} = \min_{\substack{k \in S \\ k \neq 1,j}} (C_{S-\{j\},k} + c_{kj}). \tag{21.5}$$

Второе утверждение в следствии 21.2 вытекает из применения к вершинам S первого утверждения, рассматриваемого как экземпляр задачи коммивояжера (с реберными стоимостями, унаследованными от исходного экземпляра). Min в рекуррентных уравнениях (21.4) и (21.5) реализует исчерпывающий поиск среди кандидатов, чтоб найти предпоследнюю вершину оптимального решения.

21.1.5. Подзадачи

Из диапазона всех соответствующих значений параметров S и j в рекуррентном уравнении (21.5) мы получим коллекцию подзадач. Базовые случаи соответствуют подмножествам формы $\{1, j\}$ для некоторых $j \in V - \{1\}$.[1]

ЗАДАЧА КОММИВОЯЖЕРА: ПОДЗАДАЧИ

Вычислить $C_{S,j}$, минимальную стоимость бесциклового пути из вершины 1 в вершину j, который посещает исключительно вершины в S.

(Для каждого $S \subseteq \{1, 2, ..., n\}$, содержащего вершину 1 и по меньшей мере еще одну вершину, и каждой $j \in S - \{1\}$.)

Тождество в (21.3) показывает, как вычислить минимальную стоимость тура из решений самых крупных подзадач (при $S = V$):

$$\text{стоимость оптимального тура} = \min_{j=2}^{n} (C_{1,n,j} + c_{j1}). \tag{21.6}$$

[1] Рассуждая рекурсивно, каждое применение рекуррентного уравнения (21.5) эффективно удаляет одну вершину (отличную от 1) из дальнейшего рассмотрения. Эти варианты вершин произвольны, поэтому будьте готовы к любому подмножеству вершин (которое содержит 1 и по меньшей мере еще одну другую вершину).

21.1.6. Алгоритм Беллмана — Хелда — Карпа

Получив подзадачи, рекуррентное уравнение (21.5) и шаг постобработки (21.6), алгоритм динамического программирования для задачи коммивояжера разворачивается сам собой. Отслеживанию подлежат $2^{n-1} - 1$ вариантов S (по одному на непустое подмножество из $\{2, 3, ..., n\}$), а размер подзадачи измеряется числом посещаемых вершин (размером S). Для базового варианта с подмножеством $S = \{1, j\}$ единственный вариант — это однопереходной 1-j-вершинный путь со стоимостью c_{1j}. В следующем псевдокоде куча подзадач индексируется подмножествами S вершин. Данная реализация кодирует эти подмножества целыми числами.[1]

BELLMANHELDKARP

Вход: полный неориентированный граф $G = (V, E)$, где $V = \{1, 2, ..., n\}$ и $n \geq 3$, а также вещественнозначная стоимость c_{ij} для каждого ребра $(i, j) \in E$.

Выход: минимальная суммарная стоимость тура коммивояжера в графе G.

```
// подзадачи (1 ∈ S, |S| ≥ 2, j ∈ V − {1})
// (используются только подзадачи с j ∈ S)
A := (2^(n-1) − 1) × (n − 1) // двумерный массив
// базовые случаи (|S| = 2)
for j = 2 to n do
    A[{1, j}][j] := c_1j
// систематически решить все подзадачи
for s = 3 to n do                    // s=размер подзадачи
    for S with |S| = s and 1 ∈ S do
        for j ∈ S − {1} do
            // использовать рекуррентное уравнение
            // из следствия 21.2
```

---

[1] Например, подмножества $V - \{1\}$ можно представить битовыми массивами длины $(n - 1)$, интерпретируемыми как двоичные разложения целых чисел между 0 и $2^{n-1} - 1$.

$$A[S][j] := \min_{\substack{k \neq S \\ k \neq 1, j}} (A[S - \{j\}][k] + c_{kj})$$

```
// использовать (21.6) для вычисления оптимальной
   стоимости тура
return min_{j=2}^{n} (A[V][j] + c_{j1})
```

На момент итерации цикла, ответственной за вычисление решения подзадачи $A[S][j]$, все члены формы $A[S - \{j\}][k]$ уже вычислены в предыдущей итерации самого внешнего цикла `for` (либо в базовых случаях). Эти значения готовы и ждут, чтобы их нашли за постоянное время.[1, 2]

Правильность алгоритма `BellmanHeldKarp` следует по индукции (на размере подзадачи), причем рекуррентное уравнение в следствии 21.2 оправдывает индуктивный шаг, а тождество (21.6) — заключительный шаг постобработки.[3]

Каково время выполнения? Базовые случаи и шаг постобработки занимают время $O(n)$. Подзадач всего $(2^{n-1} - 1)(n - 1) = O(n2^n)$. Решение подзадачи сводится к вычислению минимума во внутреннем цикле, которое занимает время $O(n)$. Тогда общее время работы составляет $O(n^2 2^n)$[4, 5].

---

[1] Алгоритм был предложен независимо Ричардом Э. Беллманом в статье «Трактовка задачи коммивояжера с точки зрения динамического программирования» («*Dynamic Programming Treatment of the Travelling Salesman Problem», Richard E. Bellman, Journal of the ACM*, 1962) и Майклом Хелдом и Ричардом М. Карпом в статье «Подход на основе динамического программирования к задачам секвенирования» («*A Dynamic Programming Approach to Sequencing Problems», Michael Held and Richard M. Karp, Journal of the Society for Industrial and Applied Mathematics*, 1962).

[2] Пример алгоритма `BellmanHeldKarp` см. в задаче 21.2.

[3] См. приложение А *Первой части* или ресурсы по адресу www.algorithmsilluminated.org.

[4] В обозначении (21.2) $f(n) = O(n2^n)$, $g(n) = O(n)$ и $h(n) = O(n)$.

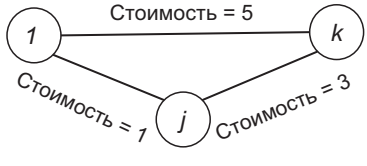
[5] Анализ времени выполнения исходит из того, что подмножества S с заданными размерами $s \geq 3$ и $1 \in S$ перечисляются во времени, линейном по их числу $\binom{n-1}{s-1}$, например, путем рекурсивного перечисления. (Если вы хотите рискнуть и залезть в дебри, поищите «хак Госпера» (Gosper's hack).)

Теорема 21.3 (свойства алгоритма `BellmanHeldKarp`). *Для каждого полного графа $G = (V, E)$ с $n \geq 3$ вершинами и вещественнозначными реберными стоимостями алгоритм* `BellmanHeldKarp` *выполняется за время $O(n^2 2^n)$ и возвращает минимальную стоимость тура.*

Алгоритм `BellmanHeldKarp` вычисляет суммарную стоимость оптимального тура, а не сам оптимальный тур. Как принято в алгоритмах динамического программирования, вы можете восстановить оптимальное решение на шаге постобработки через заполненную кучу подзадач (задача 21.6).

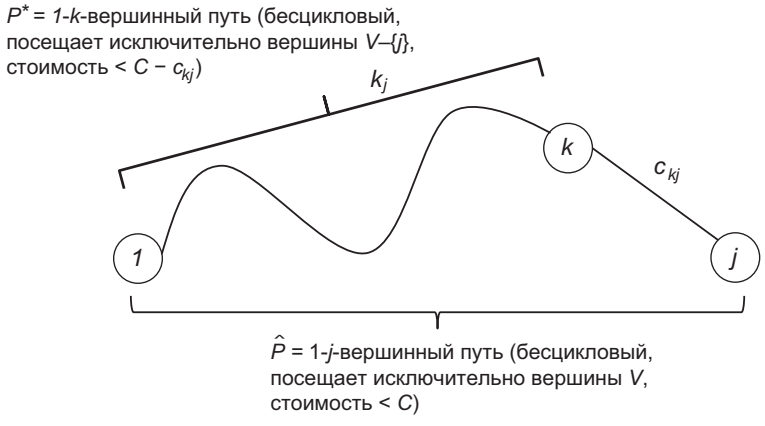
21.1.7. Решение к тестовому заданию 21.1

Правильные ответы: (а), (в), (г). Поскольку P — это бесцикловый путь из 1 в j, посещающий каждую вершину с финальным переходом (k, j), подпуть P' — это бесцикловый путь из 1 в k, который посещает все вершины $V - \{j\}$, но не j. Таким образом, (а) и (в) являются правильными ответами. Ответ (б) не верный, потому что P', возможно, не сможет конкурировать с бесцикловыми путями из 1 в k, которые посещают не только каждую вершину $V - \{j\}$, но и j:



Можно доказать (г) от противного.[1] Обозначим через C суммарную стоимость пути P, так чтобы стоимость P' была равна $C - c_{kj}$. Введем еще один бесцикловый путь P^* из 1 в k, который посещает каждую вершину в $V - \{j\}$, не посещает j и имеет стоимость $C^* < C - c_{kj}$. Тогда добавление ребра (k, j) в P^* произведет путь P из 1 в j с суммарной стоимостью $C^* + c_{kj} < C$:

[1] Вспомните, что в этом типе доказательства есть допущение, *противоречащее* тому утверждению, которое вы доказываете и на котором строится последовательность логически верных шагов, приводящих к ложному утверждению. То есть ложное допущение доказывает желаемое утверждение.



Более того, путь $\hat{P}$ является бесцикловым (поскольку P^* — бесцикловый и не посещает j) и посещает каждую вершину из V (поскольку P^* посещает каждую вершину в $V - \{j\}$). Это противоречит допущению о том, что P — самый дешевый путь.

*21.2. Поиск длинных путей посредством цветового кодирования

В изучении алгоритмов графы присутствуют повсеместно, поскольку они выразительны и легкорешаемы. Во всех книгах этой серии мы видели много алгоритмов для обработки графов (графовый поиск, связные компоненты, кратчайший путь и т. д.) и прикладных областей, моделируемых графами (дорожные сети, интернет, социальные сети и т. д.). В этом разделе приводится еще один пример — крутое применение динамического программирования и рандомизации для обнаружения структуры в биологических сетях.

21.2.1. Актуальность

Бóльшая часть работы клетки осуществляется белками (цепочками аминокислот), часто действующими согласованно. Например, ряд белков может

передавать сигнал, который поступает на клеточную мембрану к белкам, регулирующим транскрипцию ДНК в РНК. Понимание таких сигнальных путей и генетических мутаций необходимо для разработки лекарственных препаратов.

Взаимодействие белков моделируются в виде графа, именуемого сетью *белок-белковых взаимодействий* (PPI, protein-protein interaction), с одной вершиной на белок и одним ребром на пару белков, которые предположительно взаимодействуют. Простейшими сигнальными путями между вершинами являются *линейные* пути. Как быстро можно их найти?

21.2.2. Определение задачи

Задача о поиске линейного пути заданной длины в сети белок-белковых взаимодействий может быть представлена в виде задачи о самом дешевом k-вершинном пути, где k-вершинный путь графа — это бесцикловый путь из $k - 1$ ребер, посещающий k разных вершин.

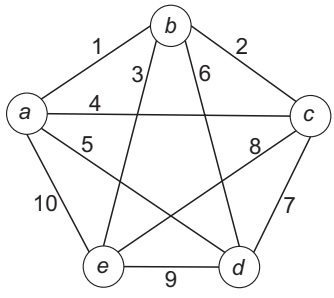
ЗАДАЧА: САМЫЙ ДЕШЕВЫЙ K-ПУТЬ

Вход: неориентированный граф $G = (V, E)$, вещественнозначная стоимость c_e для каждого ребра $e \in E$ и положительное целое число k.

Выход: k-вершинный путь P в G с минимально возможной суммарной стоимостью $\sum_{e \in P} c_e$. (Либо сообщение, что G не имеет k-вершинных путей.)

Реберные стоимости отражают неопределенности, неизбежные в биологических данных, причем их более высокая стоимость указывает на более низкую уверенность в том, что соответствующая пара белков действительно взаимодействует. (Отсутствующие ребра имеют стоимость $+\infty$.) Самый дешевый k-вершинный путь соответствует наиболее вероятному линейному пути заданной длины. В реальных случаях k может равняться 10 или 20, а число вершин может исчисляться сотнями или тысячами.

Например, в графе



минимальная стоимость четырехвершинного пути составляет 8 ($c \rightarrow a \rightarrow \rightarrow b \rightarrow e$).

Задача о самом дешевом k-вершинном пути тесно связана с задачей коммивояжера и является NP-трудной (раздел 22.3). Попробуем улучшить исчерпывающий поиск?

21.2.3. Первая атака на подзадачи

Задача о самом дешевом k-вершинном пути отличается от задачи коммивояжера ограничением длины пути величиной k. Почему бы не использовать подзадачи, полезные для исчерпывающего поиска в задаче коммивояжера (раздел 21.1.5)? То есть взять граф $G = (V, E)$ с вещественнозначными реберными стоимостями и длиной пути k и:

ПОДЗАДАЧИ (ПЕРВАЯ АТАКА)

Вычислить $C_{S,v}$, минимальную стоимость бесциклового пути, который заканчивается в вершине $v \in V$ и посещает вершины исключительно в S (либо $+\infty$, если такого пути не существует).

(Для каждого непустого подмножества $S \subseteq V$ не более чем из k вершин и каждой $v \in S$.)

Поскольку самый дешевый k-вершинный путь может начинаться где угодно, подзадачи не задают стартовую вершину (которая в задаче коммивояжера

всегда была 1). Минимальная стоимость k-вершинного пути — это наименьшее из решений самых больших подзадач (с $|S| = k$). Если граф не имеет k-вершинных путей, то все такие решения подзадач будут равны $+\infty$.

ТЕСТОВОЕ ЗАДАНИЕ 21.2

Предположим, что $k = 10$. Сколько существует подзадач в зависимости от числа вершин n? (Выбрать самый точный ответ.)

а) $O(n)$

б) $O(n^{10})$

в) $O(n^{11})$

г) $O(2^n)$

(Ответ и анализ решения см. в разделе 21.2.11.)

Между тем:

ТЕСТОВОЕ ЗАДАНИЕ 21.3

Предположим, что $k = 10$. Каково время выполнения прямой реализации исчерпывающего поиска в зависимости от n? (Выбрать самый точный ответ.)

а) $O(n^{10})$

б) $O(n^{11})$

в) $O(2^n)$

г) $O(n!)$

(Ответ и анализ решения см. в разделе 21.2.11.)

Ой-ей... алгоритм динамического программирования, использующий подзадачи на с. 146, не может превзойти исчерпывающий поиск! И любой алгоритм с таким временем выполнения, как $O(n^{10})$, является практически бесполезным, за исключением случаев с очень небольшими графами. Нужна еще одна идея.

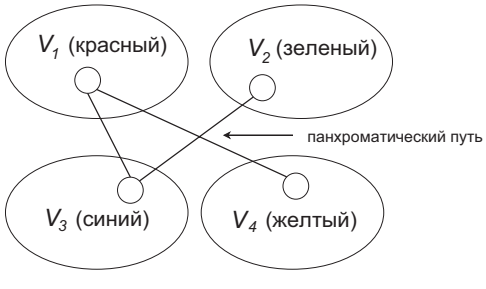
21.2.4. Цветовое кодирование

Зачем нам так много подзадач? Ах да, мы отслеживаем вершины S, посещаемые путем, во избежание непреднамеренного создания пути, который посещает вершину более одного раза (вспомним тестовое задание 21.1 и лемму 21.1). Можно ли обойтись отслеживанием меньшей информации о пути? Вот вдохновляющая идея: взять граф $G = (V, E)$ и границу k длины пути.[1]

ЦВЕТОВОЕ КОДИРОВАНИЕ

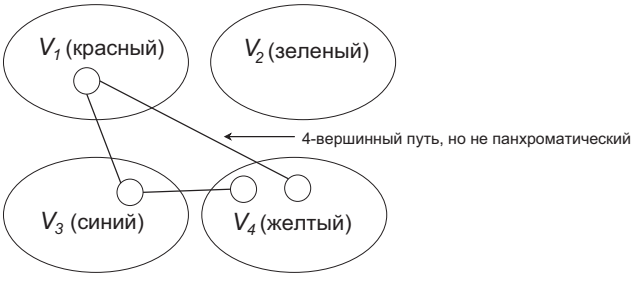
1. Подразделить множество вершин V на k групп V_1, V_2, ..., V_k так, чтобы существовал самый дешевый k-вершинный путь в G, который имеет ровно одну вершину в каждой группе.
2. Среди всех путей с ровно одной вершиной в каждой группе вычислить путь с минимальной стоимостью.

Этот метод называется *цветовым кодированием*, потому что связь каждого целого числа $\{1,2, ..., k\}$ с цветом позволяет визуализировать i-ю группу V_i подраздела на первом шаге как вершины, окрашенные i. На втором шаге ищется самый дешевый панхроматический путь (в котором цвет представлен ровно один раз):



[1] Указанная идея была предложена Ногой Алоном, Рафаэлем Юстером и Ури Цвиком в статье «Цветовое кодирование» (*«Color Coding», Noga Alon, Raphael Yuster, Uri Zwick, Journal of the ACM*, 1995).

Поскольку существует k цветов, панхроматический путь должен быть k-вершинным путем. Обратное является ложным, так как k-вершинный путь может использовать некоторый цвет более одного раза (а какой-то другой цвет не использовать вообще):



Выполнение плана цветового кодирования решит задачу о самом дешевом k-вершинном пути: второй шаг вычислит самый дешевый панхроматический путь, а первый шаг обеспечит, чтобы этот путь также был самым дешевым k-вершинным путем в G (панхроматическим или иным).

Не верите? Но это возможно. Почему вычислить самый дешевый панхроматический путь проще, чем исходную задачу? И как осуществить первый шаг, не зная о том, какими являются самые дешевые k-вершинные пути?

21.2.5. Вычисление самого дешевого панхроматического пути

Сосредоточившись на панхроматических путях, мы упростим задачу о самом дешевом k-вершинном пути, поскольку позволим алгоритму динамического программирования свободно защищаться от повторяющихся *цветов* вместо повторяющихся вершин. (Из повторяющейся вершины следует повторяющийся цвет, но не наоборот.) Подзадачи могут отслеживать цвета в пути вместе с окончательной вершиной, а не сами вершины. Почему это дает выигрыш? Потому что существует только 2^k подмножеств цветов, в отличие от $\Omega(n^k)$ подмножеств, состоящих не более чем из k вершин (см. решение к тестовому заданию 21.2).

Подзадачи и рекуррентное уравнение

Для цветового подмножества $S \subseteq \{1, 2, ..., k\}$ *S-вершинный путь* — это бесцикловый путь с $|S|$ вершинами, в котором представлены все цвета подмножества S. (Панхроматические пути — это именно S-вершинные пути с $S = \{1, 2, ..., k\}$.) Тогда для графа $G = (V, E)$ с реберными стоимостями и цветовым назначением (в $\{1, 2, ..., k\}$) каждой вершине $v \in V$ подзадачи будут таковы:

САМЫЙ ДЕШЕВЫЙ ПАНХРОМАТИЧЕСКИЙ ПУТЬ: ПОДЗАДАЧИ

Вычислить $C_{S,v}$, минимальную стоимость S-вершинного пути, который заканчивается в вершине $v \in V$ (либо $+\infty$, если такого пути не существует).

(Для каждого непустого подмножества $S \subseteq \{1,2, ..., k\}$ цветов и каждой вершины $v \in V$.)

Минимальная стоимость панхроматического пути — это наименьшее из решений самых больших подзадач (где $S = \{1, 2, ..., k\}$). Если граф не имеет панхроматических путей, то все такие решения подзадач будут равны $+\infty$.

Оптимальный путь P для подзадачи с цветовым подмножеством S и финальной вершиной v должен быть построен из оптимального пути для меньшей подзадачи:

Если финальный переход пути P равен (w, v), то его префикс $P' = P - \{(w, v)\}$ должен быть самым дешевым $(S - \{\sigma(v)\})$-вершинным путем, заканчивающимся в w, где $\sigma(v)$ обозначает цвет v.[1] Эта оптимальная подструктура немедленно приводит к следующему рекуррентному уравнению для решения всех подзадач.

Лемма 21.4 (рекуррентное уравнение самого дешевого панхроматического пути). *Продолжая то же обозначение, для каждого подмножества $S \subseteq \{1,2, ..., k\}$ по меньшей мере из двух цветов и вершины $v \in V$:*

$$c_{S,v} = \min_{(w,v) \in E} \left(C_{S-\{\sigma(v)\},w} + c_{wv} \right) \tag{21.7}$$

Алгоритм динамического программирования

В свою очередь, это рекуррентное уравнение немедленно приводит к алгоритму динамического программирования для вычисления минимальной стоимости панхроматического пути (либо $+\infty$, если такого пути не существует):

PANCHROMATICPATH

Вход: неориентированный граф $G = (V, E)$, вещественнозначная стоимость c_{vw} для каждого ребра $(v, w) \in E$ и цвет $\sigma(v) \in \{1, 2, ..., k\}$ для каждой вершины $v \in V$.

Выход: минимальная суммарная стоимость панхроматического пути в графе G (либо $+\infty$, если такой путь не существует).

```
// подзадачи (индексируемые S ⊆ {1, ..., k}, v ∈ V)
A := (2^k − 1) × |V| // двумерный массив
// базовые случаи (|S| = 1)
for i = 1 to k do
    for v ∈ V do
        if σ(v) = i then
```

[1] Формальное доказательство этого утверждения почти совпадает с доказательством леммы 21.1 (см. с. 139).

```
            A[{i}][v] := 0    // посредством пустого пути
        else
            A[{i}][v] := +∞           // такого пути нет
// систематически решить все подзадачи
for s = 2 to k do                     // s = размер подзадачи
    for S with |S| = s do
        for v ∈ V do
            // использовать рекуррентное уравнение
               из леммы 21.4
            A[S][v] := min_{(w,v)∈E} (A[S − {σ(v)}][w] + c_wv)
// лучшее решение самых крупных подзадач
return min_{v∈V} A[{1, 2, ..., k}][v]
```

См. задачу 21.5, где приведен пример алгоритма в действии.

21.2.6. Правильность и время выполнения

Правильность алгоритма панхроматического пути `PanchromaticPath` следует по индукции (на основе размера подзадачи), причем рекуррентное уравнение в лемме 21.4 оправдывает индуктивный шаг. С небольшими дополнительными действиями самый дешевый панхроматический путь может быть восстановлен в $O(k)$-временном шаге постобработки (задача 21.7).

Анализ времени выполнения повторяет анализ алгоритма Беллмана — Форда (глава 18 *Третьей части*). Почти вся работа, выполняемая алгоритмом, происходит в его тройном цикле `for`. Исходя из того что входной граф представлен списками смежности, самая внутренняя итерация цикла, вычисляющая значение $A[S][v]$, занимает время $O(deg(v))$, где $deg(v)$ обозначает степень (число инцидентных ребер) вершины v.[1] Для каждого подмножества S объ-

[1] Технически этот анализ исходит из того, что каждая вершина имеет степень по меньшей мере 1. Вершины степени 0 можно отбросить на шаге предварительной обработки.

единенное время, затраченное на решение ассоциированных $|V|$ подзадач, равно $O\left(\sum_{v \in V} \deg(v)\right) = O(m)$, где $m = |E|$ обозначает число ребер.[1] Число разных цветовых подмножеств S меньше 2^k, поэтому совокупное время работы алгоритма равно $O(2^k m)$.

Теорема 21.5 (свойства алгоритма `PanchromaticPath`). *Для каждого графа G с m ребрами, вещественнозначными реберными стоимостями и отнесением каждой вершины цвету в {1, 2, ..., k} алгоритм* `PanchromaticPath` *выполняется за время $O(2^k m)$ и возвращает минимальную стоимость панхроматического пути (если он существует) или $+\infty$ (в противном случае).*

При масштабировании времени выполнения вместе с 2^k, а не n^k, алгоритм `PanchromaticPath` выигрывает у исчерпывающего поиска. Однако на самом деле нас интересует задача о самом дешевом k-вершинном пути без каких-либо панхроматических ограничений. Как этот алгоритм нам поможет?

21.2.7. Рандомизация спешит на помощь

Первый шаг метода цветового кодирования окрашивает вершины входного графа так, чтобы хотя бы один самый дешевый k-вершинный путь становился панхроматическим. Как этого достичь, не зная, какие из k-вершинных путей самые дешевые? Самое время ввести еще один алгоритмический инструмент — *рандомизацию*. Она должна равномерно распределить случайную раскраску со здоровой вероятностью передать некоторый самый дешевый панхроматический k-вершинный путь, чтобы алгоритм `PanchromaticPath` его нашел.

ТЕСТОВОЕ ЗАДАНИЕ 21.4

Предположим, что каждой вершине графа G назначен цвет из $\{1, 2, ..., k\}$, независимо и равномерно, случайным образом. Рас-

[1] Сумма $\sum_{v \in V} \deg(v)$ степеней вершин в два раза больше числа ребер, причем каждое ребро вносит 1 в степень каждой из своих двух конечных точек.

смотрим k-вершинный путь P в G. Какова вероятность того, что P окажется панхроматическим?

а) $1/k$

б) $1/k^2$

в) $1/k!$

г) $k!/k^k$

д) $1/k^k$

(Ответ и анализ решения см. в разделе 21.2.11.)

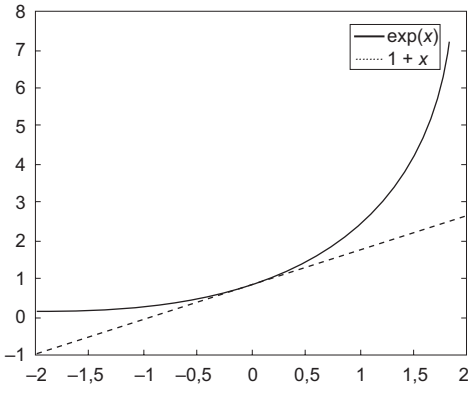
Является ли эта вероятность удачи — назовем ее p — большой или малой? Напомним, что мы знаем чрезвычайно точную оценку факториальной функции (аппроксимацию Стирлинга на с. 135). Подключим аппроксимации из (21.1), где k играет роль n:

$$p = \frac{k!}{k^k} \approx \frac{1}{k^k} \times \sqrt{2\pi k}\left(\frac{k}{e}\right)^k = \frac{\sqrt{2\pi k}}{e^k}. \tag{21.8}$$

Выглядит плохо — вероятность превращения самого дешевого k-вершинного пути в панхроматический составляет менее 1 % уже при $k = 7$. Но если мы поэкспериментируем с большим числом независимых случайных раскрасок, выполняя алгоритм `PanchromaticPath` для каждой и запоминая самый дешевый найденный k-вершинный путь, то нам потребуется лишь одна удачная раскраска. Сколько случайных испытаний T потребуется, чтобы обеспечить вероятность нахождения такой раскраски 99 %?

Испытание достигает успеха с вероятностью p и оказывается безуспешным с вероятностью $1 - p$. Поскольку испытания проводятся независимо, вероятности неуспеха умножаются. Тогда вероятность того, что все T испытаний будут безуспешными, равна $(1 - p)^T$.[1] Давайте ограничим $1 - p$ значением e^{-p}:

[1] Справочные сведения о дискретной вероятности см. в приложении к *Первой части* или в ресурсах по адресу www.algorithmsilluminated.org.



Эта вероятность неуспеха равна

$$(1-p)^T \leq (e-p)^T = e^{-pT}. \tag{21.9}$$

Установка правой части (21.9) равной цели δ (например, 0,01), взятие логарифмов обеих сторон и нахождение решения для T показывает, что

$$T \geq \frac{1}{p} \times \ln\left(\frac{1}{\delta}\right) \tag{21.10}$$

независимых испытаний достаточно, для того чтобы довести вероятность неуспеха до δ. Как и ожидалось, чем ниже вероятность успеха единичного испытания или желаемая вероятность неуспеха, тем больше требуется испытаний. Подстановка вероятности успеха p из (21.8) в (21.10) показывает, что:

Лемма 21.6 (случайные раскраски достаточно хороши). *Для каждого графа G, k-вершинного пути P в графе G и вероятности неуспеха $\delta \in (0, 1)$: если*

$$T \geq \frac{e^k}{\sqrt{2\pi k}} \times \ln\left(\frac{1}{\delta}\right),$$

то вероятность того, что по меньшей мере одна из T равномерно распределенных случайных раскрасок сделает P панхроматическим, равна по меньшей мере 1 − δ.

Экспоненциальное число попыток в лемме 21.6 может показаться экстравагантным, но оно находится в том же диапазоне, что и время, уже необходимое для одного вызова подпрограммы `PanchromaticPath`. Когда k является малым по отношению к n — режиму, относящемуся к актуальному применению (раздел 21.2.1), — число испытаний потребует намного меньше времени, чем решение задачи о самом дешевом k-вершинном пути на n-вершинных графах посредством исчерпывающего поиска (которое, согласно тестовому заданию 21.3, масштабируется вместе с n^k).

21.2.8. Окончательный алгоритм

Мы привели дела в порядок: лемма 21.6 обещает, что многочисленных независимых случайных раскрасок будет достаточно для того, чтобы реализовать первый шаг метода цветового кодирования (с. 148), а алгоритм `PanchromaticPath` позаботится о втором шаге.

COLORCODING

Вход: неориентированный граф $G = (V, E)$, вещественнозначная стоимость c_{vw} для каждого ребра $(v, w) \in E$, длина k пути и вероятность неуспеха $\delta \in (0, 1)$.

Выход: минимальная суммарная стоимость k-вершинного пути в графе G (либо $+\infty$, если такого пути не существует), за исключением случаев, когда вероятность неуспеха не превышает δ.

$C_{\text{best}} := +\infty$ `// самый дешевый k-вершинный путь, найденный до этого`

`// число случайных попыток (из леммы 21.6)`

$T := \left(e^k / \sqrt{2\pi k}\right) \ln \frac{1}{\delta}$ `// округлить до целого числа`

for $t = 1$ to T **do** `// независимые испытания`

  for each $v \in V$ **do** `// выбрать случайную раскраску`

    $\sigma_t(v)$:= случайное число из $\{1, 2, ..., k\}$

  `// лучший панхроматический путь для этой раскраски`

  C := `PanchromaticPath`(G, c, σ_t) `// см. с. 151`

if $C < C_{\text{best}}$ **then** `// найден более дешевый k-вершинный путь!`

$C_{\text{best}} := C$

return C_{best}

21.2.9. Время выполнения и правильность

Время работы алгоритма цветового кодирования определяется его $T = O\left(\left(e^k / \sqrt{k}\right) \ln \frac{1}{\delta}\right)$ вызовами $O(2^k m)$-временной подпрограммы `PanchromaticPath`, где m обозначает число ребер (теорема 21.5).

Чтобы доказать правильность, рассмотрим самый дешевый k-вершинный путь P в графе G с суммарной стоимостью C^*.[1] Для каждой раскраски δ минимальная стоимость панхроматического пути — результат, получаемый на выходе из подпрограммы `PanchromaticPath`, — это по меньшей мере C^*. Всякий раз, когда δ становится панхроматическим, эта стоимость составляет ровно C^*. В силу леммы 21.6, с вероятностью по меньшей мере $1 - \delta$ хотя бы одна из итераций внешнего цикла выбирает такую раскраску. В этом случае алгоритм `PanchromaticPath` возвращает C^*, которая является правильным ответом.[2]

Итог:

Теорема 21.7 (свойства алгоритма `ColorCoding`). *Для каждого графа G с n вершинами и m ребрами, вещественнозначными реберными стоимостями, длиной пути $k \in \{1, 2, ..., n\}$ и вероятностью неуспеха $\delta \in (0, 1)$ алгоритм цветового кодирования* `ColorCoding` *выполняется за время*

$$O\left(\frac{(2e)^k}{\sqrt{k}} m \ln\left(\frac{1}{\delta}\right)\right) \tag{21.11}$$

---

[1] Если граф G не имеет k-вершинных путей, то каждый вызов подпрограммы `PanchromaticPath` и алгоритма `ColorCoding` возвращает $+\infty$ (верный ответ).

[2] Рандомизированный алгоритм `QuickSort` имеет случайное время выполнения (от почти линейного до квадратичного), но всегда является правильным (глава 5 *Первой части*). В алгоритме `ColorCoding`, наоборот, исходы подбрасываний монет определяют его правильность, но мало влияют на время его выполнения.

и с вероятностью по меньшей мере 1 − δ возвращает минимальную стоимость k-вершинного пути в графе G (если он существует) или +∞ (в противном случае).

Как оценить время выполнения алгоритма `ColorCoding`? Плохая новость: граница времени выполнения в (21.11) является экспоненциальной — это неудивительно, ведь задача о самом дешевом k-вершинном пути NP-трудная. Хорошая новость: его экспоненциальная зависимость полностью ограничена длиной пути k, только с линейной зависимостью от размера графа. Фактически, в частном случае, при котором $k \leq c \ln m$ для константы $c > 0$, алгоритм `ColorCoding` решает задачу о самом дешевом k-вершинном пути за полиномиальное время![1, 2]

21.2.10. Пересмотр сетей белок-белковых взаимодействий

Гарантии вроде теоремы 21.7 хороши, но насколько хорошо алгоритм `ColorCoding` работает для поиска длинных линейных путей в сетях белок-белковых взаимодействий (PPI)? Он отлично подходит для применения, в котором типичные длины пути k находятся в диапазоне 10–20. (Значительно

---

[1] Обратите внимание, что $(2e)^{c \ln m} = m^{c \ln(2e)} \approx m^{1,693c}$, что является полиномиальным по размеру графа.

[2] Алгоритм `ColorCoding` является примером фиксированно-параметрического алгоритма, то есть алгоритма со временем выполнения $O(f(k) \times n^d)$, где n обозначает размер входных данных, d — константу (независимую от k и n) и k — параметр, измеряющий «трудность» экземпляра. Функция f должна быть независимой от n, но может иметь произвольную зависимость (обычно экспоненциальную или хуже) от параметра k. Фиксированно-параметрический алгоритм выполняется за полиномиальное время для всех случаев, когда k достаточно мал относительно n.

XXI век стал свидетелем грандиозного прогресса в понимании того, какие NP-трудные задачи и выбор параметров позволяют использовать фиксированно-параметрические алгоритмы. Для глубокого погружения в эту тему ознакомьтесь с книгой «Параметризованные алгоритмы» Марека Цигана, Федора В. Фомина, Лукаша Ковалика, Даниэля Локштанова, Даниэля Маркса, Марцина Пилипчука, Михаила Пилипчука и Сакета Саурабха («*Parameterized Algorithms», Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh, Springer,* 2015).

более длинные пути, если они существуют, трудно интерпретировать.) Уже на компьютерах 2007 года реализации алгоритма `ColorCoding` могли находить линейные пути длиной $k = 20$ в главенствующих сетях белок-белковых взаимодействий с тысячами вершин. Это было намного лучше исчерпывающего поиска (бесполезного даже для $k = 5$) и алгоритмов того времени (которым едва удавалось выходить за пределы $k = 10$).[1]

21.2.11. Решения к тестовым заданиям 21.2–21.4

Решение к тестовому заданию 21.2

Правильный ответ: (б). Число непустых подмножеств $S \subseteq V$ размера не более 10 равно сумме $\sum_{i=1}^{10} \binom{n}{i}$ из десяти биномиальных коэффициентов. Ограничение i-го слагаемого сверху по n^i и использование формулы (20.9) для геометрического ряда показывают, что эта сумма равна $O(n^{10})$. Разложение последнего биномиального коэффициента показывает, что он один равен $\Omega(n^{10})$.[2] При наличии не более десяти вариантов выбора конечной точки v для каждого множества S суммарное число подзадач равно $\Theta(n^{10})$.

Решение к тестовому заданию 21.3

Правильный ответ: (а). Исчерпывающий поиск перечисляет $n \times (n - 1) \times \times (n - 2) \times ... \times (n - 9) = \Theta(n^{10})$ упорядоченных десятичленных кортежей из разных вершин, вычисляет стоимость каждого кортежа, который соответствует пути (за время $O(1)$, исходя из доступа к матрице смежности, заполненной реберными стоимостями), и запоминает лучший из десяти-

---

[1] Более подробную информацию см. в работе Фалька Хюффнера, Себастьяна Верникe и Томаса Цихнера «Инженерия алгоритмов цветового кодирования с применениями к обнаружению сигнальных путей» («*Algorithm Engineering for ColorCoding with Applications to Signaling Pathway Detection», Falk Hüffner, Sebastian Wernicke, and Thomas Zichner, Algorithmica,* 2008).

[2] Вспомните, что обозначение «Омега большое» аналогично фразе «больше или равно». Формально $f(n) = \Omega(g(n))$, если и только если существует постоянная $c > 0$, такая, что $f(n) \geq c \times g(n)$ для всех достаточно больших n. Кроме того, $f(n) = \Theta(g(n))$, если и только если $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$.

вершинных путей, с которыми он сталкивается. Время выполнения этого алгоритма равно (n^{10}).

Решение к тестовому заданию 21.4

Правильный ответ: (г). Существует k^k разных способов раскрасить вершины P (k вариантами цвета для каждой из k вершин), вероятность каждого из которых равна $1/k^k$. Сколько из них делают P панхроматическим? Существует k вариантов выбора, когда вершина получает цвет 1, затем $k - 1$ оставшихся вариантов, когда вершина получает цвет 2, и так далее, в общей сложности $k!$ панхроматических раскрасок. Следовательно, вероятность панхроматичности равна $k!/k^k$.

21.3. Алгоритмы для конкретных задач против волшебных ящиков

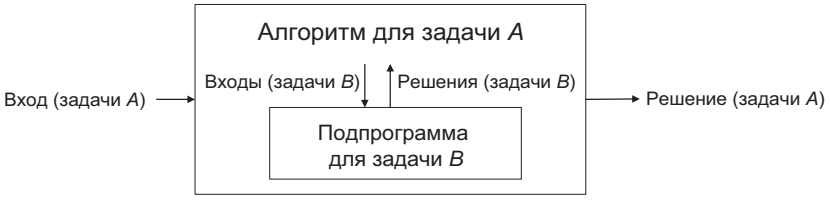
21.3.1. Редукции и волшебные ящики

Заказные решения фундаментальных задач, такие как алгоритмы `BellmanHeldKarp` (раздел 21.1.6) и `ColorCoding` (раздел 21.2.8), хороши. Но прежде чем вкладывать усилия в разработку или написание кода нового алгоритма, спросите себя:

> Является ли эта задача частным случаем или тонко замаскированной версией той задачи, решение которой мне известно?

Если ответ «нет» или даже «да, но алгоритмы для более общей задачи недостаточно хороши для этого применения», то переход к разработке задачно-специфичного алгоритма оправдан.

На протяжении всей этой серии книг мы видели несколько задач, для которых ответ был «да». Например, задача об отыскании медианы сводится к сортировке, задача о кратчайшем пути для всех пар вершин сводится к задаче о кратчайшем пути с одним истоком, а задача нахождения наибольшей общей подпоследовательности является частным случаем задачи о выравнивании рядов (раздел 19.5.2). Такие редукции переносят легкорешаемость из задачи B в задачу A:



Наши редукции до этого сводились к задачам *B*, для которых мы сами уже разработали быстрый алгоритм. Но сведение задачи *A* к задаче *B* сохраняет свою силу, *даже если вы лично не знаете*, как решить задачу *B*. Если кто-то даст вам волшебный ящик (например, непостижимую часть программного обеспечения), которая решает задачу *B*, то вы будете готовы пойти дальше: чтобы решить задачу *A*, исполнить редукцию к задаче *B* и далее вызывать волшебный ящик по мере необходимости.

21.3.2. Решатели задач MIP и SAT

«Волшебный ящик», наверное, звучит как чистая фантазия вроде единорога или фонтана вечной молодости. Существует ли он на самом деле? В разделах 21.4 и 21.5 описаны две наиболее близкие аппроксимации — решатели для задач MIP и SAT. Под решателем мы подразумеваем сложный алгоритм, тщательно настроенный и умело реализованный в виде готовой к использованию программы. Задачи MIP и SAT представляют собой очень общие задачи, достаточно выразительные, чтобы назвать большинство задач, изучаемых в этой серии книг, их частными случаями.

Несколько десятилетий инженерных усилий и изобретательности были вложены в современные решатели задач MIP и SAT. По этой причине, несмотря на свою общность, такие решатели полунадежно решают среднеразмерные экземпляры NP-трудных задач за приемлемое время. Производительность решателей сильно варьируется в зависимости от задачи (и многих других факторов), но вы можете скрестить пальцы и надеяться, что с их помощью входные данные размером в тысячи элементов будут обработаны менее чем за день или намного быстрее. В некоторых приложениях решатели задач MIP и SAT необоснованно эффективны даже для больших экземпляров с миллионами элементов входных данных.

21.3.3. Чему вы научитесь и чему не научитесь

Цели разделов 21.4 и 21.5 весьма скромны. Они не объясняют то, как работают решатели задач MIP и SAT — это материал на еще одну книгу. Вместо этого они готовят вас к взаимодействию с указанными решателями.[1]

ЦЕЛИ РАЗДЕЛОВ 21.4–21.5

1. Сообщить о существовании полунадежных волшебных ящиков под названием решатели задач MIP и SAT для обуздания NP-трудных задач на практике. (Мало кто из программистов об этом знает!)
2. Показать примеры кодирования NP-трудных задач, похожих на MIP и SAT.
3. Рассказать, куда идти дальше, чтобы узнать больше.

21.3.4. Ошибки новичка повторно

Решатели задач MIP и SAT обычно решают сложные задачи, но не обманывайтесь, думая, что NP-трудность не имеет никакого значения на практике (третья ошибка новичка из раздела 19.6). Применяя такой решатель к NP-трудной задаче, держите пальцы скрещенными и имейте наготове план Б (например, быстрый эвристический алгоритм) на случай, если решатель окажется неуспешным. И не ошибитесь: обязательно будут примеры, в том числе довольно малые, которые смогут поставить ваш решатель на колени. С NP-трудными задачами берешь все, что сможешь унести, и полунадежные волшебные ящики хороши настолько, насколько это возможно.

---

[1] С высоты птичьего полета основная идея решателя такова: рекурсивно проводить поиск в пространстве возможных решений типа поиска сначала в глубину, применяя полученные к этому моменту подсказки к агрессивно урезанным, еще не исследованным кандидатам (например, значение целевой функции которых не лучше уже найденного), отступая назад по мере необходимости в надежде, что бóльшая часть поискового пространства не будет явно рассмотрена. Разведать эти идеи дальше можно, обратившись к «ветвям и границам» (для решателей задач MIP) и «управляемому конфликтом усвоению логических выражений» (для SAT).

21.4. Решатели задач MIP

Большинство задач дискретной оптимизации можно рассматривать как задачи смешанного целочисленного программирования (mixed integerprogramming, MIP).[1] Всякий раз, когда вы сталкиваетесь с NP-трудной задачей оптимизации, которую можно закодировать как задачу MIP, попробуйте натравить на нее самый лучший решатель задач MIP.

21.4.1. Пример: задача о рюкзаке

В задаче о рюкзаке (раздел 19.4.2) входные данные задаются $2n + 1$ целыми положительными числами: n значениями предметов $v_1, v_2, ..., v_n$, n размерами элементов $s_1, s_2, ..., s_n$ и вместимостью рюкзака C. Например:

	Значение	Размер
Предмет 1	6	5
Предмет 2	5	4
Предмет 3	4	3
Предмет 4	3	2
Предмет 5	2	1
Вместимость рюкзака: 10		

Требуется вычислить подмножество предметов с максимально возможной суммарной стоимостью при условии наличия суммарного размера, не превышающего вместимость рюкзака. В спецификации задачи прописаны три ориентира:

1. *Решения, которые необходимо принять.* Для каждого из n предметов решить, нужно ли его включать в подмножество или нет. Одним из удобных способов численного кодирования этих двоичных решений по каждому предмету является использование переменных 0–1, именуемых переменными решения:

$$x_j = \begin{cases} 1 \text{ если предмет } j \text{ включен} \\ 0 \text{ если предмет } j \text{ исключен} \end{cases} \tag{21.12}$$

[1] Это такое же анахроничное употребление слова «программирование», как и в термине «динамическое программирование» (или «телевизионное программирование»); оно относится к планированию, а не к написанию кода.

2. *Ограничения, которые необходимо соблюдать.* Сумма размеров выбранных предметов должна не превышать вместимость C рюкзака. Это ограничение легко выражается в терминах переменных решения: предмет j вносит s_j в суммарный размер, если он включен (с $x_j = 1$), и 0, если он исключен (с $x_j = 0$):

$$\underbrace{\sum_{j=1}^{n} s_j x_j}_{\substack{\text{суммарный размер} \\ \text{выбранного подмножества}}} \leq C \tag{21.13}$$

3. *Целевая функция, которую нужно достичь.* Сумма значений выбранных предметов должна быть как можно большей (при условии ограничения по вместимости). Эта целевая функция одинаково легко выражается (где j вкладывает значение v_j, если предмет включен, и 0, если он исключен):

$$\text{максимизировать} \quad \underbrace{\sum_{j=1}^{n} v_j x_j}_{\substack{\text{суммарное значение} \\ \text{выбранного подмножества}}} \tag{21.14}$$

Никогда не угадаете! В (21.12)–(21.14) вы только что увидели свой первый пример *целочисленной программы*. Например, в описанном выше 5-предметном экземпляре эта целочисленная программа читается:

$$\text{максимизировать} \quad 6x_1 + 5x_2 + 4x_3 + 3x_4 + 2x_5 \tag{21.15}$$

$$\text{при условии, что} \quad 5x_1 + 4x_2 + 3x_3 + 2x_4 + x_5 \leq 10; \tag{21.16}$$

$$x_1, x_2, x_3, x_4, x_5 \in \{0, 1\}. \tag{21.17}$$

Это именно то описание, которое может быть подано непосредственно в волшебный ящик, именуемый *решателем задач MIP.*[1] Например, чтобы решить

[1] Почему именно «смешанное»? Потому что эти решатели также размещают в себе переменные решения, которые могут принимать вещественные (не обязательно целочисленные) значения. Некоторые авторы ссылаются на задачи MIP как на целочисленные линейные программы (ILP, integer linear program) или просто целочисленные программы (IP). Другие оставляют последний термин для задач MIP, в которых все переменные решения имеют целочисленные значения.

Задача целочисленного программирования, в которой ни одна из переменных решения не должна быть целым числом, называется *линейной программой* (LP).

целочисленную программу в (21.15)–(21.17) с помощью оптимизатора Гуроби — ведущего коммерческого решателя задач MIP, — просто вызовите ее из командной строки со следующим входным файлом:

```
Maximize 6 x(1) + 5 x(2) + 4 x(3) + 3 x(4) + 2 x(5)
subject to
5 x(1) + 4 x(2) + 3 x(3) + 2 x(4) + x(5) <= 10
binary
x(1) x(2) x(3) x(4) x(5)
end
```

Как по волшебству, решатель выдаст оптимальное решение (в данном случае $x_1 = 0, x_2 = x_3 = x_4 = x_5 = 1$ и значение целевой функции 14).[1]

21.4.2. Задачи MIP в более общем случае

В общем случае задачи MIP определяются тремя ориентирами из раздела 21.4.1: переменными решения, наряду со значениями, которые они могут принимать (такими как 0, или 1, или любое целочисленное, или любое вещественное число), ограничениями и целевой функцией. Но (!) и ограничения, и целевая функция должны быть *линейными* в переменных решения.[2] Другими словами, можно масштабировать переменную решения на константу и складывать переменные решения между собой, и все. Например, в (21.15)–(21.17) вы не увидите членов, таких как x_j^2, $x_j x_k$, $1/x_j$, e^{x_j} и так далее.[3]

Современные решатели работают особенно хорошо для линейных программ и часто решают тысячи из них в ходе решения одной задачи MIP. (Соответственно, линейное программирование — это задача, поддающаяся решению за полиномиальное время, тогда как общая задача MIP — NP-трудная.)

[1] Входной файл для этого игрового примера достаточно легко создается вручную. Для более крупных экземпляров потребуется написать программу, которая автоматически генерирует входной файл или взаимодействует непосредственно с API решателя.

[2] Более точным будет термин «смешанная целочисленная линейная программа» (MILP, mixed integer linear program), но менее приятным для слуха...

[3] Современные решатели могут также учитывать лимитированные типы нелинейности (например, квадратичные члены), но обычно гораздо быстрее работают с линейными ограничениями и целевыми функциями.

ЗАДАЧА: MIP (СМЕШАННОЕ ЦЕЛОЧИСЛЕННОЕ ПРОГРАММИРОВАНИЕ)

Вход: список двоичных, целочисленных или вещественных переменных решения x_1, x_2, ..., x_n; линейная целевая функция, подлежащая максимизации или минимизации, заданная ее коэффициентами c_1, c_2, ..., c_n; и m линейных ограничений, причем каждое ограничение i задается его коэффициентами a_{i1}, a_{i2}, ..., a_{in} и правосторонним b_i.

Выход: присвоение значений переменным x_1, x_2, ..., x_n, которое оптимизирует целевую функцию $\left(\sum_{j=1}^{n} c_j x_j\right)$ с учетом m ограничений $\left(\sum_{j=1}^{n} a_{ij} x_j \le b_i \text{ для всех } i=1, 2, ..., m\right)$. (Либо, если ни одно присвоение не удовлетворяет всем ограничениям, то сообщить об этом факте.)

Даже с линейным ограничением NP-трудные задачи оптимизации неприлично просто выражаются в виде задач MIP. Например, в *двумерной* задаче о рюкзаке каждый предмет j имеет *вес* w_j в дополнение к значению v_j и размеру s_j. В дополнение к вместимости C рюкзака существует весовая граница W. Требуется выбрать максимальное значение подмножества предметов с суммарным размером, не превышающим C, и суммарным весом не больше W. Будучи выпускником курса молодого бойца по алгоритмам динамического программирования, вы без проблем разработаете для этой задачи алгоритм. Но быстрее будет добавить ограничение

$$\sum_{j=1}^{n} w_j x_j \le W \tag{21.18}$$

в задачу MIP (21.12)–(21.14)!

Уже знакомые задачи оптимизации — задачи о независимом множестве с максимальным весом (раздел 19.4.2), о минимальной производственной продолжительности (раздел 20.1) и о максимальном охвате (раздел 20.2) — почти одинаково легко выразить в виде задач MIP (см. задачу 21.9).[1] Задачи

---

[1] Для начала ограничение формы $\sum_{j=1}^{n} a_{ij} x_j \ge b_i$ может быть представлено эквивалентным ограничением $\sum_{j=1}^{n} (-a_{ij}) x_j \le -b_i$, а ограничение в форме равенства $\sum_{j=1}^{n} a_{ij} x_j = b_i$ может быть представлено парой ограничений в форме неравенства.

MIP также являются базисом для точных алгоритмов для задачи коммивояжера (раздел 19.1.2), хотя это применение очень изощренное (см. задачу 21.10).[1]

Задача обычно может быть сформулирована как задача MIP несколькими способами, причем одни формулировки приводят к более высокой производительности решателя, чем другие (в некоторых случаях — на порядки). Если ваша первая попытка решить задачу оптимизации с помощью решателя задач MIP не удалась, попробуйте поэкспериментировать с альтернативными кодировками. Как и в случае с алгоритмами, разработка хороших формулировок задачи MIP требует практики; ресурсы в сноске на этой странице помогут вам начать работу.

Наконец, если на завершение работы у вашего решателя задачи MIP уходит слишком много времени, независимо от того, какую формулировку вы испытываете, то вы можете прервать его через заданное количество времени и использовать наилучшее найденное решение. (Решатели задач MIP обычно генерируют ряд постепенно более качественных допустимых решений, аналогично алгоритмам локального поиска разделов 20.4–20.5.) И так у вас получится быстрый эвристический алгоритм.

21.4.3. Решатели задач MIP: некоторые отправные точки

Теперь, когда вы готовы применить решатель задач MIP к своей любимой задаче, с чего начать? На момент написания этой книги (в 2020 году) между коммерческими и некоммерческими решателями задач MIP существует огромная пропасть в производительности. Оптимизатор Гуроби рассматривается как самый быстрый и надежный решатель задач MIP, а его конкурентами являются CPLEX и FICO Xpress. Студенты и сотрудники университета могут получить

[1] Больше примеров и трюков вы найдете в бесплатной документации для решателей, перечисленных в разделе 21.4.3, или учебнике «Построение моделей в математическом программировании» Х. Пола Уильямса («*Model Building in Mathematical Programming», H. Paul Williams, Wiley, 5th edition,* 2013). Примеры из книги Дэна Гусфилда «Целочисленное линейное программирование и биология систем» («*Integer Linear Programming in Computational and Systems Biology», Dan Gusfield, Cambridge*, 2019) склоняются в сторону биологических применений, но в целом полезны для новичков в MIP (и в особенности пользователям оптимизатора Гуроби).

для них бесплатные академические лицензии (только для исследовательских и образовательных целей).

Если вы застряли, используя некоммерческий решатель, хорошими отправными точками будут решатели SCIP, CBC, MIPCL и GLPK. Решатели CBC и MIPCL имеют более либеральные лицензионные соглашения, чем два других, которые являются бесплатными только для некоммерческого использования.

Вы можете отделить формулирование смешанной целочисленной программы от описания вашей задачи конкретному решателю, указав свою задачу на высокоуровневом решателе — независимом языке моделирования, таком как CVXPY на основе языка Python. Затем вы сможете поэкспериментировать со всеми поддерживаемыми этим языком решателями, причем ваша высокоуровневая спецификация автоматически скомпилируется в ожидаемый решателем формат.

21.5. Решатели задач SAT

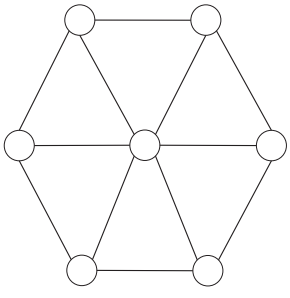
Многие приложения в первую очередь выясняют, существует ли допустимое решение (и если да, то какое), а не оптимизируют числовую целевую функцию. Задачи такого типа часто можно рассматривать как задачи о выполнимости (SAT). Всякий раз, когда вы сталкиваетесь с NP-трудной задачей, которую можно закодировать как задачу SAT, попробуйте натравить на нее самый лучший решатель задач SAT.

21.5.1. Пример: раскраска графа

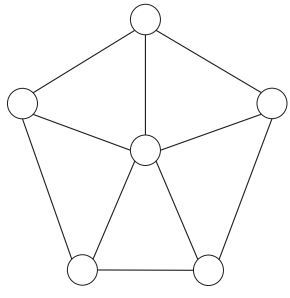
Одна из старейших графовых задач, широко изученная еще в XIX веке, — это задача о *раскраске графов*. *k-цветная* раскраска неориентированного графа $G = (V, E)$ — это закрепление $\sigma(v)$ за каждой его вершиной $v \in V$ цвета в $\{1, 2, ..., k\}$ так, что ни одно ребро не является монохроматическим (то есть $\sigma(v) \neq \sigma(w)$ всякий раз, когда $(v, w) \in E$).[1] Граф с k-цветной раскраской на-

[1] Алгоритм `ColorCoding` (раздел 21.2.8) использует случайные раскраски, которые обычно не являются k-цветными раскрасками, внутри как средство достижения более быстрого времени выполнения. В этом разделе исследуемая задача явным образом касается k-цветных раскрасок.

зывается — внимание — *пригодным для k-цветной раскраски*.[1] Например, колесный граф с шестью спицами пригоден для трехцветной раскраски, тогда как колесный граф с пятью спицами — нет (убедитесь сами):



пригоден для
трехцветной раскраски



не пригоден для
трехцветной раскраски

ЗАДАЧА: РАСКРАСКА ГРАФА

Вход: неориентированный граф $G = (V, E)$ и положительное целое число k.

Выход: k-цветная раскраска графа G либо правильное объявление того, что G не пригоден для k-цветной раскраски.

Задача о раскраске графа — не только баловство. Например, задача о назначении учебных занятий одной из k учебных аудиторий является как раз задачей о раскраске графа (с одной вершиной на учебное занятие и ребром между каждой парой учебных занятий, которые накладываются друг на друга во времени). Применение задач раскраски графов см. в главе 24.

[1] Самым известным результатом во всей теории графов является теорема о четырех цветах, утверждающая, что каждый плоский граф (нарисованный на листе бумаги, без пересечений ребер) пригоден для четырехцветной раскраски. (Второй граф в этом разделе показывает, что могут потребоваться четыре цвета.) Также оказалось, что географические карты нуждаются только в четырех цветах чернил, чтобы окрасить разными цветами каждую пару соседних стран.

21.5.2. Выполнимость булевых формул

Нечисловая и основанная на правилах природа задачи о раскраске графов предполагает, что переменные решения и ограничения должны выражаться с помощью формализма *логики*, а не арифметики. Вместо числовых переменных решения мы будем использовать *булевы* переменные (принимающие значения только «истина» и «ложь». Операция *присвоения истинности*[1] приписывает каждой переменной одно из этих двух значений. Ограничения, которые также называются элементарными дизъюнкциями, или дизъюнктивными одночленами (*клаузами*), — это логические формулы, выражающие ограничения на допустимые присвоения истинности. Кажущийся простым тип ограничения, именуемый *дизъюнкцией литералов*, использует только логические операции «или» (обозначаемые знаком $\vee$) и «не» (обозначаемые знаком $\neg$).[2] Например, ограничение $x_1 \vee \neg x_2 \vee x_3$ является дизъюнкцией литералов и удовлетворяется, если вы не испортите все три его запроса на присвоение (установив x_1 и x_3 равными значению «ложь» и x_2 равным значению «истина»):

Значение x_1	**Значение x_2**	**Значение x_3**	**$x_1 \vee \neg x_2 \vee x_3$ выполнимо?**
истина	истина	истина	да
истина	истина	ложь	да
истина	ложь	истина	да
ложь	истина	истина	да
истина	ложь	ложь	да
ложь	истина	ложь	нет
ложь	ложь	истина	да
ложь	ложь	ложь	да

В общем случае дизъюнкции литералов — это добродушные существа: одно такое с k литералами, каждый из которых соответствует отдельной переменной решения, запрещает один и только один из 2^k способов присвоения значений своим переменным.

---

[1] Присвоение истинности (truth assignment) — это предметно-истинностная функция, которая увязывает логические переменные пропозициональной формулы со значениями «истина» или «ложь» или, иными словами, отображает указанные переменные в эти значения. — *Примеч. пер.*

[2] Термин «литерал» означает переменную решения x_i или ее отрицание: $\neg x_i$, а термин «дизъюнкция» обозначает операцию логического «или».

Экземпляр задачи SAT определяется его переменными (ограниченными быть булевыми) и ограничениями (ограниченными быть дизъюнкциями литералов).

ЗАДАЧА: ВЫПОЛНИМОСТЬ

Вход: список булевых переменных решения $x_1, x_2, ..., x_n$ и список ограничений, каждое из которых является дизъюнкцией одного или нескольких литералов.

Выход: присвоение истинности переменным $x_1, x_2, ..., x_n$, удовлетворяющее всем ограничениям, либо правильное объявление о том, что такого присвоения истинности не существует.

21.5.3. Кодирование раскраски графа как задачи SAT

Является ли задача SAT с ее сугубо булевыми переменными и дизъюнкциями литералов достаточно выразительной, чтобы кодировать другие задачи? Например, в задаче о раскраске графа мы хотели бы иметь одну (не булеву) переменную решения для вершин, каждая из которых принимает одно из k разных значений (одно в расчете на возможный цвет).

Немного попрактиковавшись, вы сможете закодировать удивительно большое число задач как задачу SAT.[1, 2] Например, чтобы закодировать экземпляр за-

[1] Многие примеры, включая классические приложения для верификации аппаратного и программного обеспечения, можно найти в «Руководстве по выполнимости» под редакцией Армина Бире, Марин Хеуле, Ханса ван Маарена и Тоби Уолша («*Handbook of Satisfiability», Armin Biere, Marijn Heule, Hans van Maaren, Toby Walsh, IOS Press*, 2009). Или, если вам интересно, чем занимался Дональд Э. Кнут в последнее время, то ознакомьтесь с разделом 6 тома 4 «Искусство компьютерного программирования» («*The Art of Computer Programming», Donald E. Knuth, Addison-Wesley*, 2015). Еще один забавный факт: недавно решатели задач выполнимости были использованы для взлома некогда безопасной криптографической хэш-функции SHA-1 — см. «Первая коллизия для полного SHA-1» Марка Стивенса, Эли Бурштейна, Пьера Карпмана, Анжа Альбертини и Ярика Маркова («*The First Collision for Full SHA-1», Proceedings of the 37th CRYPTO Conference,* 2017).

[2] На самом деле теорема Кука — Левина (теоремы 22.1 и 23.2) показывает, что задача SAT универсальна (см. раздел 23.6.3).

дачи о раскраске графа, заданный графом $G = (V, E)$ и целым числом k, можно использовать k переменных в расчете на вершину. Для каждой вершины $v \in V$ и цвета $i \in \{1, 2, ..., k\}$ булева переменная x_{vi} указывает на то, назначен или нет вершине v цвет i.

А как быть с ограничениями? Для ребра $(v, w) \in E$ и цвета i ограничение

$$\neg x_{vi} \vee \neg x_{wi} \tag{21.19}$$

не удовлетворяется, только когда и v, и w окрашены i. В тандеме $|E| \times k$ ограничений формы (21.19) обеспечивают, чтобы ни одно ребро не было монохроматическим.

Мы еще не закончили, поскольку все ограничения формы (21.19) удовлетворяются присвоением всем переменным значения «ложь» (то есть ни одна вершина не получает никакого цвета). Но можно добавить одно ограничение

$$x_{v1} \vee x_{v2} \vee \cdots \vee x_{vk} \tag{21.20}$$

для каждой вершины $v \in V$, которое не удовлетворяется именно тогда, когда v не получает никакого цвета. Каждая k-цветная раскраска графа G транслируется в присвоение истинности, удовлетворяющее всем ограничениям, и опять же, наоборот, каждое присвоение истинности, удовлетворяющее всем ограничениям, кодирует одну или несколько k-цветных раскрасок графа G.[1]

Система ограничений, определяемая формулами (21.19) и (21.20), идеально подходит *решателю задач SAT*. Например, чтобы проверить, можно ли полный граф на трех вершинах раскрасить двухцветной раскраской с помощью решателя MiniSAT — популярного решателя задач о выполнимости с открытым исходным кодом, просто вызовите его из командной строки со следующим входным файлом:

```
p cnf 6 9
1 4 0
```

[1] Ограничения (21.20) позволяют вершинам получать более одного цвета, но ограничения (21.19) обеспечивают, чтобы каждый способ выборки среди назначенных цветов производил k-цветную раскраску графа G.

```
2 5 0
3 6 0
-1 -2 0
-4 -5 0
-1 -3 0
-4 -6 0
-2 -3 0
-5 -6 0
```

Как по волшебству, решатель выдаст правильное заявление о том, что нет никакого способа удовлетворить все ограничения.[1]

21.5.4. Решатели задач SAT: некоторые отправные точки

На момент написания этой книги (в 2020 году) существует много хороших вариантов свободно доступных решателей задач SAT. По меньшей мере раз в два года фанаты этих задач со всего мира собираются и проводят олимпийские игры (с медалями) между самыми последними и лучшими решателями, каждый из которых проверятся целым рядом сложных примеров. В каждом конкурсе участвуют десятки заявок, большинство из которых имеют открытый исходный код.[2] Если вам нужна рекомендация, то могу посоветовать MiniSAT, который сочетает в себе хорошую производительность с простотой использования и разрешительной лицензией MIT.[3]

---

[1] Первая строка файла предупреждает решатель о том, что экземпляр задачи SAT имеет шесть переменных решения и девять ограничений. `cnf` означает «конъюнктивная нормальная форма» и указывает на то, что каждое ограничение является дизъюнкцией литералов. Числа от 1 до 6 относятся к переменным, а «-» указывает на отрицание. Первые три и последние три переменные соответствуют первому цвету и второму цвету в порядке появления. Первые три и последние шесть ограничений имеют форму соответственно, как в (21.20) и (21.19). Нули обозначают концы ограничений.

[2] www.satcompetition.org.

[3] И чтобы подняться в решении задач SAT на следующий уровень, поищите решатели «выполнимости булевых формул в теориях», или «выполнимости формул в теориях», или «SMT-решатели» (satisfiability modulo theories), такие как решатель z3 от компании Microsoft (который также свободно доступен по лицензии MIT).

ВЫВОДЫ

- Решение задачи коммивояжера путем исчерпывающего поиска требует масштабирования по времени вместе с $n!$, где n — это число вершин.
- Алгоритм динамического программирования Беллмана — Хелда — Карпа решает задачу коммивояжера за время $O(n^2 2^n)$.
- Ключевая идея алгоритма Беллмана — Хелда — Карпа состоит в параметризации подзадачи подмножеством вершин, которые должны быть посещены ровно один раз, и вершин, которые должны быть посещены в последнюю очередь.
- В задаче о самом дешевом k-вершинном пути на вход подается неориентированный граф с вещественнозначными реберными стоимостями для вычисления бесциклового пути, посещающего k вершин с минимально возможной суммой реберных стоимостей.
- Решение задачи о самом дешевом k-вершинном пути посредством исчерпывающего поиска требует масштабирования по времени вместе с n^k, где n — это число вершин.
- Алгоритм цветового кодирования решает задачу о самом дешевом k-вершинном пути за время $O((2e)^k m \ln 1/\delta)$, где m — это число ребер и δ — заданная пользователем вероятность неуспеха.
- Первая ключевая идея алгоритма цветового кодирования — использовать подпрограмму динамического программирования, которая при наличии закрепления одного из k цветов за каждой вершиной входного графа вычисляет за время $O(2^k m)$ самый дешевый панхроматический путь.
- Вторая ключевая идея заключается в эксперименте с $O(e^k \ln 1/\delta)$ независимыми и равномерно распределенными случайными раскрасками вершин. С вероятностью по меньшей мере $1 - \delta$ как минимум одна из них будет делать некоторый самый дешевый k-вершинный путь панхроматическим.
- Смешанная целочисленная программа задается числовыми переменными решения, линейными ограничениями и линейной целевой функцией.

- ✶ Большинство задач дискретной оптимизации можно рассматривать как задачи MIP.
- ✶ Экземпляр задачи SAT задается булевыми переменными решения и ограничениями, которые представляют собой дизъюнкции литералов.
- ✶ Большинство задач о проверке допустимости могут быть сформулированы как задачи SAT.
- ✶ Новейшие решатели задач MIP и SAT способны полунадежно решать среднеразмерные экземпляры NP-трудных задач.

Задачи на закрепление материала

Задача 21.1 *(S)*. Опровергает ли алгоритм `BellmanHeldKarp` для задачи коммивояжера (раздел 21.1.6) предположение, что P ≠ NP? (Правильных ответов может быть несколько.)

а) Да.

б) Нет. Полиномиально-временной алгоритм для задачи коммивояжера не обязательно опровергает предположение, что P ≠ NP.

в) Нет. Поскольку алгоритм использует экспоненциальное (по размеру входных данных) число подзадач, он не всегда выполняется за полиномиальное время.

г) Нет. Поскольку алгоритм может выполнять экспоненциальный объем работы для решения одной подзадачи, он не всегда выполняется за полиномиальное время.

д) Нет. Поскольку алгоритм может выполнять экспоненциальный объем работы для извлечения окончательного решения из решений своих подзадач, он не всегда выполняется за полиномиальное время.

Задача 21.2 *(S)*. Для входных данных задачи коммивояжера в тестовом задании 20.7 (с. 102) каковы заключительные записи кучи подзадач в алгоритме `BellmanHeldKarp` (раздел 21.1.6)?

Задача 21.3 *(S).* Рассмотрите следующие предлагаемые подзадачи для графа $G = (V, E)$ экземпляра задачи коммивояжера:

> **ПОДЗАДАЧИ (ПОПЫТКА)**
>
> Вычислить $C_{i,v}$, минимальную стоимость бесциклового пути, который начинается в вершине 1, заканчивается в вершине v и посещает ровно i вершин (либо подтвердить $+\infty$, если такого пути не существует).
>
> (Для каждого $i \in \{2, 3, ..., |V|\}$ и $v \in V - \{1\}$.)

Что мешает использовать эти подзадачи с i в качестве размера подзадачи для разработки полиномиально-временного алгоритма динамического программирования для задачи коммивояжера? (Правильных ответов может быть несколько.)

а) Число подзадач является сверхполиномиальным по размеру входных данных.

б) Оптимальные решения больших подзадач невозможно легко вычислить из оптимальных решений меньших подзадач.

в) Оптимальный тур невозможно легко вычислить из оптимальных решений всех подзадач.

г) Ничего не мешает.

Задача 21.4 *(S).* Какая из следующих задач может быть решена за время $O(n^2 2^n)$ для n-вершинных графов с использованием незначительной вариации алгоритма `BellmanHeldKarp`? (Правильных ответов может быть несколько.)

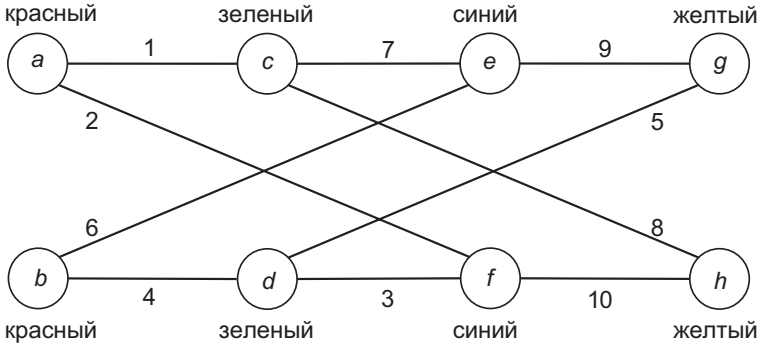
а) Имея n-вершинный неориентированный граф, определить, имеет ли он гамильтонов путь (бесцикловый путь с $n - 1$ ребрами).

б) Имея n-вершинный ориентированный граф, определить, имеет ли он ориентированный гамильтонов путь (бесцикловый ориентированный путь с $n - 1$ ребрами).

в) Имея полный неориентированный граф и вещественнозначные реберные стоимости, вычислить максимальную стоимость тура коммивояжера.

г) Имея полный n-вершинный ориентированный граф (с наличием всех $n(n-1)$ ориентированных ребер) и вещественнозначные реберные стоимости, вычислить минимальную стоимость ориентированного тура коммивояжера (ориентированный цикл, который посещает каждую вершину ровно один раз).

Задача 21.5 *(S)*. Возьмите в качестве экземпляра



Каковы заключительные записи кучи подзадач алгоритма `PanchromaticPath` (раздел 21.2.5)?

Задача 21.6 *(H)*. Предложите реализацию шага постобработки, который восстанавливает самый дешевый тур коммивояжера из кучи подзадач, вычисленной алгоритмом `BellmanHeldKarp`. Можно ли достичь линейного (по числу вершин) времени выполнения (возможно, после добавления в указанный алгоритм дополнительной служебной работы)?

Задача 21.7 *(H)*. Предложить реализацию шага постобработки, который восстанавливает самый дешевый панхроматический путь из кучи подзадач, вычисленной алгоритмом `PanchromaticPath`. Можно ли достичь линейного (по числу цветов) времени выполнения (возможно, после добавления в указанный алгоритм дополнительной служебной работы)?

Задачи повышенной сложности

Задача 21.8 *(H)*. Оптимизируйте алгоритм `BellmanHeldKarp` для задачи коммивояжера (раздел 21.1.6) так, чтобы его потребность в памяти снизилась с $O(n \times 2^n)$ до $O(\sqrt{n} \times 2^n)$ для n-вершинных экземпляров. (Вы несете ответственность только за расчет минимальной стоимости тура, а не за сам оптимальный тур.)

Задача 21.9 *(S)*. Покажите, как закодировать экземпляры следующих задач в виде смешанных целочисленных программ:

а) Независимое множество с минимальным весом (раздел 19.4.2).

б) Минимизация производственной продолжительности (раздел 20.1.1).

в) Максимальный охват (раздел 20.2.1).

Задача 21.10 *(H)*. Имея граф G экземпляра задачи коммивояжера с множеством вершин $V = \{1, 2, ..., n\}$ и реберными стоимостями c, рассмотрите задачу MIP

$$\text{минимизировать} \sum_{i=1}^{n} \sum_{j \neq i} c_{ij} x_{ij} \tag{21.21}$$

$$\text{при условии, что} \sum_{j \neq i} x_{ij} = 1 \left[\text{для каждой вершины } i\right]; \tag{21.22}$$

$$\sum_{j \neq i} x_{ij} = 1 \left[\text{для каждой вершины } i\right]; \tag{21.23}$$

$$x_{ij} \in \{0,1\} \left[\text{для каждого } i \neq j\right]. \tag{21.24}$$

Цель состоит в кодировании тура коммивояжера (ориентированного в одном из двух возможных направлений) с x_{ij} равным 1, если и только если тур посещает j сразу после i. Ограничения (21.22)–(21.23) гарантируют, что каждая вершина имеет ровно одного непосредственного предшественника и одного непосредственного преемника в туре.

а) Докажите, что для каждого графа G экземпляра задачи коммивояжера и тура коммивояжера по графу G существует допустимое решение соответствующей задачи MIP (21.21)–(21.24) с тем же самым значением целевой функции.

б) Докажите, что существует граф G экземпляра задачи коммивояжера и допустимое решение соответствующей задачи MIP (21.21)–(21.24) со значением целевой функции строго меньше минимальной суммарной стоимости тура коммивояжера по графу G. (Эта задача MIP имеет мнимые допустимые решения далеко за пределами туров коммивояжера и неправильно кодирует задачу коммивояжера.)

в) При следующих дополнительных ограничениях:

$$y_{1j} = (n-1)x_{1j} \left[\text{для всех } j \in V - \{1\}\right]; \tag{21.25}$$

$$y_{ij} = (n-1)x_{ij} \left[\text{для всех } i \neq j\right]; \tag{21.26}$$

$$\sum_{i \neq 1} y_{ji} - \sum_{j \neq 1} y_{ij} = 1 \left[\text{для всех } i \in V - \{1\}\right]; \tag{21.27}$$

$$y_{ij} \in \{0, 1, \ldots, n-1\} \left[\text{для всех } i \neq j\right], \tag{21.28}$$

где y_{ij} — это дополнительные переменные решения, докажите (а) заново для расширенной задачи MIP (21.21)–(21.28).

г) Докажите, что для каждого графа G экземпляра задачи коммивояжера каждое допустимое решение соответствующей расширенной задачи MIP (21.21)–(21.28) транслируется в тур коммивояжера по графу G с тем же значением целевой функции. (Как следствие, расширенная задача MIP правильно кодирует задачу коммивояжера.)[1]

Задача 21.11 *(H)*. Покажите, как закодировать экземпляр задачи SAT в виде смешанной целочисленной программы.

---

[1] Добавление еще большего числа ограничений хотя и не обязательно для правильности, дает решателю задач MIP больше подсказок для работы и может привести к значительному ускорению. Например, добавление логически избыточных неравенств $x_{ij} + x_{ji} \leq 1$ для всех $i \neq j$ в расширенную задачу MIP (21.21)–(21.28) обычно уменьшает время ее решения. Современные решатели задач MIP, адаптированные к задаче коммивояжера, такие как решатель Concorde TSP, опираются на *экспоненциально большое* множество дополнительных неравенств, лениво генерируемых по мере необходимости. (Поищите термин «ослабление подтура» (subtour relaxation) для задачи коммивояжера.)

Задача 21.12 *(H)*. Для положительного целого числа k задача k-SAT является частным случаем задачи SAT, в которой каждое ограничение имеет не более k литералов. Покажите, что задача о выполнимости двулитеральных булевых формул (2-SAT) может быть решена за время $O(m + n)$, где m и n обозначают число ограничений и переменных соответственно. (Можно допустить, что входные данные представлены в виде кучи литералов и кучи ограничений с указателями из каждого ограничения на его литералы и из каждого литерала на ограничения, которые его содержат.)[1]

Задача 21.13 *(H)*. Задача выполнимости 3-литеральных булевых формул (3-SAT) является NP-трудной (теорема 22.1). Можно ли усовершенствовать исчерпывающий поиск, который перечисляет все 2^n допустимых присвоений истинности n переменным решениям? Ниже приведен рандомизированный алгоритм Шёнинга `Schöning`, параметризованный рядом испытаний T:

SCHÖNING

Вход: экземпляр задачи 3-SAT с n переменными и вероятность неуспеха $\delta \in (0, 1)$.

Вывод: с вероятностью по меньшей мере $1 - \delta$ присвоение истинности, удовлетворяющее всем ограничениям, либо объявление, что присвоение невозможно.

```
ta := булев массив длины n                  // присвоение истинности
for t = 1 to T do                          // T независимых испытаний
    for i = 1 to n do                  // случайное первое присвоение
        ta[i] := «истина» or «ложь»    // каждый с 50%-м шансом
    for k = 1 to n do                      // n локальных модификаций
        if ta удовлетворяет всем ограничениям then // готово!
            return ta
```

---

[1] Формулировка выполнимости в разделе 21.5.3 может рассматриваться как редукция из задачи k-цветной раскраски к задаче k-SAT. Благодаря этой формулировке алгоритм 2-SAT в этой задаче преобразуется в линейно-временной алгоритм для проверки того, является или нет граф двухцветным (двудольным). В качестве альтернативы двухцветность может быть проверена непосредственно за линейное время с помощью поиска сначала в ширину.

```
    else                    // зафиксировать нарушенное ограничение
        выбрать произвольное нарушение ограничения C
        выбрать переменную x_i в C, равномерно наугад
        ta[i] := ¬ta[i]     // перевернуть ее значение
return «решения нет»        // отказаться от поиска
```

а) Докажите, что всякий раз, когда нет никакого присвоения истинности, удовлетворяющего всем ограничениям данного экземпляра задачи 3-SAT, алгоритм `Schöning` возвращает «решения нет».

б) Для этой и следующих трех частей ограничить внимание входными данными с присвоением истинности, которое удовлетворяет всем ограничениям. Обозначим через p вероятность того, что при подбрасывании монет при помощи алгоритма `Schöning` итерация самого внешнего цикла `for` обнаружит удовлетворяющее присвоение. Доказать, что при $T = 1/p \ln 1/\delta$ независимых случайных испытаниях алгоритм `Schöning` находит удовлетворяющее присвоение с вероятностью по меньшей мере $1 - \delta$.

в) В этой и следующей частях обозначим через ta^* удовлетворяющее присвоение для данного экземпляра задачи 3-SAT. Докажите, что каждое подбрасывание переменной, сделанное алгоритмом `Schöning` в его внутреннем цикле, имеет по меньшей мере 1 из 3 шансов увеличить число переменных с одинаковым значением как в ta, так и в ta^*.

г) Докажите, что вероятность того, что равномерно распределенное случайное присвоение истинности согласуется с ta^* по крайней мере на $n/2$ переменных, составляет 50 %.

д) Докажите, что вероятность p, определенная в (б), по меньшей мере $1/(2 \times 3^{n/2})$; следовательно, при $T = 2 \times 3^{n/2} \ln 1/\delta$ испытаний алгоритм `Schöning` возвращает удовлетворяющее присвоение с вероятностью по меньшей мере $1 - \delta$.

е) Заключите, что существует рандомизированный алгоритм, который решает задачу 3-SAT (с вероятностью неуспеха, не превышающей δ)

за время $O((1{,}74)^n \ln 1/\delta)$, экспоненциально быстрее исчерпывающего поиска.[1]

Задачи по программированию

Задача 21.14. Реализуйте на своем любимом языке программирования алгоритм `BellmanHeldKarp` для задачи коммивояжера (раздел 21.1.6). Как и в задаче 20.15, попробуйте свою реализацию на экземплярах с реберными стоимостями, выбранными независимо и равномерно случайным образом из множества {1, 2, ..., 100}, или, в качестве альтернативы, для вершин, соответствующих точкам, выбранным независимо и с равномерным распределением случайным образом из единичного квадрата (с реберными стоимостями, равными евклидовым расстояниям). Насколько большим является размер входных данных (сколько вершин), который ваша программа способна надежно обрабатывать менее чем за минуту? А что будет через час? Что является самым узким местом: время или память? Поможет ли реализация оптимизации

---

[1] Этот алгоритм был предложен Уве Шёнингом. Его работа «Вероятностный алгоритм для задач k-SAT, основанный на ограниченном локальном поиске и перезапуске» («*A Probabilistic Algorithm for k-SAT Based on Limited Local Search and Restart», Algorithmica*, 2002) достигает границы времени выполнения, равной $O((1{,}34)^n \ln 1/\delta)$, на экземплярах с n переменными посредством более тщательного анализа, а также расширяет алгоритм и анализ до задачи k-SAT для всех k (причем основание в экспоненциальном времени выполнения увеличивается $c \approx 4/3$ до $\approx 2 - 2/k$). С тех пор было разработано несколько более быстрых алгоритмов (как рандомизированных, так и детерминированных), но ни один из них не достиг времени выполнения $O((1{,}3)^n)$.

Раздел 23.5 описывает гипотезу об экспоненциальном времени (ETH, Exponential Time Hypothesis) и сильную гипотезу об экспоненциальном времени (SETH, Strong ETH), которые постулируют, что недостатки алгоритма Шёнинга разделяются всеми алгоритмами для задач k-SAT. Гипотеза об экспоненциальном времени является более смелой формой предположения о том, что P ≠ NP, утверждающей, что решение задачи 3-SAT требует экспоненциального времени $\Omega(a^n)$ для некоторой постоянной $a > 1$, и, следовательно, единственные улучшения, возможные для алгоритма Шёнинга, заключены в основании экспоненты. Сильная гипотеза об экспоненциальном времени утверждает, что основание экспоненты времени выполнения алгоритмов для задач k-SAT должно деградировать до 2 по мере увеличения k.

в задаче 21.8? (Тестовые случаи и наборы данных для сложных задач см. на веб-сайте www.algorithmsilluminated.org.)

Задача 21.15. Попробуйте один или несколько решателей задач MIP на экземплярах задачи коммивояжера из задачи 21.14, используя формулировку задачи MIP из задачи 21.10. Насколько большим является размер входных данных, который решатель способен надежно обрабатывать менее чем за минуту или менее чем за час? Насколько сильно ответ зависит от решателя? Поможет ли добавление дополнительных неравенств из сноски на с. 179? (Тестовые случаи и наборы данных для сложных задач см. на веб-сайте www.algorithmsilluminated.org.)

22 Доказательство NP-трудных задач

Главы 20 и 21 снабдили вас алгоритмическим инструментарием для решения NP-трудных задач: от быстрых эвристических алгоритмов до точных алгоритмов, превосходящих исчерпывающий поиск. Когда нужно использовать этот инструментарий? Если ваш босс вручает вам вычислительную задачу и говорит, что она является NP-трудной, прекрасно. Но что, если вы и есть босс? Задачи в реальном мире не отмечены штампом о вычислительном статусе, а умелое распознавание NP-трудных задач — компетентность уровня 3 (раздел 19.2) — результат тренировок. В этой главе вы сначала потренируетесь распознавать задачу 3-SAT, а затем после проведения восемнадцати редукций получите список из девятнадцати NP-трудных задач. Список послужит отправной точкой для доказательств NP-трудности, а редукции станут шаблонами для вашей дальнейшей работы.

22.1. Редукции повторно

Итак, что такое NP-трудность? В разделе 19.3.7 мы условно определили NP-трудную задачу как задачу, для которой полиномиально-временной алгоритм опровергает предположение, что $P \neq NP$. Это предположение неофициально

обозначает, что проверить готовое решение задачи (например, судоку) может быть фундаментально проще, чем придумать свое решение с нуля. (Глава 23 будет источником строгих определений.) Опровержение предположения, что $P \neq NP$, немедленно решило бы тысячи задач, включая почти все те, которые изучались в этой книге, которые сопротивлялись усилиям бесчисленных умов на протяжении многих десятилетий. То есть NP-трудность является убедительным (если не герметичным) доказательством необходимости компромиссов, описанных в главах 20 и 21.

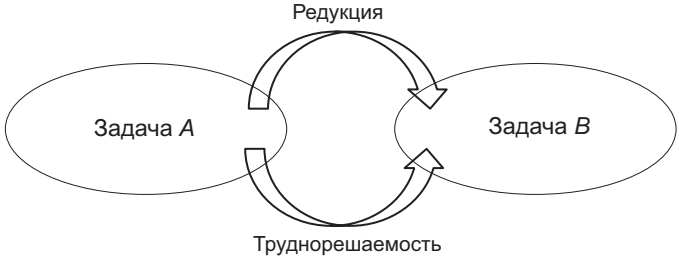
Чтобы применить теорию NP-трудности, на самом деле не нужно понимать причудливые математические определения. Это одна из причин, почему указанная теория была успешно принята повсюду, в том числе в инженерии, естественных и социальных науках.[1] Единственным предварительным условием является понимание редукций, которым вы уже обладаете (раздел 19.5.1):

Формально задача *A сводится* к задаче *B*, если *A* может быть решена с использованием полиномиального (по размеру входных данных) числа вызовов подпрограммы, решающей задачу *B*, наряду с полиномиальным объемом дополнительной работы (вне вызовов подпрограммы). Мы видели несколько примеров редукций, которые распространяют легкорешаемость от одной задачи (*B*) к другой (*A*): если *A* сводится к *B*, то полиномиально-временной алгоритм, решающий *B*, автоматически производит решение для *A* (просто нужно выполнить редукцию, вызывая предполагаемую подпрограмму для *B* по мере необходимости).

---

[1] Вы поймете, что я имею в виду, когда посчитаете результаты поискового запроса терминов «NP-трудный» или «NP-полный» в вашей любимой академической базе данных!

Доказательство NP-трудности переворачивает этот вывод с ног на голову, используя редукцию для нехорошей цели распространения труднорешаемости от одной задачи к другой (в противоположном направлении):



Ибо если NP-трудная задача *A* сводится к *B*, то любой полиномиально-временной алгоритм для *B* автоматически создает его для *A*, тем самым опровергая предположение, что P ≠ NP. То есть *B* должна сама быть NP-трудной.

Итак, как доказать, что задача является NP-трудной? Следуйте двухшаговому рецепту.

КАК ДОКАЗАТЬ, ЧТО ЗАДАЧА ЯВЛЯЕТСЯ NP-ТРУДНОЙ

Чтобы доказать, что задача *B* является NP-трудной, нужно:

1. Выбрать NP-трудную задачу *A*.
2. Доказать, что *A* сводится к *B*.

Остальная часть этой главы дает варианты выбора для *A* на первом шаге и оттачивает ваши навыки редуцирования для второго шага.

NP-трудность типичной NP-трудной задачи *B* может быть доказана с использованием любого числа вариантов для известной NP-трудной задачи *A* на первом шаге. Чем больше *A* напоминает *B*, тем проще детали второго шага. Например, редукция в разделе 19.5.4 задачи об ориентированном гамильтоновом пути к задаче о бесцикловом кратчайшем пути относительно проста из-за сходства между этими задачами.

22.2. Задача 3-SAT и теорема Кука — Левина

Каждое применение двухшагового рецепта идентифицирует одну новую NP-трудную задачу с помощью одной старой. Применив его тысячи раз, вы получите каталог из тысяч NP-трудных задач. Но как этот процесс начинается? С одного из самых важных результатов в computer science: *теоремы Кука — Левина*, которая доказывает с нуля, что кажущаяся безобидной задача 3-SAT является NP-трудной.[1]

Теорема 22.1 (теорема Кука — Левина). *Задача 3-SAT является NP-трудной.*

Задача 3-SAT (введенная в задаче 21.13) является частным случаем задачи SAT (раздел 21.5), в котором каждое ограничение является дизъюнкцией не более трех литералов.[2, 3]

---

[1] Была доказана независимо примерно в 1971 году Стивеном А. Куком и Леонидом Левиным по разные стороны железного занавеса (в Торонто и Москве), хотя потребовалось время, чтобы работу Левина признали на Западе. Пророчество этой теоремы об NP-трудности многих фундаментальных задач было исполнено Ричардом М. Карпом, который в 1972 году с помощью двухшагового рецепта доказал, что неожиданно разнообразные пресловутые задачи являются NP-трудными, а NP-трудность является основным препятствием на пути алгоритмического прогресса во многих разных направлениях. Его первоначальный список из двадцати одной NP-трудной задачи включает большинство из показанных в этой главе задач.

Кук и Карп были удостоены премии ACM Turing Award — эквивалента Нобелевской премии в computer science — соответственно в 1982 и 1985 годах. Левин был удостоен премии Кнута — пожизненной награды за вклад в основы информатики — в 2012 году.

[2] Почему три? Это наименьшее значение k, для которого задача k-SAT является NP-трудной (см. задачу 21.12).

[3] Между теоремой Кука — Левина и замечательными успехами решателей задач SAT нет никакого противоречия (раздел 21.5). Решатели дают лишь полунадежный результат, решая некоторые, но не все экземпляры задач SAT за разумное время. Они не показывают, что задача SAT поддается решению за полиномиальное время, поэтому предположение, что $P \neq NP$, продолжает жить!

ЗАДАЧА: 3-SAT

Вход: список булевых переменных решения $x_1, x_2, ..., x_n$ и список ограничений, каждое из которых является дизъюнкцией не более трех литералов.

Выход: присвоение истинности переменным $x_1, x_2, ..., x_n$, удовлетворяющее всем ограничениям, либо объявление, что такого присвоения истинности не существует.

Например, невозможно выполнить все восемь ограничений

$$x_1 \vee x_2 \vee x_3 \qquad x_1 \vee \neg x_2 \vee x_3 \qquad \neg x_1 \vee \neg x_2 \vee x_3 \qquad x_1 \vee \neg x_2 \vee \neg x_3$$
$$\neg x_1 \vee x_2 \vee x_3 \qquad x_1 \vee x_2 \vee \neg x_3 \qquad \neg x_1 \vee x_2 \vee \neg x_3 \qquad \neg x_1 \vee \neg x_2 \vee \neg x_3,$$

поскольку каждое из них запрещает одно из восьми возможных присвоений истинности. Если какое-то ограничение устранено, то остается одно присвоение истинности, удовлетворяющее остальным семи ограничениям. Экземпляры задачи 3-SAT с удовлетворяющим присвоением истинности и без него называются соответственно *выполнимыми* и *невыполнимыми*.

Задача 3-SAT занимает центральное место в теории NP-трудности как по историческим причинам, так и благодаря своей выразительности и простоте. По сей день именно ее выбирают в качестве задачи A в двухшаговом рецепте.

В этой главе мы примем теорему Кука — Левина на веру. Стоя на плечах этих гигантов, мы будем исходить из того, что только одна задача (3-SAT) является NP-трудной, а затем сгенерируем посредством редукций еще восемь NP-трудных задач. В разделе 23.3.5 изложена высокоуровневая идея, лежащая в основе доказательства теоремы Кука — Левина, и даны направления для дальнейшего изучения.[1]

[1] Это доказательство стоит увидеть хотя бы раз в жизни, но почти никто не помнит кровавых подробностей. Большинство ученых-компьютерщиков просто грамотно применяют теорему Кука — Левина для доказательства NP-трудности, и все.

22.3. Общая картина

У нас много задач, а между ними много редукций, которые придется отслеживать. Так что давайте организуемся.

22.3.1. Ошибка новичка повторно

Как разработчики алгоритмов мы привыкли к редукциям, распространяющим легкорешаемость от одной задачи к другой. Вместе с тем редукции распространяют и труднорешаемость в *противоположном* направлении, по причине чего существует непреодолимое искушение конструировать редукции в неправильном направлении (пятая ошибка новичка из раздела 19.6).

ТЕСТОВОЕ ЗАДАНИЕ 22.1

Раздел 21.4 доказывает, что задача о рюкзаке сводится к задаче MIP. Что из этого следует? (Правильных ответов может быть несколько.)

а) Если задача MIP является NP-трудной, то и задача о рюкзаке тоже.

б) Если задача о рюкзаке является NP-трудной, то и задача MIP тоже.

в) Полунадежный решатель задач MIP может быть переведен в полунадежный алгоритм для задачи о рюкзаке.

г) Полунадежный алгоритм для задачи о рюкзаке может быть переведен в полунадежный решатель задач MIP.

(Ответ и анализ решения см. в разделе 22.3.4.)

22.3.2. Восемнадцать редукций

Рисунок 22.1 подытоживает восемнадцать редукций, из которых (исходя из теоремы Кука — Левина) следует, что все девятнадцать задач на рисунке являются NP-трудными.[1]

Рис. 22.1. Восемнадцать редукций и девятнадцать NP-трудных задач. Стрелка от задачи *A* к задаче *B* указывает на то, что *A* сводится к *B*. Труднорешаемость распространяется в том же направлении, что и редукции, от задачи 3-SAT (которая является NP-трудной по теореме Кука — Левина) к другим восемнадцати задачам

Шесть из этих редукций являются либо прямыми, либо перенесенными контрабандой из предыдущих глав книги.

---

[1] В главе 23 объяснено, что поисковые версии почти всех этих задач являются NP-полными и, как следствие, любая из них может кодировать другую. Для разработчика алгоритмов различие между NP-трудностью и NP-полнотой не имеет первостепенного значения: в любом случае задача не поддается решению за полиномиальное время (исходя из предположения, что $P \neq NP$).

РЕДУКЦИИ, КОТОРЫЕ МЫ УЖЕ ВИДЕЛИ

1. Задача 3-SAT является частным случаем задачи SAT (с. 170) и сводится к ней.
2. Задача о пути коммивояжера (задача 19.7 на с. 62) сводится к задаче о самом дешевом k-вершинном пути (с. 145), поскольку является ее частным случаем, в котором длина пути k равна числу вершин.
3. Лемма 19.2 в разделе 19.5.4 доказывает, что задача об ориентированном гамильтоновом пути (с. 52) сводится к задаче о бесцикловом кратчайшем пути (с. 51).
4. Задача 19.7 доказывает, что задача коммивояжера (с. 22) сводится к задаче о пути коммивояжера.
5. Задача 20.9 доказывает, что задача о максимальном охвате (с. 80) сводится к задаче о максимизации влияния (с. 95).
6. Задача 21.11 доказывает, что задача SAT сводится к задаче MIP (с. 166).

Задачи конца главы охватывают восемь более легких редукций.

НЕКОТОРЫЕ БОЛЕЕ ЛЕГКИЕ РЕДУКЦИИ

1. Задача 22.4: задача о независимом множестве (с. 195) сводится к задаче о клике (с. 219).
2. Задача 22.5: задача о независимом множестве сводится к задаче об охвате вершин (задача 20.4 на с. 125).
3. Задача 22.6: задача о вершинном покрытии сводится к задаче о покрытии множества (задача 20.2 на с. 123).
4. Задача 22.7: задача о сумме подмножества (с. 220) сводится к задаче о рюкзаке (с. 41).
5. Задача 22.8: задача о покрытии множества сводится к задаче о максимальном охвате.

6. Задача 22.9: задача об ориентированном гамильтоновом пути сводится к задаче о неориентированном гамильтоновом пути (с. 207).
7. Задача 22.10: задача о сумме подмножества сводится к задаче о минимизации продолжительности (с. 64).
8. Задача 22.11: задача 3-SAT сводится к задаче о проверке пригодности графа к раскраске тремя цветами (с. 169).[1]

И остался список неотложных дел из четырех наиболее трудных редукций:

БОЛЕЕ ТРУДНЫЕ РЕДУКЦИИ

1. Задача 3-SAT сводится к задаче о независимом множестве (раздел 22.5).
2. Задача 3-SAT сводится к задаче об ориентированном гамильтоновом пути (раздел 22.6).
3. Задача о неориентированном гамильтоновом пути сводится к задаче коммивояжера (раздел 22.7). (Это не так уж и сложно.)
4. Задача о независимом множестве сводится к задаче о сумме подмножества (раздел 22.8).

22.3.3. Зачем продираться через доказательства NP-трудности?

Буду честен, доказательства NP-трудности могут быть запутанными и своеобразными, и почти никто не помнит их деталей. Зачем мучить вас ими целые пять разделов? Есть несколько веских причин:

[1] Она меняет направление редукции в разделе 21.5. Цель состоит в распространении (в худшем случае) труднорешаемости, а не полунадежной легкорешаемости. Кроме того, открою тайну: эта редукция является несколько более трудной, чем в задачах 22.4–22.10.

ЦЕЛИ РАЗДЕЛОВ 22.4–22.8

1. Выполнить ранее данное обещание показать, что все задачи, изучаемые в этой книге, являются NP-трудными и, следовательно, требуют компромиссов, описанных в главах 20 и 21.
2. Дать вам длинный список известных NP-трудных задач для использования в ваших собственных редукциях (на первом шаге двухшагового рецепта).[1]
3. Придать вам уверенность в том, что при необходимости вы можете разработать редукции, требующиеся для доказательства NP-трудности задачи, которая вам встретится.

22.3.4. Решение к тестовому заданию 22.1

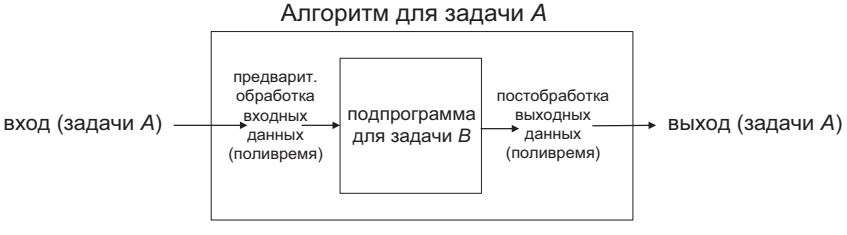
Правильные ответы: (б), (в). Редукция из задачи *A* к задаче *B* распространяет легкорешаемость от *B* к *A* и труднорешаемость в противоположном направлении (рис 19.2 и 19.3). Принимая *A* и *B* в качестве задач соответственно о рюкзаке и MIP, редукция в разделе 21.4 переносит легкорешаемость от задачи MIP к задаче о рюкзаке (то есть (в) верен) и труднорешаемость в обратном направлении (то есть (б) верен).

22.4. Шаблон для редукций

Редукции для доказательства NP-трудности делаются по лекалу. Конечно, редукция из задачи *A* к задаче *B* может иметь изощренный характер, вызывая предполагаемую подпрограмму для *B* любое полиномиальное число раз и обрабатывая ее ответы за полиномиальное время хитроумными способами (раздел 22.1). Но как выглядит самая простая редукция?

[1] Действительно длинный список (с более чем 300 NP-трудными задачами) содержится в классической книге «Компьютеры и труднорешаемость: руководство по теории NP-полноты» Майкла Р. Гэри и Дэвида С. Джонсона («*Computers and Intractability: A Guide to the Theory of NPCompleteness», Michael R. Garey and David S. Johnson, Freeman,* 1979). Немногие книги по computer science 1979 года остаются такими же полезными, как эта!

Если задача *A* является NP-трудной (и предположение, что P ≠ NP, истинно), то каждая редукция *A* к *B* должна использовать подпрограмму для *B* не менее одного раза, чтобы не стать автономным полиномиально-временным алгоритмом для *A*. И для проверки типов, возможно, потребуется предварительно обработать экземпляр задачи *A*, предоставляемый для редукции до его подачи в подпрограмму для *B* — например, если на входе граф, а подпрограмма ожидает список целых чисел. Также и ответ подпрограммы может нуждаться в постобработке для его использования в качестве выходных данных редукции:



Для доказательств NP-трудности нам хватит знания простейшей редукции:

ПРОСТЕЙШАЯ МЫСЛИМАЯ РЕДУКЦИЯ A К B

1. *Препроцессор*: преобразовывает экземпляр задачи *A* за полиномиальное время в экземпляр задачи *B*.
2. *Подпрограмма*: вызывает предполагаемую подпрограмму для *B*.
3. *Постпроцессор*: приводит выходные данные подпрограммы за полиномиальное время в соответствие требованиям экземпляра *A*.

Препроцессор и постпроцессор, как правило, конструируются в тандеме, причем преобразование первого явно ориентируется на потребности последнего. Во всех наших примерах будет видно, что препроцессор и постпроцессор работают за полиномиальное (если не линейное) время.

Редукция в лемме 19.2 из задачи об ориентированном гамильтоновом пути к задаче о бесцикловом кратчайшем пути является простейшей. Она исполь-

зует препроцессор, который конвертирует экземпляр первой задачи в один из экземпляров второй, многократно используя тот же граф и назначая каждому ребру длину –1, и постпроцессор, который немедленно делает правильное логическое заключение из выходных данных подпрограммы бесциклового кратчайшего пути. Редукции в следующих четырех разделах — это более сложные варианты одной и той же идеи.

22.5. Задача о независимом множестве является NP-трудной

Первое доказательство NP-трудности этой главы относится к задаче о *независимом множестве*, в которой на вход подается неориентированный граф $G = (V, E)$ для вычисления независимого множества (то есть множества взаимно несмежных вершин) максимального размера.[1] Например, если G есть циклический граф с n вершинами, то максимальный размер независимого множества равен $n/2$ (если n четное число) или $(n - 1)/2$ (если n нечетное). Если ребра представляют конфликты между людьми или задачами, то независимые множества соответствуют бесконфликтным подмножествам.

Сейчас в нашем распоряжении есть только одна NP-трудная задача — задача 3-SAT. Поэтому доказывать NP-трудность задачи о независимом множестве мы будем через редукцию к ней из задачи 3-SAT. Эти две задачи имеют мало общего: одна касается логики, а другая — графов. Тем не менее результат этого раздела таков:

Теорема 22.2 (редукция из задачи 3-SAT к задаче о независимом множестве). *Задача 3-SAT сводится к задаче о независимом множестве.*

Согласно двухшаговому рецепту, поскольку задача 3-SAT является NP-трудной (теорема 22.1), то же самое относится и к задаче о независимом множестве.

---

[1] Указанная задача является частным случаем задачи о взвешенном независимом множестве (с. 41), в которой вес каждой вершины равен 1. Если частный случай является NP-трудным (как мы увидим), то более общая задача тоже является таковой.

Следствие 22.3 (NP-трудность задачи о независимом множестве). *Задача о независимом множестве является NP-трудной.*

22.5.1. Главная идея

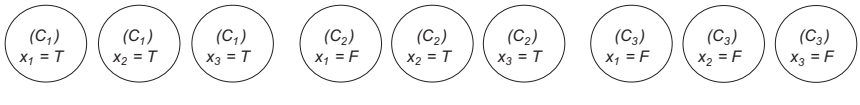
Редукция из задачи об ориентированном гамильтоновом пути к задаче о бесцикловом кратчайшем пути в лемме 19.2 использовала сильное сходство между этими задачами. Если же задачи не связаны, но мы стремимся к простейшей мыслимой редукции (раздел 22.4), что просить у препроцессора и постпроцессора? Постпроцессор должен каким-то образом извлечь удовлетворяющее присвоение истинности для экземпляра задачи 3-SAT (либо сделать вывод, что его не существует) из наибольшего независимого множества графа, изготовленного препроцессором. Проиллюстрируем эти идеи на примере (раздел 22.5.2 содержит формальное описание редукции и доказательство ее правильности).

Чтобы объяснить препроцессор редукции, представьте дизъюнкцию из k литералов как чей-то список запросов на присвоение значений его k любым переменным. Например, ограничение $x_1 \vee x_2 \vee x_3$ спрашивает: «Не могли бы вы установить x_1 равным “ложь”?»; «Или как насчет того, чтобы сделать x_2 равным “истина”?»; «Ну или хотя бы x_3 — равным “истина”?» Удовлетворите по меньшей мере одно из их требований, и препроцессор уйдет счастливым, а ограничение будет удовлетворено.

Ключевая идея препроцессора состоит в кодировании экземпляра задачи 3-SAT в виде графа, каждая вершина которого представляет один запрос на присвоение со стороны одного ограничения.[1] Например, ограничения

$$\underbrace{x_1 \vee x_2 \vee x_3}_{C_1} \quad \underbrace{\neg x_1 \vee x_2 \vee x_3}_{C_2} \quad \underbrace{\neg x_1 \vee \neg x_2 \vee \neg x_3}_{C_3}$$

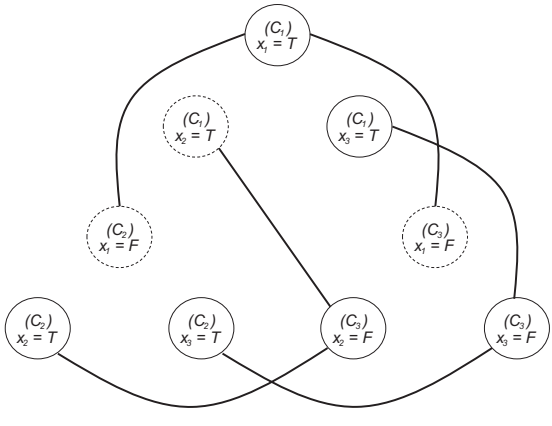
будут представлены тремя группами из трех вершин каждая:



[1] Возможно, вы захотите превратить экземпляр задачи 3-SAT с n переменными в граф с n вершинами, где 2^n подмножеств вершин соответствуют 2^n возможным присвоениям истинности. Увы, здесь такой простой подход не сработает.

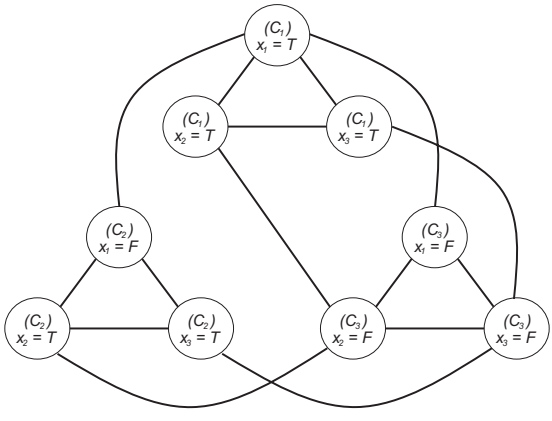
Четвертая вершина, например, кодирует просьбу второго ограничения установить переменную x_1 равной «ложь» (соответствующую ее литералу $\neg x_1$).

Обращаясь к постпроцессору, встает вопрос, кодируют ли подмножества этих вершин присвоения истинности? Да, но не всегда. Проблема в том, что некоторые запросы несовместимы и требуют противоположных назначений одной и той же переменной (например, первая и четвертая вершины выше). Но помните, что смысл задачи о независимом множестве — представить конфликт! Поэтому препроцессор должен добавлять ребро между каждой парой вершин, которое соответствует несовместимым присвоениям. Поскольку каждое независимое множество должно выбирать не более одной конечной точки на ребро, все конфликты будут исключены. Применим эту идею к нашему рабочему примеру (где пунктирные вершины указывают на одно конкретное независимое множество):



Постпроцессор теперь может извлекать удовлетворяющее присвоение истинности из любого независимого множества *S*, содержащего хотя бы одну вершину в каждой группе, просто выполнив все запросы на присвоение значений переменным, результатом чего станет как минимум один выполненный запрос в расчете на ограничение. (Переменной без запросов в любом направлении может быть безопасно присвоено либо «истина», либо «ложь».) Поскольку все вершины *S* не смежные, запросы не конфликтуют. Например, три приведенные выше пунктирные вершины будут транслироваться в одно из двух удовлетворяющих присвоений {ложь, истина, истина} или {ложь, истина, ложь}.

Наконец, редукция должна также распознавать невыполнимые экземпляры задачи 3-SAT. Как мы увидим в следующем разделе, препроцессор может сделать невыполнимость для постпроцессора очевидной с помощью добавления ребра между каждой парой вершин одной группы:



22.5.2. Доказательство теоремы 22.2

Доказательство теоремы 22.2 просто масштабирует пример и рассуждения из раздела 22.5.1 до общих экземпляров задачи 3-SAT.

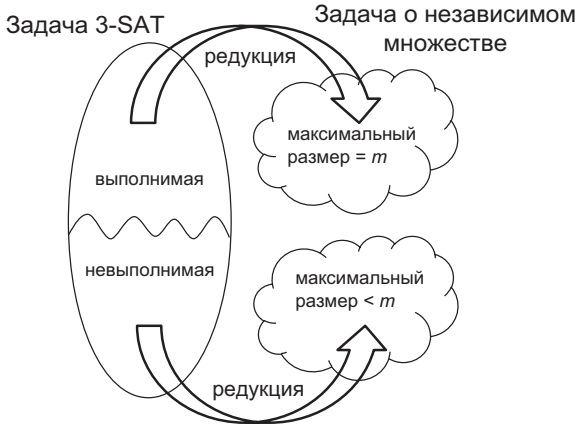
Описание редукции

Препроцессор. Получив произвольный экземпляр задачи 3-SAT с n переменными и m ограничениями, препроцессор строит граф $G = (V, E)$. Он определяет $V = V_1 \cup V_2 \cup ... \cup V_m$, где V_j — это группа с одной вершиной в расчете на литерал j-го ограничения, и $E = E_1 \cup E_2$, где E_1 содержит одно ребро в расчете на пару вершин, находящихся в одной группе, а E_2 содержит одно ребро в расчете на пару конфликтующих вершин (соответствующих запросам на противоположные присвоения истинности одной и той же переменной).

Постпроцессор. Если предполагаемая подпрограмма возвращает независимое множество графа G, построенного препроцессором по меньшей мере с m вершинами, то постпроцессор возвращает произвольное присвоение истинности.

Доказательство правильности

В правильной редукции препроцессор транслирует выполнимые и невыполнимые экземпляры задачи 3-SAT в графы с размером независимого множества, не превышающим m:



Случай 1: невыполнимые экземпляры. Предположим, что для доказательства от противного редукция не возвращает «решения нет» для некоторого невыполнимого экземпляра задачи 3-SAT. Это означает, что подпрограмма возвращает независимое множество S графа $G = (V, E)$, построенное препроцессором и включающее по меньшей мере m вершин. Ребра группы E_1 исключают более одной вершины из группы, поэтому S должно иметь ровно m вершин — по одной на группу. Из-за ребер в E_2 хотя бы одно присвоение истинности согласуется со всеми запросами на присвоение, соответствующими вершинам S. Поскольку S включает в себя одну вершину из каждой группы, присвоение истинности, извлеченное из S постпроцессором, будет удовлетворять всем ограничениям. Это противоречит допущению, что данный экземпляр задачи 3-SAT является невыполнимым.

Случай 2: выполнимые экземпляры. Предположим, что экземпляр задачи 3-SAT имеет удовлетворимое присвоение истинности. Выберем один выполненный запрос на присвоение истинности переменной из каждого ограничения — поскольку оно удовлетворяет каждому ограничению, достаточно

одного запроса — и обозначим через X соответствующее подмножество из m вершин. Множество X является независимым множеством графа G, не содержит конечных точек ни одного ребра из E_1 (поскольку содержит только одну вершину в расчете на группу) и из E_2 (поскольку получено из совместимого присвоения истинности). При наличии хотя бы одного независимого от размера m множества графа G подпрограмма должна возвращать что-то одно — или X, или другое множество, имеющее одну вершину в расчете на группу. Как и в случае 1, постпроцессор извлечет из независимого множества удовлетворяющее присвоение, которое вернет как правильный результат редукции. *Ч. Т. Д.*

Чтобы такие доказательства не выглядели слишком педантично, давайте завершим этот раздел примером редукции, которая пошла наперекосяк.

ТЕСТОВОЕ ЗАДАНИЕ 22.2

В каком месте доказательство теоремы 22.2 разрушается, если внутригрупповые ребра в E_1 опущены из графа G? (Правильных ответов может быть несколько.)

а) Независимое множество графа G больше не транслируется в четко определенное присвоение истинности.

б) Выполнимый экземпляр задачи 3-SAT не нужно транслировать в граф, в котором максимальный размер независимого множества равен по меньшей мере m, числу ограничений.

в) Невыполнимый экземпляр задачи 3-SAT не нужно транслировать в граф, в котором максимальный размер независимого множества меньше m.

г) На самом деле доказательство по-прежнему работает.

(Ответ и анализ решения см. ниже.)

Правильный ответ: (в). При невыполнимом экземпляре задачи 3-SAT и даже без внутригрупповых ребер из E_1 ни одно независимое множество графа G не содержит одну вершину из каждой из m групп (поскольку постпроцессор мог бы перевести любое такое независимое множество в удовлетворимое присвоение). Однако, поскольку независимые множества графа G теперь могут

свободно рекрутировать многочисленные вершины из группы, одна из них вполне может включать m вершин (или больше).

*22.6. Ориентированный гамильтонов путь является NP-трудным

Имея за плечами редукцию из задачи 3-SAT к графовой задаче, почему бы не сделать еще одну? В задаче об *ориентированном гамильтоновом пути* (с. 52) на вход подается ориентированный граф $G = (V, E)$, стартовая вершина $s \in V$ и финальная вершина $t \in V$. Нужно вернуть s-t-вершинный путь, посещающий каждую вершину графа G ровно один раз (именуемый *s-t-вершинным гамильтоновым путем*), либо правильно объявить, что такого пути не существует[1]. В отличие от большинства из девятнадцати задач этой главы, здесь интерес представляет не столько применение задачи, сколько ее полезность для доказательства NP-трудности других важных задач.

Главный результат этого раздела таков:

Теорема 22.4 (редукция из задачи 3-SAT к задаче об ориентированном гамильтоновом пути). *Задача 3-SAT сводится к задаче об ориентированном гамильтоновом пути.*

В сочетании с теоремой Кука — Левина (теорема 22.1) и двухшаговым рецептом теорема 22.4 выполняет обещание, данное еще в разделе 19.5.4:

Следствие 22.5 (NP-трудность задачи об ориентированном гамильтоновом пути). *Задача об ориентированном гамильтоновом пути является NP-трудной.*

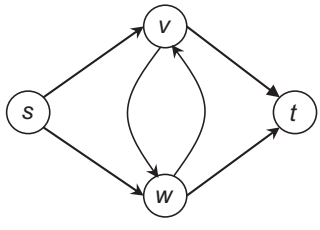
22.6.1. Кодирование переменных и присвоение истинности

Чтобы обойтись простейшей мыслимой редукцией (раздел 22.4), нужен план для препроцессора (ответственного за создание ориентированного графа из

[1] Ответом к задаче на с. 52 будет «да» или «нет», а не путь. Задача 22.3 просит вас показать, что две версии задачи являются эквивалентными и сводятся друг к другу.

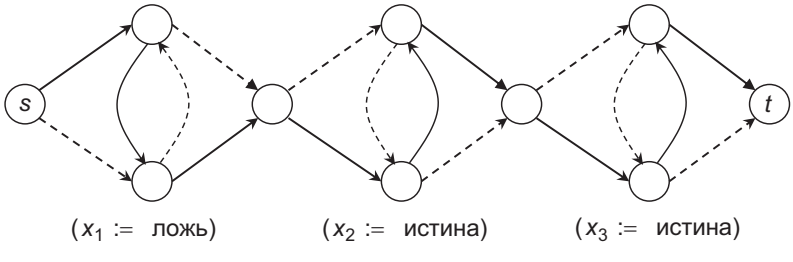
экземпляра задачи 3-SAT) и постпроцессора (ответственного за извлечение удовлетворяющего присвоения истинности из *s*-*t*-вершинного гамильтонова пути этого графа).

Сначала построим граф, в котором *s*-*t*-вершинный гамильтонов путь делает ряд двоичных решений, которые затем могут быть интерпретированы как присвоения истинности постпроцессором. Например, в алмазе



есть два *s*-*t*-вершинных гамильтоновых пути: один зигзагообразно ориентирован вниз ($s \to v \to w \to t$), а другой — вверх ($s \to w \to v \to t$). Идентифицируя ориентированность вниз и вверх как «истину» и «ложь», *s*-*t*-вершинные гамильтоновы пути кодируют возможные присвоения истинности одной булевой переменной.

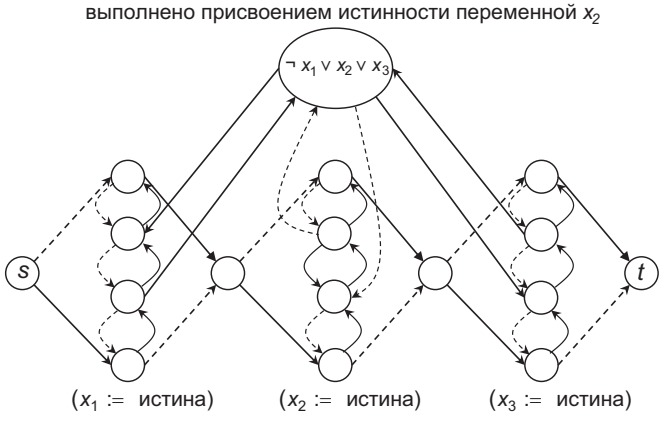
А как быть с другими переменными? Препроцессор будет разворачивать один алмазный граф в расчете на переменную, который будет сцеплен в ожерелье. Например, пунктирный *s*-*t*-вершинный гамильтонов путь в ожерельеподобном графе



может быть интерпретирован постпроцессором как присвоение истинности {ложь, истина, истина}. Остальные *s*-*t*-вершинные гамильтоновы пути схожим образом закодируют остальные семь присвоений истинности.

22.6.2. Кодирование ограничений

Далее препроцессор должен дополнить граф, чтобы отразить ограничения экземпляра задачи 3-SAT: только удовлетворяющие присвоения истинности выживают как *s*-*t*-вершинные гамильтоновы пути. Можно добавлять по одной новой вершине на ограничение, так чтобы ее посещение соответствовало удовлетворению ограничения. Рассмотрим ограничение $\neg x_1 \vee x_2 \vee x_3$ и следующий граф (где пунктирные ребра указывают на один конкретный *s*-*t*-вершинный гамильтонов путь):



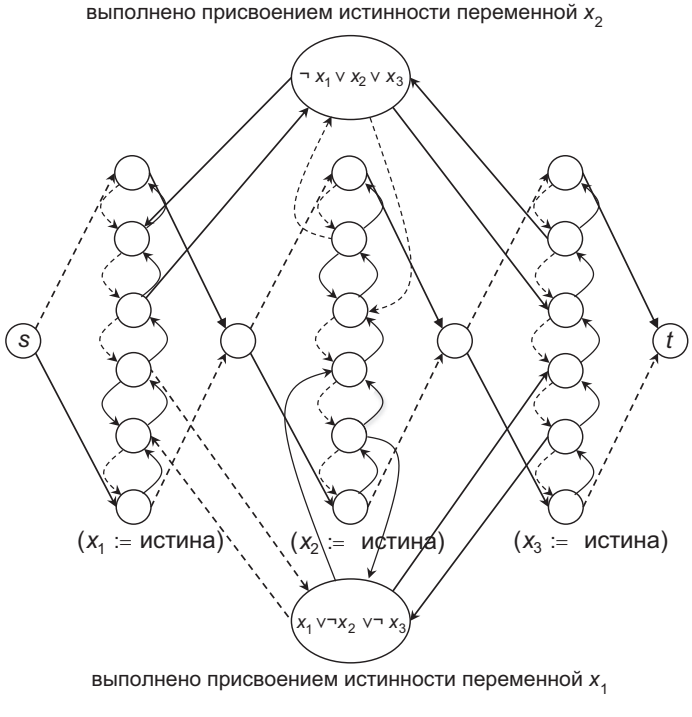
Ребра между ожерельем и новой вершиной позволяют посещать эту вершину по *s*-*t*-вершинным гамильтоновым путям только из алмазных графов, которые пересекаются в направлении, соответствующем удовлетворяющему присвоению истинности.[1]

Рассмотрим пунктирные ребра. *s*-*t*-вершинный гамильтонов путь проходит вниз в каждом из трех алмазов, что соответствует присвоению всем переменным только значений «истина». Присвоение переменой x_1 значения «истина» не удовлетворяет ограничению $\neg x_1 \vee x_2 \vee x_3$. Соответственно, нельзя посетить новую вершину из первого алмаза без пропуска какой-то вершины

[1] Поскольку в этом ограничении участвует каждая переменная, каждый алмаз имеет ребро как в ограничительную вершину, так и из нее. Если бы некоторая переменная в ограничении отсутствовала, то соответствующий алмаз не имел бы таких ребер.

или посещения ее дважды. Присвоение переменной x_2 значения «истина» удовлетворяет ограничению, поэтому пунктирный путь может совершить быструю однодневную поездку туда и обратно к ограничительной вершине из второго алмаза, прежде чем возобновить свое нисходящее путешествие от места остановки. Присвоение переменной x_3 также удовлетворяет ограничению, и однодневная поездка возможна с третьего алмаза тоже. Однако с уже посещенной ограничительной вершиной *s*-*t*-вершинный гамильтонов путь пойдет прямо вниз по третьему алмазу и перейдет к *t*. (Есть второй *s*-*t*-вершинный гамильтонов путь, соответствующий тому же присвоению истинности, проходящий прямо вниз по второму алмазу и совершающий однодневную поездку из третьего.)

Чтобы закодировать второе ограничение, например $x_1 \vee \neg x_2 \vee \neg x_3$, препроцессор может добавить еще одну новую вершину и подсоединить ее к ожерелью.



Две новые вершины в каждом алмазе предоставляют место для любых *s*-*t*-вершинных гамильтоновых путей, которые могут совершать однодневные по-

ездки туда и обратно к обеим ограничительным вершинам из одного алмаза.[1] Пунктирный путь — это один из двух s-t-вершинных гамильтоновых путей, соответствующих присвоению переменным только значений «истина». Посетить новую ограничительную вершину можно только из первого алмаза. Из остальных семи присвоений истинности пять удовлетворяющих соответствуют одному или нескольким s-t-вершинным гамильтоновым путям, а два других — нет.

22.6.3. Доказательство теоремы 22.4

Доказательство теоремы 22.4 масштабирует пример из раздела 22.6.2 до общих экземпляров задачи 3-SAT.

Описание редукции

Препроцессор. Получив экземпляр задачи 3-SAT с n переменными и m ограничениями, препроцессор строит ориентированный граф:

- Определяет множество V из $2mn + 3n + m + 1$ вершин (стартовая вершина s), 3 внешних алмазных вершины v_i, w_i, t_i для каждой переменной x_i, $2m$ внутренних алмазных вершин $a_{i,1}$, $a_{i,2}$, ..., $a_{i,2m}$ для каждой переменной i, и m ограничительных вершин c_1, c_2, ..., c_m.
- Определяет для ожерелья множество ребер E_1 путем присоединения s к v_1 и w_1, t_i к v_{i+1} и w_{i+1} для каждого $i = 1, 2, ..., n - 1$, v_i и w_i к t_i, v_i к и из $a_{i,1}$ и w_i в и из $a_{i,2m}$ для каждого $i = 1, 2, ..., n$; и $a_{i,j}$ к и из $a_{i,j+1}$ для каждого $i = 1, 2, ..., n$ и $j = 1, 2, ..., 2m - 1$.
- Определяет множество ограничительных ребер E_2 путем присоединения $a_{i,2j-1}$ к c_j и c_j к $a_{i,2j}$ всякий раз, когда j-е ограничение содержит литерал x_i (то есть запрашивает x_i = истина) и $a_{i,2j}$ к c_j и c_j к $a_{i,2j-1}$ всякий раз, когда j-е ограничение содержит литерал $\neg x_i$ (запрашивает x_i = ложь).

Препроцессор завершает работу графом $G = (V, E_1 \cup E_2)$. Стартовая и финальная вершины построенного экземпляра определяются, соответственно, как s и t_n.

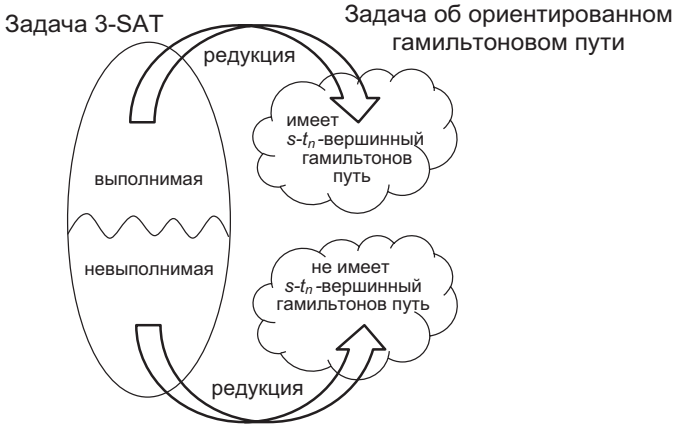
---

[1] В этом примере таких путей нет, но они были бы при втором ограничении, скажем, $x_1 \vee \neg x_2 \vee x_3$.

Постпроцессор. Если предполагаемая подпрограмма вычисляет s-t_n-вершинный гамильтонов путь P графа G, построенного препроцессором, то постпроцессор возвращает присвоение истинности, в котором переменная x_i устанавливается равной «истина», если P посещает вершину v_i перед w_i, и равной «ложь» в противном случае. Если предполагаемая подпрограмма отвечает «решения нет», то постпроцессор тоже отвечает «решения нет».

Доказательство правильности

В правильной редукции препроцессор должен транслировать выполнимые и невыполнимые экземпляры задачи 3-SAT в графы соответственно с s-t_n-вершинным гамильтоновым путем и без него:



Случай 1: невыполнимые примеры. Предположим, что редукция не возвращает «решения нет» для невыполнимого экземпляра задачи 3-SAT. Тогда подпрограмма возвращает s-t_n-вершинный гамильтонов путь P в графе G, построенном препроцессором. Путь P, как и пути, описанные в разделе 22.6.2, должен проходить каждый алмаз вверх или вниз, а также посещать каждую ограничительную вершину. Для последнего путь проведет однодневное путешествие туда и обратно, прерывающее обход некоторого алмаза в направлении, соответствующем запросу на присвоение значения одной из

ограничительных переменных. (Если путь не может сразу вернуться из ограничительной вершины в тот же алмаз, у него не будет возможности посетить остальную часть алмаза позже, не посетив какую-то вершину дважды.) Поэтому присвоение истинности, извлеченное из P постпроцессором, было бы удовлетворяющим присвоением, противоречащим допущению о том, что данный экземпляр задачи 3-SAT невыполним.

Случай 2: выполнимые случаи. Предположим, что экземпляр задачи 3-SAT имеет удовлетворимое присвоение истинности. Тогда построенный препроцессором граф G имеет s-t_n-вершинный гамильтонов путь, пересекающий каждый алмаз в направлении, предложенном присвоением (вниз для переменных, установленных равными «истина», вверх для остальных), совершив однодневную поездку туда и обратно к каждой ограничительной вершине при первой же возможности (из алмаза, соответствующего первой переменной, присвоение значения которой удовлетворяет ограничению). При наличии хотя бы одного s-t_n-вершинного гамильтонова пути в графе G подпрограмма должна вернуть его. Как и в случае 1, постпроцессор затем извлекает из этого пути и возвращает удовлетворимое присвоение. *Ч. Т. Д.*

22.7. Задача коммивояжера является NP-трудной

Вернемся к давно волнующей нас задаче коммивояжера из раздела 19.1.2.

22.7.1. Задача о неориентированном гамильтоновом пути

Возьмем за основу для плана NP-трудную задачу об ориентированном гамильтоновом пути (следствие 22.5) и редукцию из раздела 19.5.4 к задаче о бесцикловом кратчайшем пути. Но нам нужно будет обойти ошибку прямой проверки типов, поскольку задача коммивояжера касается неориентированных, а не ориентированных графов. Поэтому возьмем *неориентированную* версию задачи о гамильтоновом пути.

ЗАДАЧА: НЕОРИЕНТИРОВАННЫЙ ГАМИЛЬТОНОВ ПУТЬ

Вход: неориентированный граф $G = (V, E)$, стартовая вершина $s \in V$ и финальная вершина $t \in V$.

Выход: s-t-вершинный путь в G, который посещает каждую вершину ровно один раз (то есть s-t-вершинный гамильтонов путь), либо правильное объявление, что такого пути не существует.

Задача 22.9 просит показать, что задачи о неориентированном и ориентированном гамильтоновом пути являются эквивалентными и сводятся друг к другу. Следствие 22.5 переносится и на неориентированные графы.

Следствие 22.6 (NP-трудность задачи о неориентированном гамильтоновом пути). *Задача о неориентированном гамильтоновом пути является NP-трудной.*

Главный результат этого раздела тогда таков:

Теорема 22.7 (редукция из задачи о неориентированном гамильтоновом пути к задаче коммивояжера). *Задача о неориентированном гамильтоновом пути сводится к задаче коммивояжера.*

Объединение этой редукции со следствием 22.6 показывает, что задача коммивояжера действительно является NP-трудной.

Следствие 22.8 (NP-трудность задачи коммивояжера). *Задача коммивояжера является NP-трудной.*

22.7.2. Доказательство теоремы 22.7

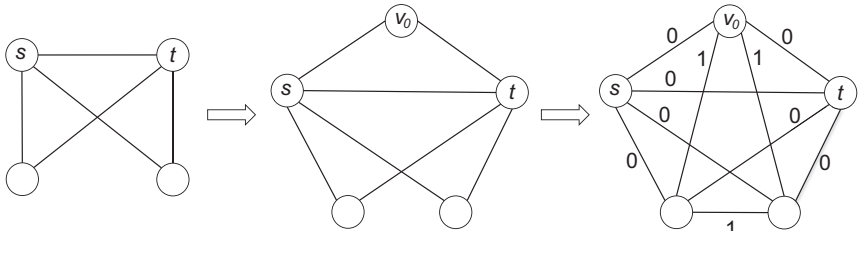
Вздохнем с облегчением: теорема 22.7, в отличие от теорем 22.2 и 22.4, связывает две похожие задачи. Но как конвертировать экземпляр неориен-

тированного гамильтонова пути в экземпляр задачи коммивояжера, чтобы гамильтонов путь (либо объявление, что его не существует) можно было извлечь из самого дешевого тура коммивояжера? Потребуется симулирование недостающих ребер более дорогими ребрами.

Описание редукции

Препроцессор. Получив неориентированный граф $G = (V, E)$, стартовую вершину s и финальную вершину t, препроцессор сначала наводит мост между путями и циклами, посещающими все вершины, дополняя G новой вершиной v_0 и ребрами, соединяющими v_0 с s и t. Затем он назначает стоимость 0 всем ребрам в дополненном графе. Для того чтобы завершить построение экземпляра задачи коммивояжера, препроцессор добавляет все недостающие ребра (чтобы сформировать полный граф G' с множеством вершин $V \cup \{v0\}$) и назначает каждой из них стоимость 1.

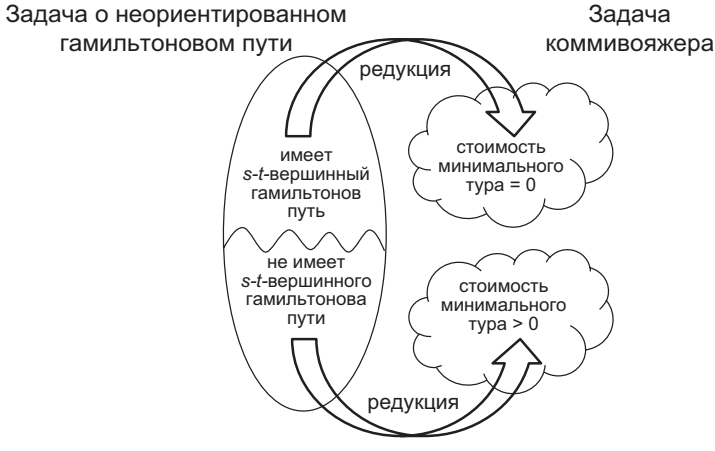
Например, препроцессор транслирует следующий ниже граф без s-t-вершинного гамильтонова пути в экземпляр задачи коммивояжера с отсутствующим нуль-стоимостным туром:



Постпроцессор. Если подпрограмма вычисляет нуль-стоимостной тур T коммивояжера в графе G', построенном препроцессором, то постпроцессор удаляет v_0 и два его инцидентных ребра из T и возвращает результирующий путь. В противном случае, как и в приведенном выше примере, постпроцессор сообщает: «решения нет».

Доказательство правильности

Чтобы доказать правильность редукции, обоснуем такой рисунок:



Случай 1: негамильтоновы экземпляры. Предположим, что редукция не возвращает «решения нет» для графа G экземпляра задачи о неориентированном гамильтоновом пути, который не имеет s-t-вершинного гамильтонова пути. Тогда подпрограмма возвращает нуль-стоимостной тур T графа G', построенного препроцессором. Этот тур избегает все ребра стоимостью 1 в G'. Поскольку только ребра графа G и ребра (v_0, s) и (v_0, t) имеют стоимость нуль в G' и два ребра пути T, то ребра, инцидентные v_0, должны быть равны (v_0, s) и (v_0, t), а остаток пути T должен быть бесцикловым s-t-вершинным путем, посещающим все вершины в V, используя только ребра в G. То есть $T - \{(v_0, s), (v_0, t)\}$ — это s-t-вершинный гамильтонов путь в G, что противоречит первому допущению.

Случай 2: гамильтоновы экземпляры. Предположим, что данный экземпляр неориентированного гамильтонова пути имеет s-t-вершинный гамильтонов путь P. Тогда граф G' экземпляра задачи коммивояжера, построенный препроцессором, имеет нуль-стоимостной тур в $P \cup \{(v_0, s), (v_0, t)\}$. При наличии хотя бы одного нуль-стоимостного тура подпрограмма должна вернуть его. Как и в случае 1, постпроцессор извлечет из этого тура и возвратит s-t-вершинный гамильтонов путь в G. *Ч. Т. Д.*

22.8. Задача о сумме подмножества является NP-трудной

Последним в нашем параде доказательств NP-трудности будет доказательство для задачи о *сумме подмножества*.[1] Как следствие, и задача о рюкзаке, и задача о минимизации производственной продолжительности также являются NP-трудными (см. задачи 22.7 и 22.10).

ЗАДАЧА: СУММА ПОДМНОЖЕСТВА

Вход: положительные целые числа $a_1, a_2, ..., a_n$ и положительное целое число t.

Выход: подмножество элементов a_i с суммой, равной t (либо объявление, что такого подмножества не существует.)

Если элементы a_i — это все степени десяти от 1 до 10^{100}, то существует подмножество с целевой суммой t, если и только если t (записанное с основанием 10) не превышает 101 вариант комбинаций цифр 0 или 1.

В центре внимания задачи о сумме подмножества — горстка чисел. Казалось бы, указанная задача не имеет ничего общего с задачами о сложных графах. Тем не менее главный результат этого раздела таков:

Теорема 22.9 (редукция из задачи о независимом множестве к задаче о сумме подмножества). *Задача о независимом множестве сводится к задаче о сумме подмножества.*

Этот результат в сочетании со следствием 22.3 показывает, что:

Следствие 22.10 (NP-трудность задачи о сумме подмножества). *Задача о сумме подмножества является NP-трудной.*

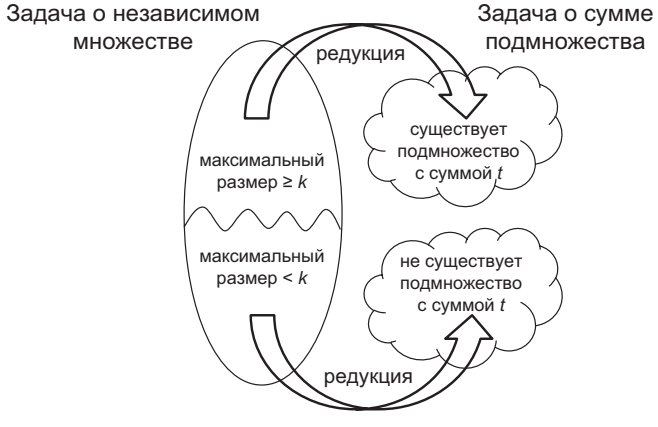
---

[1] Суть задачи о сумме подмножества такова: имея множество неотрицательных целых чисел и сумму значений, определить, существует ли подмножество данного множества с суммой, равной данной сумме. — *Примеч. пер.*

22.8.1. Базовый подход

Сосредоточимся на проверке, имеет ли граф независимое множество целевого размера k, и не будем вычислять наибольшее независимое множество. (Потом можно использовать линейный или двоичный поиск для идентификации наибольшего значения k, для которого граф имеет независимое множество размера k).

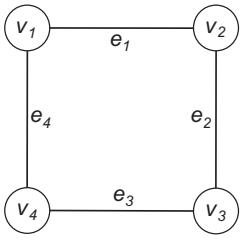
Препроцессор редукции должен превратить граф и целевой размер в то, что ожидает наша подпрограмма для задачи о сумме подмножества — горстку положительных целых чисел.[1] Проще всего задать одно число в расчете на вершину (вместе с целью t), так чтобы независимые множества целевого размера k соответствовали подмножествам чисел, которые в сумме составляют t:



[1] В качестве частного случая задачи о рюкзаке (см. задачу 22.7) задача о сумме подмножества может быть решена динамическим программированием за *псевдополиномиальное* время, то есть за время, полиномиальное по размеру входных данных и магнитуд входных чисел (см. с. 41 и задачу 20.11 (а)). Следовательно, мы должны ожидать, что препроцессор построит экземпляр суммы подмножества с экспоненциально большими числами, и алгоритмы динамического программирования не превзойдут исчерпывающий поиск. NP-трудные задачи, которые поддаются решению за псевдополиномиальное время, называются *слабо NP-трудными*, в то время как *сильно* NP-трудные задачи остаются NP-трудными в случаях, когда все входные числа ограничены полиномиальной функцией от размера входных данных. (NP-трудная задача без чисел во входных данных, такая как задача 3-SAT, автоматически является сильно NP-трудной.) Из девятнадцати задач на рис. 22.1 все, кроме задачи о сумме подмножества и задачи о рюкзаке, являются сильно NP-трудными.

22.8.2. Пример: четырехвершинный цикл

Здесь ключевая идея состоит в использовании каждой из младших цифр числа для кодирования того, является ли ребро инцидентным соответствующей вершине. Например, препроцессор может закодировать вершины четырехвершинного цикла



следующими ниже пятизначными числами (по основанию 10):

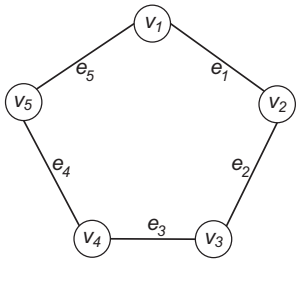
v_1	v_2	v_3	v_4
11 001	11 100	10 110	10 011

Например, последние четыре цифры кодировки v_2 указывают на то, что она примыкает к e_1 и e_2, но не к e_3 или e_4.

Эта идея выглядит многообещающе. Два двучленных независимых множества четырехвершинного цикла, $\{v_1, v_3\}$ и $\{v_2, v_4\}$, соответствуют двум парам чисел с одинаковой суммой: 11 001 + 10 110 = 11 100 + 10 011 = 21 111. Все остальные подмножества имеют разные суммы; например, сумма, соответствующая множеству $\{v_3, v_4\}$, которое не является независимым, равна 10 110 + 10 011 = = 20 121. Постпроцессор может оттранслировать любое подмножество чисел с суммой 21 111 в двучленное независимое множество четырехвершинного цикла.

22.8.3. Пример: пятивершинный цикл

Попытамся проделать тот же маневр с пятивершинным циклом:



При этом каждая вершина (и инцидентные ей ребра) кодируется шестизначным числом. Разные двучленные независимые множества теперь соответствуют парам чисел с разными суммами — например, 211 101 и 211 110 для $\{v_1, v_3\}$ и $\{v_2, v_4\}$. В общем случае цифра нижнего порядка суммы будет равна 0, если она соответствует ребру, не имеющему ни одной конечной точки в независимом множестве, и 1 — в противном случае.

Для того чтобы исправить цифры нижнего порядка, которые в противном случае могут быть равны 0, препроцессор определит одно дополнительное число в расчете на ребро. Для пятивершинного цикла окончательный список чисел таков:

v_1	v_2	v_3	v_4	v_5
110 001	111 000	101 100	101 110	100 011
e_1	e_2	e_3	e_4	e_5
10 000	1000	100	10	1

Теперь целевая сумма $t = 211\ 111$ может быть достигнута путем взятия чисел, соответствующих двучленному независимому множеству (например, $\{v_1, v_3\}$ или $\{v_2, v_4\}$), а также чисел, соответствующих ребрам, не имеющим ни одной конечной точки в независимом множестве (например, соответственно e_4 или e_5). Нет никакого другого способа достичь этой целевой суммы (убедитесь сами).

22.8.4. Доказательство теоремы 22.9

Доказательство теоремы 22.9 масштабирует пример из предыдущего раздела до общих экземпляров независимых множеств.

Описание редукции

Препроцессор. Получив неориентированный граф $G = (V, E)$ с множеством вершин $V = \{v_1, v_2, ..., v_n\}$ и множеством ребер $E = \{e_1, e_2, ..., e_m\}$ и целевой размер k, препроцессор строит $n + m + 1$ положительных целых чисел, которые определяют экземпляр задачи о сумме подмножества:

- Для каждой вершины v_i определить число $a_i = 10^m + \sum_{e_j \in A_i} 10^{m-j}$, где A_i обозначает ребра, инцидентные v_i. (Написанная с основанием 10, начальная цифра равна 1, и j-я цифра после этого равна 1, если e_j инцидентно v_i, и 0 — в противном случае.)
- Для каждого ребра e_j определить число $b_j = 10^{m-j}$.
- Определить целевую сумму $t = k \times 10^m + \sum_{j=1}^{m} 10^{m-j}$. (Написанные с основанием 10 цифры числа k с последующими m единицами.)

Постпроцессор. Если предполагаемая подпрограмма вычисляет подмножество $\{a_1, a_2, ..., a_n, b_1, b_2, ..., b_m\}$ с суммой t, то постпроцессор возвращает вершины v_i, соответствующие a_i в подмножестве. (Например, при вручении подмножества $\{a_2, a_4, a_7, b_3, b_6\}$ постпроцессор возвращает подмножество вершин $\{v_2, v_4, v_7\}$.) Если предполагаемая подпрограмма отвечает «решения нет», то постпроцессор тоже отвечает «решения нет».

Внешний цикл. Препроцессор и постпроцессор предназначены для проверки наличия независимого множества целевого размера k. Чтобы вычислить наибольшее независимое множество входного графа $G = (V, E)$, редукция проверяет все возможные значения $k = n, n - 1, n - 2, ..., 2, 1$:

1. Вызывает препроцессор для преобразования G и текущего значения k в экземпляр задачи о сумме подмножества.
2. Вызывает подпрограмму для задачи о сумме подмножества.
3. Вызывает постпроцессор на выходных данных подпрограммы. Если тот возвращает k-размерное независимое множество S графа G, то внешний цикл остановливает выполнение и возвращает S.

В целом редукция вызывает подпрограмму вычисления суммы подмножества не более n раз и выполняет объем дополнительной работы не выше полиномиального.

Доказательство правильности

Редукция правильна, при условии что каждая итерация внешнего цикла верно определяет для текущего значения k наличие во входном графе k-размерного независимого множества.

Случай 1: *k*-размерное независимое множество отсутствует. Предположим, что итерация внешнего цикла редукции не возвращает «решения нет» для некоторого графа $G = (V, E)$, который не имеет k-размерного независимого множества. Тогда подпрограмма возвращает подмножество N чисел $\{a_1, a_2, ..., a_n, b_1, b_2, ..., b_m\}$, построенное препроцессором с целевой суммой t. Обозначим через $S \subseteq V$ вершины, соответствующие элементам a_i в N. Чтобы получить противоречие, примем S за k-размерное независимое множество графа G.

В общем случае для каждого подмножества s из элементов a_i и любого числа b_j сумма (записанная с основанием 10) представляет собой цифры из s, за которыми следуют m цифр, каждая из которых принадлежит {0, 1, 2, 3}. (Ровно три числа могут внести свой вклад в j-ю из m конечных цифр: b_j и два a_i, которые соответствуют конечным точкам e_j.) Поскольку начальные цифры целевой суммы t совпадают с цифрами k, подмножество N должно содержать k из a_i. Таким образом, S имеет размер k. Поскольку m конечных цифр суммы t равны 1, подмножество N не может содержать два a_i, соответствующих конечным точкам общего ребра, и S является независимым множеством графа G.

Случай 2: существует хотя бы одно *k*-размерное независимое множество. Предположим, что входной граф $G = (V, E)$ имеет k-размерное независимое множество S. Тогда экземпляр суммы подмножества, построенной препроцессором, имеет подмножество с целевой суммой t. Нужно выбрать k элементов a_i, соответствующих вершинам S, а также b_j, соответствующих ребрам без конечных точкек в S. При наличии хотя бы одного допустимого решения подпрограмма должна вернуть подмножество N с суммой t. Как и в случае 1, постпроцессор извлечет из N и вернет k-размерное независимое множество графа G. *Ч. Т. Д.*

ВЫВОДЫ

- Чтобы доказать, что задача B является NP-трудной, следует выполнить двухшаговый рецепт: (1) выбрать NP-трудную задачу A; (2) доказать, что A сводится к B.
- Задача 3-SAT является частным случаем задачи SAT, в которой каждое ограничение имеет не более трех литералов.

- ✶ Теорема Кука — Левина доказывает, что задача 3-SAT является NP-трудной.
- ✶ Тысячи использований двухшагового рецепта доказали, что тысячи задач являются NP-трудными, начиная с задачи 3-SAT.
- ✶ Редукции в доказательствах NP-трудности соответствуют шаблону: предобработка входных данных; вызов подпрограммы; постобработка выходных данных.
- ✶ В задаче о независимом множестве на вход подается неориентированный граф для вычисления наибольшего подмножества несмежных вершин.
- ✶ Задача 3-SAT сводится к задаче о независимом множестве, в доказательство того что последняя является NP-трудной.
- ✶ *s*-*t*-вершинный гамильтонов путь графа *G* начинается в вершине *s*, заканчивается в вершине *t* и посещает каждую вершину графа *G* ровно один раз.
- ✶ Задача 3-SAT сводится к ориентированной версии задачи о гамильтоновом пути, в доказательство того что последняя является NP-трудной.
- ✶ Неориентированная версия задачи о гамильтоновом пути сводится к задаче коммивояжера, в доказательство того что последняя является NP-трудной.
- ✶ В задаче о сумме подмножества нужно составить подмножество заданного множества положительных целых чисел с суммой, равной целевой (либо делается вывод, что его не существует).
- ✶ Задача о независимом множестве сводится к задаче о сумме подмножества, в доказательство того что последняя является NP-трудной.

Задачи на закрепление материала

Задача 22.1 *(S)*. Если предположение, что $P \neq NP$, является истинным, какая из следующих задач может быть решена за полиномиальное время? (Правильных ответов может быть несколько.)

а) Взяв связный неориентированный граф, вычислить остовное дерево с наименьшим возможным числом листьев.

б) Взяв связный неориентированный граф, вычислить остовное дерево с минимально возможной максимальной степенью. (Степень вершины — это число инцидентных ребер.)

в) Взяв связный неориентированный граф с неотрицательными длинами ребер, стартовой вершиной s и финальной вершиной t, вычислить минимальную длину бесциклового s-t-вершинного пути ровно с $n - 1$ ребрами (либо $+\infty$, если такого пути не существует).

г) Взяв связный неориентированный граф с неотрицательными длинами ребер, стартовой вершиной s и финальной вершиной t, вычислить минимальную длину (не обязательно бесциклового) s-t-вершинного пути ровно с $n - 1$ ребрами (либо $+\infty$, если такого пути не существует).

Задача 22.2 *(S)*. Если предположение, что $P \neq NP$, является истинным, какая из следующих задач может быть решена за полиномиальное время? (Правильных ответов может быть несколько.)

а) Взяв ориентированный граф $G = (V, E)$ с неотрицательными длинами ребер, вычислить наибольшую длину кратчайшего пути между любой парой вершин (то есть $\max_{v,w \in V} dist(v, w)$, где $dist(v, w)$ обозначает расстояние кратчайшего пути из вершины v в вершину w).

б) Взяв ориентированный ациклический граф с вещественнозначными длинами ребер, вычислить наибольшую длину пути между любой парой вершин.

в) Взяв ориентированный граф $G = (V, E)$ с неотрицательными длинами ребер, вычислить наибольшую длину бесциклового пути между любой парой вершин (то есть $\max_{v,w \in V} maxlen(v, w)$, где $maxlen(v, w)$ обозначает наибольшую длину бесциклового пути из v в w).

г) Взяв ориентированный граф с вещественнозначными длинами ребер, вычислить наибольшую длину бесциклового пути между любой парой вершин.

Задача 22.3 *(S)*. Назовем версию задачи об ориентированном гамильтоновом пути на с. 52, в которой требуется только ответ «да» или «нет», *решательной*

версией. Назовем версию на с. 201, в которой требуется сам *s*-*t*-вершинный гамильтонов путь (всякий раз, когда он существует), *поисковой версией*. Назовем версию задачи коммивояжера в разделе 19.1.2 *оптимизационной версией* и определим *поисковую версию* задачи коммивояжера следующим образом: имея полный граф, вещественнозначные реберные стоимости и целевую стоимость C, вернуть тур коммивояжера с суммарной стоимостью, не превышающей C (либо правильно объявить, что его не существует).

Какое утверждение истинно? (Правильных ответов может быть несколько.)

а) Решательная версия задачи об ориентированном гамильтоновом пути сводится к поисковой версии.

б) Поисковая версия задачи об ориентированном гамильтоновом пути сводится к решательной версии.

в) Поисковая версия задачи коммивояжера сводится к оптимизационной версии.

г) Оптимизационная версия задачи коммивояжера сводится к поисковой версии.

Задача 22.4 *(H)*. В задаче о клике на вход подается неориентированный граф для вывода клики — подмножества взаимно смежных вершин — с максимально возможным размером. Докажите, что задача о независимом множестве сводится к задаче о клике, подразумевая (в силу следствия 22.3), что последняя является NP-трудной.

Задача 22.5 *(H)*. В задаче об охвате вершин на вход подается неориентированный граф $G = (V, E)$ для идентификации наименьшего подмножества $S \subseteq V$ вершин, включающего хотя бы одну конечную точку каждого ребра в E. Докажите, что задача о независимом множестве сводится к задаче об охвате вершин, подразумевая (в силу следствия 22.3), что последняя является NP-трудной.

Задача 22.6 *(H)*. В задаче о покрытии множества входные данные содержат m подмножеств $A_1, A_2, ..., A_m$ универсального множества U для идентификации наименьшей коллекции подмножеств, объединение которых равно U. Докажите, что задача об охвате вершин сводится к задаче о покрытии множества, подразумевая (в силу задачи 22.5), что последняя является NP-трудной.

Задача 22.7 *(H)*. Докажите, что задача о сумме подмножества сводится к задаче о рюкзаке (с. 41), подразумевая (в силу следствия 22.10), что последняя является NP-трудной.

Задачи повышенной сложности

Задача 22.8 *(S)*. Докажите, что задача о покрытии множества сводится к задаче о максимальном охвате (раздел 20.2.1), подразумевая (в силу задачи 22.6), что последняя является NP-трудной.

Задача 22.9 *(H)*. Докажите, что задача о неориентированном гамильтоновом пути сводится к задаче об ориентированном гамильтоновом пути и наоборот. (В частности, следствие 22.6 вытекает из следствия 22.5.)

Задача 22.10 *(H)*.

а) Докажите, что задача о сумме подмножества остается NP-трудной в частном случае, когда целевая сумма равна половине суммы входных чисел (то есть $t = \frac{1}{2}\sum_{i=1}^{n} a_i$).[1]

б) Докажите, что этот частный случай сводится к задаче о минимизации производственной продолжительности с двумя машинами, подразумевая (в силу (а)), что последняя является NP-трудной.[2]

Задача 22.11 *(H)*. Докажите, что задача 3-SAT сводится к частному случаю задачи о раскраске графа (с. 168), в которой число k разрешимых цветов равно 3, подразумевая (в силу теоремы Кука — Левина), что последняя является NP-трудной.[3]

---

[1] Этот частный случай задачи о сумме подмножества часто именуется задачей о *подразделении* (partition problem).

[2] Двухмашинный частный случай задачи о минимизации производственной продолжительности может быть решен за псевдополиномиальное время динамическим программированием (убедитесь сами) и, следовательно, является лишь слабо NP-трудным (см. сноску на с. 179). Более сложная редукция показывает, что общая версия задачи является сильно NP-трудной.

[3] Задача о раскраске графа может быть решена за линейное время при $k = 2$ (см. сноску на с. 179).

Задача 22.12 *(H)*. Задача 20.12 содержит метрический частный случай задачи коммивояжера, в котором реберные стоимости c входного графа $G = (V, E)$ являются неотрицательными и удовлетворяют неравенству треугольника:

$$c_{vw} \leq \sum_{e \in P} c_e$$

для каждой пары $v, w \in V$ вершин и v-w-вершинного пути P в G. Задача 20.12 показывает эвристический полиномиально-временной алгоритм, который, получая метрический экземпляр задачи коммивояжера, гарантированно возвращает тур с суммарной стоимостью не более чем в два раза больше минимально возможной. Можем ли мы добиться лучшего и точно решить метрический частный случай либо расширить гарантию приближенной правильности эвристического алгоритма до уровня гарантии общей задачи коммивояжера?

а) Доказать, что метрический частный случай задачи коммивояжера является NP-трудным.

б) Если предположение, что P ≠ NP, является истинным, докажите, что не существует полиномиально-временного алгоритма, который для каждого экземпляра задачи коммивояжера с неотрицательными реберными стоимостями (и без других допущений) возвращает тур с суммарной стоимостью не более чем в 10^{100} раз больше минимально возможной.

23 P, NP и все такое прочее

Главы 19–22 охватывают все, что нужно знать разработчику алгоритмов об NP-трудных задачах: алгоритмические последствия NP-трудности, алгоритмические инструменты достижения прогресса в решении NP-трудных задач и способы распознавания NP-трудных задач в дикой природе. Мы дали предварительное определение понятию NP-трудности в контексте предположения, что $P \neq NP$, описанного в разделе 19.3.5 без формальных математических определений. Эта необязательная глава заполняет недостающие основы.[1]

В разделе 23.1 изложен план накопления свидетельств труднорешаемости задачи с помощью сведения к ней большого числа задач. В разделе 23.2 выделяются три типа вычислительных задач: задачи принятия решения, поиска и оптимизации. Раздел 23.3 определяет класс сложности $\mathcal{NP}$ как множество

[1] Эта глава представляет собой введение в прекрасную и математически глубокую область, именуемую *теорией вычислительной сложности*, которая изучает количество вычислительных ресурсов (таких как время, память или вероятность), необходимых для решения разных вычислительных задач (в зависимости от размера входных данных). Мы будем фокусироваться на алгоритмических следствиях этой теории, что приведет к несколько нетрадиционному подходу. Если вы хотите узнать о теории вычислительной сложности больше, то я рекомендую начать с превосходных (и свободно доступных) видеолекций Райана О'Доннелла (http://www.cs.cmu.edu/~odonnell/).

всех поисковых задач с распознаваемыми решениями, формально определяет NP-трудные задачи и пересматривает теорему Кука — Левина. Раздел 23.4 формально определяет предположение, что P ≠ NP, и рассматривает его текущее состояние. В разделе 23.5 описаны две гипотезы, по силе превосходящие предположение, что P ≠ NP, — гипотеза об экспоненциальном времени (ETH) и сильная гипотеза об экспоненциальном времени (SETH) и их алгоритмические следствия (например, для задачи о выравнивании рядов). Раздел 23.6 завершается обсуждением редукций Левина и NP-полных задач — универсальных задач, способных кодировать все задачи с распознаваемыми решениями.

*23.1. Накопление свидетельств труднорешаемости

В 1967 году Джек Эдмондс выдвинул предположение, что задача коммивояжера не решается никаким полиномиально-временным алгоритмом, даже с временем выполнения $O(n^{100})$ или $O(n^{10\,000})$ для входных данных с n вершинами (с. 24). Каким образом без математического доказательства можно было бы построить убедительный случай, показывающий, что эта гипотеза является истинной? Неудавшиеся попытки придумать такой алгоритм столь многими умами за последние семьдесят лет являются косвенным свидетельством труднорешаемости, но есть ли другие ее подтверждения?

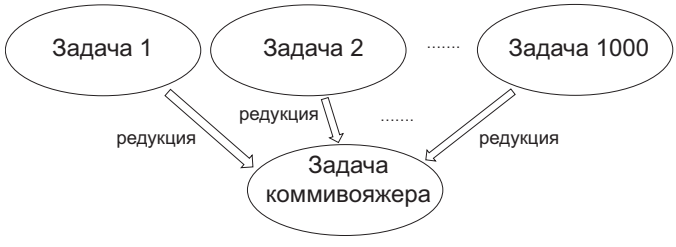
23.1.1. Построение убедительного случая с помощью редукций

Необходимо показать, что полиномиально-временной алгоритм для задачи коммивояжера решит не одну, а *тысячи* нерешенных задач.

СБОР ДОКАЗАТЕЛЬСТВ ТОГО, ЧТО ЗАДАЧА КОММИВОЯЖЕРА ТРУДНОРЕШАЕМА

1. Выбрать действительно большую коллекцию C вычислительных задач.

2. Доказать, что *каждая* задача в *C* сводится к задаче коммивояжера.[1]



Тогда полиномиально-временной алгоритм для задачи коммивояжера автоматически подойдет каждой задаче в множестве *C*. Выражаясь иначе, если даже *одну* задачу в множестве *C* невозможно решить полиномиально-временным алгоритмом, то и задачу коммивояжера тоже. Чем больше множество *C*, тем вероятнее, что задача коммивояжера не подается решению за полиномиальное время.

23.1.2. Выборка множества C для задачи коммивояжера

Выступая против полиномиально-временной решаемости задачи коммивояжера, почему бы не принять *C* как множество *всех* вычислительных задач в мире? Потому что это слишком амбициозно. Какой бы трудной ни была задача коммивояжера, существуют вычислительные задачи намного труднее. Наконец, *неразрешимые* задачи не могут быть решены компьютером за конечный промежуток времени (даже за экспоненциальное время, даже за

[1] Согласно разделам 19.5.1 и 22.1, редукция из задачи *A* к задаче *B* — это алгоритм, который решает задачу *A*, используя самое большее полиномиальное (по размеру входных данных) число вызовов подпрограммы, решающей задачу *B*, и полиномиальный объем дополнительной работы. Этот тип редукции иногда именуется *редукцией Кука* (в честь Стивена Кука) или *полиномиально-временной редукцией Тьюринга* (в честь Алана Тьюринга) и является наиболее разумным, на котором следует сосредоточиться при изучении алгоритмов. Более ограниченные типы редукций важны для определения NP-полных задач (см. раздел 23.6).

дважды экспоненциальное время и т. д.). Одним из известных примеров неразрешимой задачи является задача об *остановке*: имея программу (скажем, тысячу строк кода на Python), определить, входит ли она в бесконечный цикл или в конечном итоге останавливается. Очевидный подход — выполнить программу и посмотреть, что она делает. Но если программа не остановилась через столетие, как вы узнаете, находится ли она в бесконечном цикле или остановится завтра? Вы можете надеяться на какой-то обходной путь поумнее, чем механическая симуляция кода, но, к сожалению, его не существует.[1]

Задача коммивояжера больше не выглядит так плохо — по меньшей мере она может быть решена за конечное (хотя и экспоненциальное) количество времени с помощью исчерпывающего поиска. Задача об остановке никоим образом не может быть сведена к задаче коммивояжера, так как такое сведение переведет экспоненциально-временной алгоритм для задачи коммивояжера в алгоритм для решения задачи об остановке (которого, согласно Тьюрингу, не существует).

Итак, каково наибольшее множество вычислительных задач C, которое можно свести к задаче коммивояжера? Интуитивно, это множество *всех задач, поддающихся решению аналогичным алгоритмом исчерпывающего поиска*, — тогда среди них задача коммивояжера будет самой трудной. Есть ли математическое определение, отражающее эту идею?

---

[1] В 1936 году Алан М. Тьюринг опубликовал свою работу «О вычислимых числах, с применением к проблеме разрешимости» («*On Computable Numbers, with an Application to the Entscheidungsproblem», Proceedings of the London Mathematical Society*, 1936). Я и многие другие ученые-компьютерщики рассматриваем эту работу как рождение нашей дисциплины и по этой причине считаем, что имя Тьюринга должно быть так же широко признано, как, скажем, имя Альберта Эйнштейна.

Что сделало этот документ таким важным? Две вещи. Во-первых, Тьюринг представил формальную модель того, что могут делать компьютеры общего назначения, теперь называемые машиной Тьюринга. (Имейте в виду, это было за десять лет до того, как кто-либо действительно создал универсальный компьютер!) Во-вторых, определение того, что компьютеры могут делать, позволило Тьюрингу изучить то, что они не могут сделать, и доказать, что проблема остановки неразрешима. Таким образом, буквально с первого дня ученые остро осознавали ограничения компьютеров и необходимость компромисса при решении сложных вычислительных задач.

*23.2. Решение, поиск и оптимизация

Прежде чем определить множество «задач, поддающихся решению наивным исчерпывающим поиском» — класс сложности $\mathcal{NP}$ — давайте сделаем шаг назад и упорядочим по категориям разные типы форматов входных-выходных данных, которые мы видели.

ТРИ ТИПА ВЫЧИСЛИТЕЛЬНЫХ ЗАДАЧ

1. Задача принятия решения. Выдать «да», если есть допустимое решение, или «нет» в противном случае.
2. Задача поиска. Выдать допустимое решение, если оно существует, или «решения нет» в противном случае.
3. Задача оптимизации. Выдать допустимое решение с наилучшим возможным значением целевой функции (либо «решения нет», если такового не существует).

На практике задачи принятия решения встречаются реже других, и в этой книге мы увидели только один их пример: оригинальное описание задачи об ориентированном гамильтоновом пути (с. 52), в которой допустимые решения соответствуют *s*-*t*-вершинным гамильтоновым путям. Задачи поиска и оптимизации являются общими. Из девятнадцати задач в главе 22 (рис. 22.1) шесть являются задачами поиска: задачи 3-SAT, SAT, о раскраске графа, об ориентированном гамильтоновом пути (версия на с. 201), о неориентированном гамильтоновом пути и о сумме подмножества. Остальные тринадцать — это задачи оптимизации.[1] Определение понятия допустимого решения — например, удовлетворяющее присвоение истинности, гамильтонов путь или тур коммивояжера — специфично для конкретной задачи. Для задач оптимизации целевая функция — например, минимизация или максимизация суммарной стоимости — также специфична для конкретной задачи.

[1] Чтобы включить задачу бесциклового кратчайшего пути в определение задачи оптимизации, рассмотрим вариант, в котором на вход подаются две вершины, и на выходе требуется кратчайший бесцикловый путь из первой во вторую. Доказательство NP-трудности в разделе 19.5.4 (лемма 19.2) также применимо к этой версии задачи.

Классы сложности удобно рассматривать в рамках задач одного типа, чтобы избежать ошибок проверки типов, поэтому мы ограничим определение NP задачами поиска.[1] Не беспокойтесь, что задачи оптимизации, такие как задача коммивояжера, останутся за бортом: каждая задача оптимизации имеет поисковую версию. Входные данные поисковой версии включают значение *t* целевой функции: не менее *t* (для задач максимизации) или не более *t* (для задач минимизации), либо объявление, что решения нет. Как мы увидим, поисковые версии почти всех изученных нами задач оптимизации относятся к NP.[2]

*23.3. NP: задачи с легко распознаваемыми решениями

Подойдем к сути обсуждения. Как задать множество задач, «поддающихся решению исчерпывающим поиском», которые могут правдоподобно сводиться к задаче коммивояжера? Что нужно для решения задачи наивным исчерпывающим поиском?

23.3.1. Определение класса сложности NP

Главная мысль, лежащая в основе класса сложности NP, заключается в *эффективном распознавании предполагаемых решений*. Если кто-то вручил вам

---

[1] В большинстве книг класс сложности NP определяется в терминах задач принятия решения. Это удобно для развития теории сложности, но отдаляет от естественных алгоритмических задач. Версия используемого здесь класса иногда называется FNP, где F означает «функциональный». Все алгоритмические последствия NP-трудности, включая истинность или ложность предположения, что $P \neq NP$, остаются неизменными независимо от того, какое определение используется.

[2] Поисковая версия задачи оптимизации напрямую сводится к исходной версии: либо оптимальное решение удовлетворяет заданному значению *t* целевой функции, либо никаких допустимых решений нет. Еще интереснее обратное: типичная задача оптимизации сводится к ее поисковой версии посредством двоичного поиска над целью *t* (см. также задачу 22.3). Такая задача оптимизации поддается решению за полиномиальное время, если и только если ее поисковая версия поддается решению за полиномиальное время, и схожим образом является NP-трудной, если и только если ее поисковая версия является NP-трудной.

якобы допустимое решение задачи на блюдечке с голубой каемочкой, вы сможете быстро проверить, действительно ли оно является допустимым. Легко узнать, соблюдены ли все правила в заполненной головоломке судоку или кенкен. Так же просто проверить, составляют ли тур коммивояжера готовые вершины в графе и, если да, не превышает ли стоимость тура заданную цель t.[1]

КЛАСС СЛОЖНОСТИ NP

Задача поиска относится к классу сложности NP, если и только если:

1. Для каждого экземпляра каждое кандидатное решение имеет длину описания (скажем, в битах), ограниченную сверху полиномиальной функцией от размера входных данных.
2. Для каждого экземпляра и кандидатного решения предполагаемая допустимость решения может быть подтверждена или опровергнута за время, полиномиальное по размеру входных данных.

23.3.2. Примеры задач в NP

Вступительные требования для членства в NP столь легко соблюсти, что почти все задачи поиска, которые вы видели, им соответствуют. Например, поисковая версия задачи коммивояжера относится к классу NP: тур из n вершин может быть описан с помощью $O(n \log n)$ бит — примерно $\log_2 n$ бит для именования каждой вершины, — и по списку вершин можно проверить, составляют ли они тур с суммарной стоимостью, не превышающей заданную цель t. Задача 3-SAT (раздел 22.2) также относится к NP: описание присвоения истинности n булевым переменным занимает n бит, и проверка, удовлетворяет ли она каждому заданному ограничению, выполняется прямолинейно. Так же

[1] NP можно описать как задачи поиска, которые поддаются эффективному решению в фиктивной вычислительной модели, определенной «недетерминированными машинами Тьюринга». Аббревиатура NP означает «недетерминированное полиномиальное время» (а вовсе не «не полиномиальное»!). В контексте алгоритмов размышляйте об NP как о задачах с легко распознаваемыми решениями.

легко проверить, является ли предлагаемый путь гамильтоновым, имеет ли предложенный план работ данную производственную продолжительность или является ли предлагаемое подмножество вершин независимым множеством, вершинным покрытием или кликой заданного размера.

ТЕСТОВОЕ ЗАДАНИЕ 23.1

Из девятнадцати задач, перечисленных на рис. 22.1, для скольких поисковая версия принадлежит NP?

а) 16

б) 17

в) 18

г) 19

(Ответ и анализ решения см. в разделе 23.3.6.)

23.3.3. Задачи NP поддаются решению исчерпывающим поиском

Изначально мы хотели определить множество задач, поддающихся решению наивным исчерпывающим поиском, чтобы свести их к задаче коммивояжера, но вместо этого определили класс NP как задачи поиска с легко распознаваемыми решениями. В чем связь? Каждая задача в NP может быть решена за экспоненциальное время через поочередную проверку кандидатных решений наивным исчерпывающим поиском:

ИСЧЕРПЫВАЮЩИЙ ПОИСК ДЛЯ ГЕНЕРИЧЕСКОЙ ЗАДАЧИ NP

1. Перечислить кандидатные решения одно за другим:
   а) если текущий кандидат допустим, вернуть его.
2. Вернуть «решения нет».

Для задачи в NP описание кандидатных решений требует $O(n^d)$ бит, где n обозначает размер входных данных, а d — константу (независимую от n). Поэтому число возможных кандидатов (и следовательно, итераций цикла) равно $2^{O(n^d)}$.[1] В силу второго определяющего свойства NP-задачи каждая итерация цикла может быть выполнена за полиномиальное время.

Следовательно, исчерпывающий поиск правильно решает задачу за время, исключительно экспоненциальное по размеру n входных данных.

23.3.4. NP-трудные задачи

Требования к членству в классе сложности NP крайне слабы и опираются на нашу способность распознавать правильные решения. NP — это обширный класс задач поиска, с которыми вы с большой вероятностью встретитесь. Поэтому, если *каждая* задача в NP сводится к задаче A, которая является как минимум такой же трудной, как и каждая задача в NP, полиномиально-временной алгоритм для A приведет к появлению таких же алгоритмов для всех задач NP. Это убедительное свидетельство внутренне присущей труднорешаемости и формальное определение *NP-трудной* задачи.

NP-ТРУДНАЯ ЗАДАЧА (ФОРМАЛЬНОЕ ОПРЕДЕЛЕНИЕ)

Вычислительная задача является *NP-трудной*, если каждая задача в NP сводится к ней.

После того как в разделе 23.4 мы формально определим предположение, что P ≠ NP, вы увидите, что каждая задача, которая является NP-трудной в соответствии с этим определением, также удовлетворяет и предыдущему определению (раздел 19.3.7), используемому в главах 19–22. По-прежнему теорема Кука — Левина (теорема 22.1) показывает, что задача 3-SAT яв-

[1] Обозначение «O большое» в экспоненте подавляет константные факторы (и члены более низкого порядка). Например, функция $T(n)$ равна $2^{O(\sqrt{n})}$, если существуют константы c, $n_0 > 0$ такие, что $T(n) \leq 2^{c\sqrt{n}}$ для всех $n \geq n_0$. (Тогда как $T(n) = O(2^{\sqrt{n}})$ означает, что $T(n) \leq c \times 2^{\sqrt{n}}$ для всех $n \geq n_0$.)

ляется NP-трудной (раздел 23.3.5), редукции продолжают распространять NP-трудность (задача 23.4), и, как следствие, девятнадцать задач из главы 22 остаются NP-трудными.[1]

23.3.5. Теорема Кука — Левина повторно

В главе 22 мы приняли теорему Кука — Левина (теорема 22.1) на веру и объединили ее с двухшаговым рецептом для доказательства NP-трудности. Согласно этой теореме, каждая задача в NP сводится к задаче 3-SAT. Как такое может быть? Задача 3-SAT кажется такой простой, а класс NP — таким обширным.

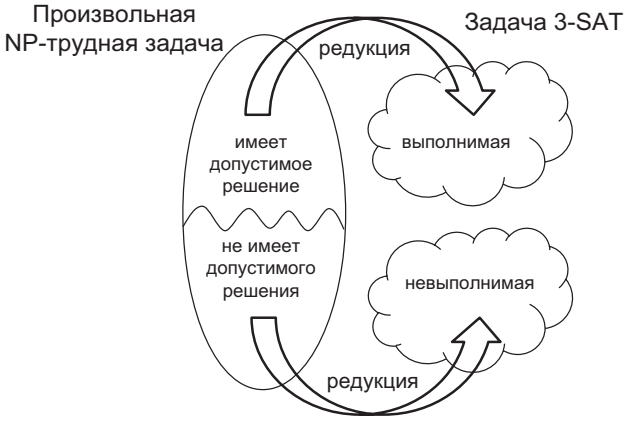
Детали доказательства имеют запутанную форму, но вот его суть:[2] зафиксировать произвольную задачу A класса сложности NP, чтобы показать, что A сводится к задаче 3-SAT. Об A известно, что она удовлетворяет двум определяющим требованиям задачи NP: (1) допустимые решения для n-размерных экземпляров могут быть описаны с использованием не более $c_1 n^{d_1}$ бит и (2) предполагаемые допустимые решения n-размерных экземпляров могут быть проверены за время, не превышающее $c_2 n^{d_2}$ (где c_1, c_2, d_1, d_2 — это константы). Обозначим через $\texttt{verify}_A$ алгоритм в (2), который проверяет допустимость предполагаемого решения.

Обратимся к простейшей редукции (раздел 22.4). Препроцессор транслирует экземпляры A с допустимым решением и без него соответственно в выполнимые и невыполнимые экземпляры задачи 3-SAT:

---

[1] Обратившись к другим книгам, вы часто увидите более требовательное определение NP-трудности, которое требует редукций очень специфической формы, именуемых редукциями Левина. (Раздел 23.6.1 определяет такие редукции, а раздел 23.6.2 использует их для определения NP-полных задач.) В соответствии с этим более строгим определением только задачи поиска имеют право на NP-трудность; вместо того чтобы говорить, что задача коммивояжера является NP-трудной, нужно говорить: поисковая версия задачи коммивояжера является NP-трудной. Используемое здесь более либеральное определение с общими редукциями (Кука) лучше согласуется с алгоритмической точкой зрения книг этой серии.

[2] Полное доказательство см. в любом учебнике по вычислительной сложности.



Препроцессор. Получив n-размерный экземпляр I_A, препроцессор строит экземпляр I_{3SAT}:

- Определите $c_1 n^{d_1}$ переменных решения, чтобы они записывали биты, описывающие кандидатное решение для I_A.
- Определите $(c_2 n^{d_2})^2 = c_2^2 \times n^{2d_2}$ переменных состояния, чтобы они кодировали исполнение $\texttt{verify}_A$ на кандидатном решении для I_A, кодированном переменными решения.
- Определите ограничения для обеспечения семантики переменных состояния. Типичное ограничение утверждает: j-й бит памяти после шага $i + 1$ согласуется с соответствующим содержимым памяти после шага i, данным экземпляром I_A, кандидатным решением для I_A, кодированным переменными решения, и кодом алгоритма $\texttt{verify}_A$.
- Определите ограничения, которые обеспечивают, чтобы вычисление, кодированное переменными состояния, завершалось утверждением допустимости кандидатного решения, кодированного переменными решения.

Почему именно $(c_2 n^{d_2})^2$ переменных состояния? Алгоритм $\texttt{verify}_A$ выполняет не более $c_2 n^{d_2}$ примитивных операций и, исходя из вычислительной модели, в которой один бит памяти может быть доступен в расчете на операцию, ссылается не более чем на $c_2 n^{d_2}$ бит памяти. Значит, все его вычисления могут

быть подытожены (более или менее) в таблице размера $c_2 n^{d_2} \times c_2 n^{d_2}$, строки которой соответствуют шагам i, а столбцы — битам j памяти. Тогда каждая переменная состояния кодирует содержимое одного бита памяти в одной точке вычисления.[1]

Ограничения по согласованности внешне выглядят сложно. Но, поскольку один шаг алгоритма (например, машины Тьюринга) достаточно прост, то каждое из логических ограничений можно реализовать с малым числом трехлитеральных дизъюнкций (причем детали будут зависеть от вычислительной модели). Конечным результатом будет экземпляр задачи I_{3SAT} с полиномиальным (в n) числом переменных и ограничений.

Постпроцессор. Если предполагаемая подпрограмма возвращает удовлетворяющее присвоение истинности для построенного препроцессором экземпляра задачи I_{3SAT}, то постпроцессор возвращает кандидатное решение для I_A, кодированное присвоениями значений переменным решения. Если предполагаемая подпрограмма отвечает, что I_{3SAT} не выполнима, то постпроцессор сообщает, что у I_A нет допустимых решений.

Набросок правильности. Ограничения экземпляра I_{3SAT} определяются так, чтобы удовлетворяющие присвоения истинности соответствовали допустимым решениям для экземпляра I_A (кодируемого переменными решения) наряду с поддерживающей верификацией, выполняемой алгоритмом $\texttt{verify}_A$ (который кодируется переменными состояния). Поэтому если I_A не имеет допустимых решений, то экземпляр I_{3SAT} не должен быть выполнимым. И наоборот, если у I_A есть допустимое решение, то I_{3SAT} должен иметь удовлетворимое присвоение истинности. Предполагаемая подпрограмма задачи 3-SAT должна вычислить такое присвоение, которое будет конвертировано постпроцессором в допустимое решение для I_A.

---

[1] Здесь есть дополнительные детали, которые зависят от используемой точной вычислительной модели и определения «примитивной операции». Самый простой подход заключается в использовании машины Тьюринга (см. сноску на с. 225): на каждом шаге требуется еще один пакет булевых переменных для кодирования текущего внутреннего состояния машины. Доказательство можно заставить работать для любой разумной модели вычисления.

23.3.6. Решение к тестовому заданию 23.1

Правильный ответ: (в). Исключение? Задача максимизации влияния. В отличие от вычисления суммарной стоимости тура или производственной продолжительности плана, которое является прямым, влияние подмножества k вершин определяется как математическое ожидание с экспоненциально большим числом членов (см. (20.10) и решение задачи 20.6). Поскольку непонятно, как оценивать целевую функцию в задаче максимизации влияния за полиномиальное время, поисковая версия задачи явно не принадлежит NP.

*23.4. Предположение, что P ≠ NP

В разделе 19.3.5 мы неформально определили предположение, что P ≠ NP: фундаментально проще проверить предполагаемое решение задачи, чем создать собственное решение с нуля. Сформулируем эту гипотезу формально.

23.4.1. P: задачи NP, поддающиеся решению за полиномиальное время

Некоторые задачи в NP могут быть решены за полиномиальное время, например задача 2-SAT (задача 21.12) и поисковая версия задачи о минимальном остовном дереве (раздел 19.1.1). Класс сложности P определяется как множество таких задач.

> **КЛАСС СЛОЖНОСТИ P**
>
> Поисковая задача относится к классу сложности P, если и только если она принадлежит NP и может быть решена с помощью полиномиально-временного алгоритма.

По определению, каждая задача в P принадлежит NP:

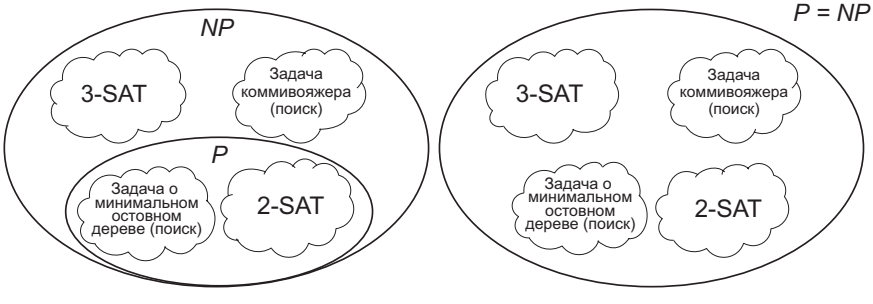
$$P \subseteq NP.$$

23.4.2. Формальное определение предположения

Теперь нетрудно угадать, как звучит формальное определение предположения о P ≠ NP, а именно, что P является строгим подмножеством NP:

ПРЕДПОЛОЖЕНИЕ, ЧТО P ≠ NP (ФОРМАЛЬНАЯ ВЕРСИЯ)

$P \subsetneq NP$.



а) Мир в случае, если *P ≠ NP* б) Мир в случае, если *P = NP*

Предположение, что P ≠ NP, утверждает существование поисковой задачи с легко распознаваемыми решениями (задачи NP), которая не может быть решена полиномиально-временным алгоритмом, — задачи, для которой легко проверить предполагаемое допустимое решение, но трудно придумать собственное с нуля. Если указанное предположение является ложным, то P = NP, и эффективное распознавание допустимых решений автоматически приведет к эффективному вычислению допустимых решений (как бы они ни были далеки).

Полиномиально-временной алгоритм для NP-трудной задачи *A* будет непосредственно приводить к решению каждой задачи NP, причем каждая задача в NP будет сводиться к *A*, а легкорешаемость будет распространяться от *A* ко всем NP в подтверждение, что P = NP. Таким образом, предварительное определение NP-трудной задачи в разделе 19.3.7 является логическим следствием формального определения на с. 230:

СЛЕДСТВИЕ NP-ТРУДНОСТИ

Если предположение, что $P \neq NP$, является истинным, то никакая NP-трудная задача не может быть решена полиномиально-временным алгоритмом.

23.4.3. Статус предположения, что $P \neq NP$

Предположение, что $P \neq NP$, возможно, является самым важным открытым вопросом в computer science и математике. Например, решение указанного предположения является одной из семи «задач тысячелетия», предложенных в 2000 году Математическим институтом Клэя: решите одну из этих задач, и получите приз в размере миллиона долларов США.[1, 2]

Большинство экспертов считают, что предположение, что $P \neq NP$, является истинным. (Хотя легендарный логик Курт Гедель в письме 1956 года к еще более легендарному Джону фон Нейману высказал предположение, эквивалентное $P = NP$.) Почему? Во-первых, люди очень искусны в обнаружении быстрых алгоритмов. Если бы действительно каждая задача в NP решалась быстрым алгоритмом, почему его до сих пор не открыли? Между тем доказательства, очерчивающие границы алгоритмов, были немногочисленны и редки, ведь $P \subsetneq NP$.

Во-вторых, как примирить $P = NP$ с миром вокруг? Очевидно, что проверка чужой работы (например, математическое доказательство) занимает гораздо меньше времени и творчества, чем поиск собственного решения в большом пространстве кандидатов. Однако из $P = NP$ следует, что творчество можно автоматизировать. Например, доказательство последней теоремы Фер-

---

[1] Остальные шесть: гипотеза Римана, уравнение Навье — Стокса, гипотеза Пуанкаре, гипотеза Ходжа, гипотеза Берча — Свиннертона — Дайера и теория Янга — Миллса. На момент написания этой книги (в 2020 году) была решена только гипотеза Пуанкаре (в 2006 году Григорием Перельманом, который отказался от призовых денег).

[2] Хотя один миллион долларов — это немало, указанная сумма явно недооценивает важность доказательства предположения, что $P \neq NP$.

ма можно сгенерировать алгоритмом за время, полиномиальное по длине доказательства![1]

Рассматривая его как математическое утверждение, какие у нас есть математические свидетельства в пользу или против предположения о том, что P ≠ NP? Здесь мы знаем поразительно мало. Может показаться странным, что никто не смог доказать столь очевидное утверждение. Пугающим барьером является головокружительно богатая фауна страны полиномиально-временных алгоритмов. Если «очевидная» нижняя граница кубического времени выполнения для матричного умножения является ложной (как показано алгоритмом Штрассена в *Первой части*), то где гарантия, что некоторые другие экзотические виды не смогут пробиться через другие «очевидные» нижние границы, включая те, которые предполагаются для NP-трудных задач?

Кто прав: Гедель или Эдмондс? Можно надеяться, что с течением времени мы так или иначе приблизимся к решению предположения о P ≠ NP. Вместо этого, по мере того как все больше и больше математических подходов к задаче оказывались неадекватными, решение все больше отдалялось. Следует осознать суровую реальность, что мы, возможно, не узнаем ответа в течение долгого времени — разумеется, это будут годы, возможно, десятилетия, а может быть, даже столетия.[2]

---

[1] Последствия равенства P = NP будут зависеть от того, можно ли решить все задачи NP алгоритмами, быстрыми на практике или технически выполняемыми за полиномиальное время, но слишком медленными или сложными для применения. Первый и более неправдоподобный сценарий будет иметь огромные последствия для общества, включая конец криптографии и современной электронной коммерции, какой мы ее знаем (см. сноску 2 на с. 56). Об этой возможности выпущена книга Лэнса Фортноу «Золотой билет. P, NP и границы возможного» (М.: Лаборатория знаний: Лаборатория Пилот, 2016). Второй сценарий, о котором размышлял сам Дональд Э. Кнут, необязательно будет иметь практические последствия, а будет только подвергать сомнению определение «поддающийся решению быстрым алгоритмом в физическом мире».

[2] Читайте о гипотезе и ее текущем состоянии в главе Скотта Ааронсона «P = NP» в книге «Открытые задачи по математике» под редакцией Джона Ф. Нэша-младшего и Майкла Т. Рассиаса (*Open Problems in Mathematics, John F. Nash, Jr., Michael Th. Rassias, Springer,* 2016).

*23.5. Гипотеза об экспоненциальном времени

23.5.1. Требуют ли NP-трудные задачи экспоненциального времени?

NP-трудные задачи обычно объединяются с задачами, для решения которых в худшем случае требуется экспоненциальное время (третья допустимая неточность в разделе 19.6). Вместе с тем предположение, что P ≠ NP, не подтверждает это, и даже если оно является истинным, оставляет открытой возможность того, что NP-трудная задача может быть решена за время $n^{O(\log n)}$ или $2^{O(\sqrt{n})}$ на экземплярах с n вершинами. Широко распространенное мнение о том, что типичные NP-трудные задачи требуют экспоненциального времени, формализуется *гипотезой об экспоненциальном времени*.[1]

ГИПОТЕЗА ОБ ЭКСПОНЕНЦИАЛЬНОМ ВРЕМЕНИ

При константе $c > 1$ каждый алгоритм, решающий задачу 3-SAT, в худшем случае требует времени по меньшей мере c^n, где n обозначает число переменных.

Гипотеза об экспоненциальном времени не исключает алгоритмы для задачи 3-SAT, превосходящие исчерпывающий поиск (выполняющийся за время, масштабируемое вместе с 2^n), и не случайно: задача 21.13 показывает, что для 3-SAT *существуют* гораздо более быстрые (если не экспоненциально-временные) алгоритмы. Однако все известные алгоритмы для этой задачи требуют времени c^n для некоторого $c > 1$ (причем текущий рекорд составляет $c \approx 1{,}308$). Гипотеза об экспоненциальном времени предполагает, что это неизбежно.

Редукции показывают, что если гипотеза об экспоненциальном времени является истинной, то многие другие естественные NP-трудные задачи также требуют экспоненциального времени. Например, из гипотезы об экспоненци-

[1] Более сильное утверждение о том, что каждая NP-трудная задача требует экспоненциального времени, является ложным; см. задачу 23.5 для надуманной NP-трудной задачи, которая может быть решена за субэкспоненциальное время.

альном времени следует, что существует константа $a > 1$, такая, что каждый алгоритм для одной из NP-трудных графовых задач в главе 22 в худшем случае требует времени, равного по меньшей мере a^n, где n обозначает число вершин.

23.5.2. Сильная гипотеза об экспоненциальном времени

Гипотеза об экспоненциальном времени является более сильным допущением, чем предположение, что $P \neq NP$: если первая является истинной, то верно и второе. Более сильные допущения приводят к более сильным заключениям и, к сожалению, большей вероятности оказаться ложными (рис. 23.1)! Тем не менее большинство экспертов считают, что гипотеза об экспоненциальном времени является истинной.

Далее следует еще более сильное допущение, которое является более спорным, но имеет замечательные алгоритмические последствия. Что может быть сильнее, чем допустить, что решение NP-трудной задачи требует экспоненциального времени? *Допустить, что для решения задачи нет алгоритма значительно более быстрого, чем исчерпывающий поиск.* Это не может быть правдой для задачи 3-SAT (в силу задачи 21.13), а для другой задачи? Не нужно далеко ходить — общая задача SAT без ограничения числа дизъюнкций в расчете на ограничение (с. 170) станет кандидатом.

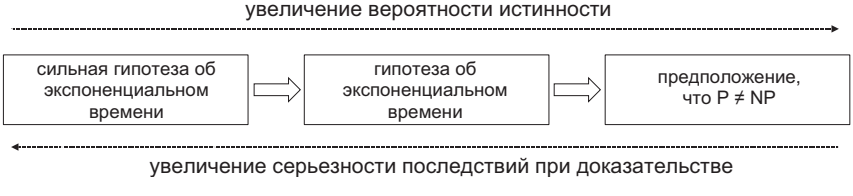


Рис. 23.1. Три недоказанных предположения о труднорешаемости NP-трудных задач, упорядоченные от самых сильных (и наименее правдоподобных) до самых слабых (и наиболее правдоподобных)

Труднорешаемость задачи k-SAT не уменьшается при максимальном числе литералов в расчете на ограничение. Действительно ли трудность задачи строго возрастает вместе с k? Например, рандомизированный алгоритм для задачи 3-SAT в задаче 21.13 может быть расширен до задачи k-SAT для каждого

положительного целого числа k, но его время выполнения ухудшится вместе с k примерно как $(2 - 2/k)^n$, где n — это число переменных (см. сноску 1 на с. 182). Эта же деградация времени выполнения до 2^n по мере увеличения k проявляется во всех известных алгоритмах для задачи k-SAT. Это неизбежно?

СИЛЬНАЯ ГИПОТЕЗА ОБ ЭКСПОНЕНЦИАЛЬНОМ ВРЕМЕНИ

Для каждой константы $c < 2$ существует положительное целое число k, при котором каждый алгоритм, решающий задачу k-SAT, в худшем случае требует времени по меньшей мере c^n, где n обозначает число переменных.[1]

Опровержение сильной гипотезы об экспоненциальном времени повлекло бы за собой значительный теоретический прогресс в алгоритмах для задач SAT — семействе алгоритмов, выполняющихся за время $O((2 - \epsilon)^n)$, где n обозначает число переменных и $\epsilon > 0$ является константой (независимой от k и n, наподобие 0,01 или 0,001). Такой прорыв может произойти или не произойти в ближайшем будущем, в любом случае все готовы к тому, что она будет опровергнута.[2]

23.5.3. Нижние границы времени выполнения для простых задач

Зачем рассказывать о гипотезе, которая может оказаться ложной? Потому что сильная гипотеза об экспоненциальном времени, то есть предположение о труднорешаемости NP-трудных задач, имеет поразительные алгоритмические последствия для задач, поддающихся решению за полиномиальное время.[3]

---

[1] Хотя это и не очевидно, гипотеза об экспоненциальном времени вытекает из сильной гипотезы об экспоненциальном времени.

[2] Обе гипотезы были сформулированы Расселом Импальяццо и Рамамоханом Патури в статье «О сложности задачи k-SAT» («*On the Complexity of k-SAT», Journal of Computer and System Sciences,* 2001).

[3] Гипотеза об экспоненциальном времени имеет интересные алгоритмические следствия, которые, как известно, не следуют из предположения о том, что $P \neq NP$. Например, если она является истинной, то многие NP-трудные задачи не будут допускать алгоритмы с фиксированными параметрами (см. сноску 2 на с. 158).

От сильной гипотезы об экспоненциальном времени к выравниванию рядов

В предыдущих частях мы стремились к Святому Граалю в виде ослепительно быстрого линейно- или почти линейно-временного алгоритма. И даже достигли этой цели для ряда задач (в *Первой* и *Второй частях*), но для других (в *Третьей части*) мы застопорились. Например, для задачи о выравнивании рядов (см. главу 17 *Третьей части*) мы объявили победу с помощью алгоритма динамического программирования `NW` (Нидлмана — Вунша), который выполняется за время $O(n^2)$, где n обозначает длину более длинной из двух входных строк.

Сможем ли мы превзойти квадратично-временной алгоритм выравнивания рядов? Или накопить свидетельства, подтверждающие, что не сможем? Теория NP-трудности, разработанная для рассуждения о задачах, которые выглядят не поддающимися решению за полиномиальное время, внешне не имеет отношения к этому вопросу. Но относительно новая область теории вычислительной сложности, именуемая *мелкозернистой сложностью*, показывает, что допущения о трудности для NP-трудных задач (такие как сильная теория об экспоненциальном времени) содержательно транслируются в задачи, поддающиеся решению за полиномиальное время.[1] Например, лучше-чем-квадратично-временной алгоритм для задачи выравнивания рядов автоматически будет приводить к алгоритму лучшему-чем-исчерпывающий-поиск для задачи k-SAT для всех k![2]

Факт 23.1 (из сильной гипотезы об экспоненциальном времени следует, что алгоритм `NW` по существу является оптимальным). *Для любой константы $\epsilon > 0$ $O(n^{2-\epsilon})$-временной алгоритм для задачи о выравнивании рядов, где*

---

[1] Для глубокого погружения ознакомьтесь с обзором «О некоторых точных вопросах в алгоритмах и сложности» Вирджинии Василевски Уильямс (*«On Some Fine-Grained Questions in Algorithms and Complexity», Virginia Vassilevska Williams, Proceedings of the International Congress of Mathematicians*, 2018).

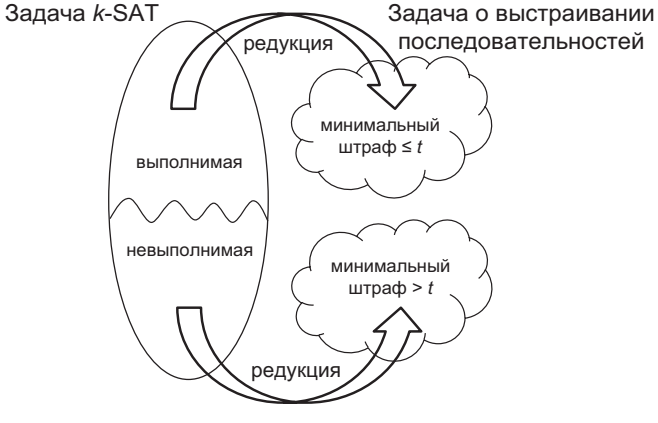
[2] Этот результат появляется в статье «Расстояние редактирования нельзя вычислить за строго субквадратичное время (если только сильная гипотеза об экспоненциальном времени не является ложной)», написанной Артуром Бакуром и Петром Индиком (*«Edit Distance Cannot Be Computed in Strongly Subquadratic Time (Unless SETH Is False)», Arturs Backurs, Piotr Indyk, SIAM Journal on Computing*, 2018).

n — это длина более длинной входной строки, опровергнет сильную гипотезу об экспоненциальном времени.

Другими словами, единственный способ улучшить время работы алгоритма `NW` — это добиться значительного прогресса в решении задачи SAT! Именно в этом состоит потрясающая связь между двумя задачами, которые кажутся крайне разными.

Редукции с экспоненциальным взрывом

Факт 23.1, как и все наши доказательства NP-трудности в главе 22, сводится к редукции — фактически к одной редукции для каждого положительного целого числа k, — где задача k-SAT играет роль известной трудной задачи, и задача о выравнивании рядов — роль целевой задачи:



Но как свести NP-трудную задачу к задаче, поддающейся решению за полиномиальное время, не опровергая предположение, что P ≠ NP? Каждая редукция в основе факта 23.1 задействует препроцессор, который транслирует экземпляр задачи k-SAT с n переменными в экземпляр задачи о выравнивании рядов. Второй экземпляр *экспоненциально больше*, поскольку его входные строки имеют длину N в диапазоне $2^{n/2}$.[1] Редукция обеспечивает, чтобы эк-

[1] Этот экспоненциальный взрыв вызывает экспоненциально большие числа, крайне необходимые для нашей редукции из задачи о независимом множестве к задаче о сумме подмножеств (теорема 22.9); см. сноску на с. 212.

земпляр был выровнен с суммарным штрафом, не превышающим цель t, если и только если экземпляр задачи k-SAT выполним, а постпроцессор может легко извлекать удовлетворимое присвоение из выравнивания.

Почему именно взрыв $N \approx 2^{n/2}$? Потому что это число совпадает с временем работы современных алгоритмов для задачи выравнивания рядов (динамическое программирование) и задачи k-SAT (исчерпывающий поиск). Составление редукции с помощью $O(N^2)$-временного алгоритма выравнивания рядов приводит к алгоритму для задачи k-SAT со временем выполнения $\approx 2^n$, не превосходящему исчерпывающий поиск. Согласно тому же рассуждению, гипотетическая $O(N^{1,99})$-временная подпрограмма (скажем) выравнивания рядов автоматически будет приводить (для каждого k) к алгоритму, который решает задачу k-SAT за время примерно $O((2^{n/2})^{1,99}) = O((1{,}9931)^n)$. Поскольку основание этой экспоненты меньше 2 (для всех k), такой алгоритм опровергнет сильную гипотезу об экспоненциальном времени.[1]

*23.6. NP-полнота

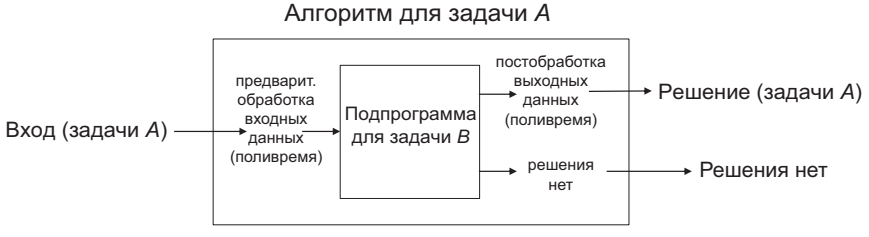
Полиномиально-временная подпрограмма для NP-трудной задачи, такой как задача 3-SAT, — это все, что нужно для решения задачи в NP за полиномиальное время. Но верно и нечто еще более сильное: каждая задача в NP — это *тонко замаскированный частный случай задачи 3-SAT*. Другими словами, задача 3-SAT является универсальной среди задач NP, поскольку она одновременно кодирует каждую отдельную задачу NP! В этом и заключается смысл NP-полноты. Поисковые версии почти всех задач, рассмотренных в главе 22, являются NP-полными.

23.6.1. Редукции Левина

Идея о том, что одна поисковая задача A является «тонко замаскированным частным случаем» другой поисковой задачи B, выражается через высоко ограниченный тип редукции, именуемой редукцией Левина. Подобно простейшим редукциям, введенным в разделе 22.4, *редукция Левина* выполняет только три

[1] Задача 23.7 очерчивает более простую редукцию этого типа для задачи вычисления диаметра графа.

шага: преобразовывает в шаге предварительной обработки экземпляр *A* в один из *B*, вызывает подпрограмму для *B* и преобразовывает в шаге постобработки допустимое решение, возвращаемое подпрограммой (если таковое имеется), в решение для экземпляра *A*.[1, 2]



РЕДУКЦИЯ ЛЕВИНА ИЗ A К B

1. *Препроцессор*: получив экземпляр *I* задачи *A*, преобразовать его за полиномиальное время в экземпляр *I′* задачи *B*.
2. *Подпрограмма*: вызвать предполагаемую подпрограмму для *B* с входными данными *I′*.
3. *Постпроцессор (допустимый случай)*: если подпрограмма возвращает в *I′* допустимое решение, то преобразовать его за полиномиальное время в допустимое решение в *I*.
4. *Постпроцессор (недопустимый случай)*: если подпрограмма возвращает «решения нет», то вернуть «решения нет».

Во всех книгах этой серии мы непреднамеренно использовали только редукции Левина, а не полную мощь общих редукций (Кука). Наша редукция в теореме 22.4 из задачи 3-SAT к задаче об ориентированном гамильтоновом

[1] Редукции Левина подчиняются лекалу из раздела 22.4, причем: (1) обе задачи должны быть поисковыми и (2) постпроцессор должен откликнуться сообщением «решения нет», если и только если об этом говорит предполагаемая подпрограмма.

[2] Если *A* и *B* являются задачами принятия решения («да»/«нет»), а не задачами поиска, то постобработка не требуется, и (двоичный) ответ, возвращаемый подпрограммой для *B*, может быть передан без изменений в качестве окончательного результата. Этот аналог редукции Левина для задач принятия решения имеет ряд названий: редукция Карпа, полиномиально-временная редукция «многие-к-одному» и редукция с полиномиально-временным отображением.

пути (с. 205) является каноническим примером: получив экземпляр задачи 3-SAT, препроцессор строит направленный граф, который затем подается в подпрограмму для вычисления *s*-*t*-вершинного гамильтонова пути, и если подпрограмма возвращает такой путь, то постпроцессор извлекает из него удовлетворяющее присвоение истинности.[1]

23.6.2. Самые трудные задачи в NP

Задача *B* является NP-трудной, если она алгоритмически достаточна для решения всех задач NP за полиномиальное время, имея в виду, что для каждой задачи *A* в NP существует редукция (Кука) из *A* к *B* (с. 230). Чтобы быть NP-полной, задача *B* должна принадлежать к классу NP и включать все другие задачи NP как замаскированные частные случаи.

NP-ПОЛНАЯ ЗАДАЧА

Вычислительная задача *B* является NP-полной, если:

1. Для каждой задачи в NP есть редукция Левина из *A* к *B*;
2. *B* является членом класса NP.

Поскольку редукция Левина является частным случаем редукции (Кука), каждая NP-полная задача автоматически является NP-трудной.[2]

NP-полные задачи — наиболее трудные в NP. Каждая такая задача одновременно кодирует все задачи поиска с легко распознаваемыми решениями.[3]

---

[1] Остальные три главные редукции в главе 22 (теоремы 22.2, 22.7 и 22.9) превращаются в редукции Левина, как только задача оптимизации в оригинале заменяется ее поисковой версией (убедитесь сами). Например, редукция из задачи о неориентированном гамильтоновом пути к задаче коммивояжера (теорема 22.7) требует только подпрограммы для поисковой версии задачи коммивояжера (чтобы проверить наличие нуль-стоимостного тура).

[2] По причине второго условия на NP-полноту могут претендовать только поисковые задачи. Например, задача коммивояжера является NP-трудной, но не NP-полной, тогда как ее поисковая версия оказывается как NP-трудной, так и NP-полной.

[3] В большинстве книг NP-полнота определяется с помощью задач принятия решений (а не поиска) и редукций Карпа (а не Левина) (сноска 2 на с. 244). Интерпретация и алгоритмические последствия NP-полноты по сути одни и те же.

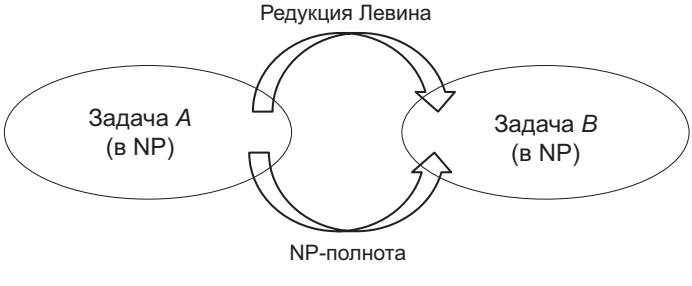
23.6.3. Существование NP-полных задач

Разве не круто звучит определение NP-полной задачи? *Одна-единственная* задача поиска с эффективно распознаваемыми решениями, которая одновременно кодирует *все* такие задачи поиска? Удивительно, что такая задача вообще может существовать!

Но подождите... на самом деле мы не видели никаких примеров NP-полных задач. Существуют ли они вообще? Может ли действительно существовать такая «универсальная» задача поиска? Да, и теорема Кука — Левина уже это доказывает! Причина в том, что ее доказательство (раздел 23.3.5) использует только редукцию Левина — препроцессор, который преобразовывает экземпляры произвольной задачи NP в экземпляры задачи 3-SAT, и постпроцессор, который извлекает допустимые решения из удовлетворяющих присвоений истинности. Поскольку задача 3-SAT также является членом NP, она с честью проходит оба теста на NP-полноту.

Теорема 23.2 (теорема Кука — Левина (более сильная версия)). *Задача 3-SAT является NP-полной.*

Доказать с нуля, что задача является NP-полной, весьма непросто — Кук и Левин не зря получили премии, — зато повторять этот опыт больше не нужно. Редукции Левина распространяют NP-полноту так же, как редукции (Кука) распространяют NP-трудность — от одной задачи к другой (задача 23.4):



Поэтому, чтобы доказать NP-полноту задачи, просто следуйте трехшаговому рецепту (где третьим шагом будет проверка принадлежности задачи к NP).

КАК ДОКАЗАТЬ, ЧТО ЗАДАЧА ЯВЛЯЕТСЯ NP-ПОЛНОЙ

Доказать, что задача *B* является NP-полной:

1. Доказать, что *B* является членом класса NP.
2. Выбрать NP-полную задачу *A*.
3. Доказать, что существует редукция Левина из *A* к *B*.

Этот рецепт применялся много раз, и мы теперь знаем, что тысячи задач являются NP-полными, включая задачи из всех областей техники, естественных и социальных наук. Например, поисковые версии почти всех задач, рассмотренных в главе 22, являются NP-полными (задача 23.3).[1] В книге Гэри и Джонсона (см. сноску на с. 193) перечисляются еще сотни подобных примеров.[2]

ВЫВОДЫ

- ✶ Чтобы накопить свидетельства труднорешаемости задачи, нужно доказать, что многие другие труднорешаемые задачи сводятся к ней.
- ✶ В задаче поиска нужно вывести либо допустимое решение, либо сообщение о его отсутствии.

---

[1] Исключение? Задача о максимизации влияния, поисковая версия которой явно не находится в NP (см. решение к тестовому заданию 23.1).

[2] Термин «NP-полная» оказывает медвежью услугу фундаментальному понятию, которое он определяет, что заслуживает признания и удивления. Как зафиксировано в статье Дональда Э. Кнута «Терминологическое предложение» («*A Terminological Proposal», SIGACT News,* 1974), Кнут предлагал названия «геркулесовая» (Herculean), «внушительная» (formidable) и «тяжелая» (arduous). Кеннет Штейглиц обыграл фамилию Кук (Cook — повар), предложив термин «hard-boiled» (дословно «сваренная вкрутую»). Альберт Р. Мейер называл задачи «крепкими», сократив название «трудная в выполнении» до hard-ass (крепкозадая). Шэнь Линь предложил «PET» как гибкую аббревиатуру, означающую то «вероятно, экспоненциальное время» (probably exponential time), то «доказуемо экспоненциальное время» (provably exponential time), то «ранее экспоненциальное время» (previously exponential time). (Но не будем поднимать проблему 23.5...)

- NP — это множество всех поисковых задач, для которых допустимые решения имеют полиномиальную длину и могут быть проверены за полиномиальное время.
- Задача является NP-трудной, если каждая задача в NP сводится к ней.
- P — это множество всех задач NP, которые могут быть решены с помощью полиномиально-временного алгоритма.
- Предположение, что $P \neq NP$, утверждает, что $P \subsetneq NP$.
- Гипотеза об экспоненциальном времени утверждает, что естественные NP-трудные задачи, такие как 3-SAT, требуют экспоненциального времени.
- Сильная гипотеза об экспоненциальном времени утверждает, что по мере увеличения k ни один алгоритм для задачи k-SAT не может превзойти исчерпывающий поиск.
- Если сильная гипотеза об экспоненциальном времени является истинной, то ни один алгоритм для задачи о выравнивании рядов существенно не превосходит алгоритм Нидлмана — Вунша.
- Редукция Левина выполняет минимально возможную работу: предварительная обработка входных данных; вызов предполагаемой подпрограммы; постобработка выходных данных.
- Задача B является NP-полной, если она принадлежит классу NP и для каждой задачи $A \in$ NP существует редукция Левина из A к B.
- Чтобы доказать, что задача B является NP-полной, нужно выполнить трехшаговый рецепт: (1) доказать, что $B \in$ NP, (2) выбрать NP-полную задачу A и (3) разработать редукцию Левина из A к B.
- Теорема Кука — Левина доказывает, что задача 3-SAT является NP-полной.

Задачи на закрепление материала

Задача 23.1 *(S)*. Какое из следующих утверждений может быть истинным, учитывая имеющиеся знания? (Правильных ответов может быть несколько.)

а) Существует NP-трудная задача, которая поддается решению за полиномиальное время.

б) Предположение, что P ≠ NP, является истинным, и задача 3-SAT может быть решена за время $2^{O(\sqrt{n})}$, где n — это число переменных.

в) Не существует NP-трудной задачи, которую можно решить за времея $2^{O(\sqrt{n})}$, где n — это размер входных данных.

г) Некоторые NP-полные задачи поддаются решению за полиномиальное время, а некоторые — нет.

Задача 23.2 *(S)*. Докажите, что предположение Эдмондса 1967 года о том, что оптимизационная версия задачи коммивояжера не может быть решена никаким полиномиально-временным алгоритмом, эквивалентно предположению, что P ≠ NP.

Задача 23.3 *(S)*. Какую из восемнадцати редукций в разделе 22.3.2 можно легко превратить в редукции Левина между поисковыми версиями соответствующих задач?

Задачи повышенной сложности

Задача 23.4 *(H)*. Эта задача формально оправдывает рецепты на с. 186 и 246 для доказательства, что задача является NP-трудной и NP-полной.

а) Докажите, что если задача A сводится к задаче B и задача B сводится к задаче C, то задача A сводится к задаче C.

б) Заключите, что если NP-трудная задача сводится к задаче B, то B также является NP-трудной. (Использовать формальное определение NP-трудности на с. 230.)

в) Докажите, что если существуют редукции Левина из задачи A к задаче B и из B к задаче C, то существует редукция Левина из A к C.

г) Заключите, что если задача B принадлежит NP и существует редукция Левина из NP-полной задачи к B, то B также является NP-полной.

Задача 23.5 *(S)*. Назовем экземпляр задачи 3-SAT *дополненным*, если ее список ограничений завершается n^2 избыточными копиями однолитераль-

ного ограничения «x_1», где n обозначает число булевых переменных, а x_1 — первая из них.

В *дополненной задаче 3-SAT* входные данные такие же, как и в обычной такой задаче. Если данный экземпляр указанной задачи не дополнен или невыполним — нужно вернуть «решения нет». В противном случае — вернуть удовлетворяющее присвоение истинности для дополненного экземпляра.

а) Докажите, что дополненная задача 3-SAT является NP-трудной (или даже NP-полной).

б) Докажите, что дополненная задача 3-SAT может быть решена за субэкспоненциальное время, а именно $2^{O(\sqrt{N})}$ для N-размерных входных данных.

Задача 23.6 *(H)*. Допустим, что гипотеза об экспоненциальном времени (с. 238) является истинной. Докажите, что существует задача в NP, которая не поддается решению за полиномиальное время и не является NP-трудной.[1]

Задача 23.7 *(H)*. Диаметр неориентированного графа $G = (V, E)$ равен максимальному расстоянию кратчайшего пути между любыми двумя вершинами: $\max_{v,w \in V} dist(v, w)$, где $dist(v, w)$ обозначает минимальное число ребер в v-w-вершинном пути в графе G (либо $+\infty$, если такого пути не существует).

а) Объясните, как вычислить диаметр графа за время $O(mn)$, где n и m обозначают число соответственно вершин и ребер графа G. (Можете принять, что n и m не меньше 1.)

б) Допустим, что сильная гипотеза об экспоненциальном времени (с. 239) является истинной. Докажите, что для каждой константы $\epsilon > 0$ не существует $O((mn)^{1-\epsilon})$-временного алгоритма для вычисления диаметра графа.

---

[1] Знаменитый и труднее доказуемый результат, известный как теорема Ладнера, показывает, что вывод остается истинным, исходя только из более слабого предположения о том, что $P \neq NP$.

24 Практический пример: стимулирующий аукцион FCC

NP-трудность — это не чисто академическая концепция, она действительно управляет массой вычислительно допустимых вариантов при решении реальных задач. Эта глава подробно описывает недавнюю иллюстрацию важности NP-трудности в контексте ответственной экономической задачи: эффективного перераспределения дефицитного ресурса (беспроводного радиочастотного спектра). Решение, развернутое правительством США и известное как стимулирующий аукцион Федеральной комиссии по связи (FCC), опиралось на удивительно широкий диапазон алгоритмического инструментария, который есть в этой книге. Внимательно изучая его детали, найдите время, чтобы оценить мастерское владение алгоритмами, которое вы тоже приобрели, начиная с умножения Карацубы и алгоритма слияния в главе 1 *Первой части*. То, что начиналось как какофония таинственных и несвязных трюков, превратилось в симфонию взаимосвязанных методов проектирования алгоритмов.[1]

---

[1] Узнать больше о стимулирующем аукционе FCC от его ведущих дизайнеров — Кевина Лейтона-Брауна, Пола Милгрома и Ильи Сегала — можно из их статьи «Экономика и информатика перераспределения радиочастотного спектра» (*Economics and Computer Science of a Radio Spectrum Reallocation, Proceedings of the National Academy of Sciences,* 2017). Для глубокого погружения в параллели между аукционами и алгоритмами ознакомьтесь с моей книгой «Двадцать лекций по алгоритмической теории игр» (*Twenty Lectures on Algorithmic Game Theory, Cambridge University Press,* 2016).

24.1. Перенацеливание беспроводного спектра

24.1.1. От телевидения к мобильным телефонам

Телевидение распространилось в США подобно лесному пожару в 1950-х гг. В те дни телевизионные программы передавались исключительно по воздуху с помощью радиоволн, посылаемых передатчиком станции и передаваемых телевизионной антенной. Для координации передач станций и предотвращения помех между ними FCC разделила используемые частоты — *спектр* — на блоки по 6 МГц, именуемые *каналами*. Разные станции в одном и том же городе стали транслироваться по разным каналам. Например, канал 14 относится к частотам между 470 МГц и 476 МГц; канал 15 — к частотам между 476 МГц и 482 МГц и так далее.[1]

Знаете, что еще передается по радиоволнам по воздуху? Данные, которыми мобильные телефоны обмениваются с ближайшей базовой станцией. Например, если сейчас 2020 год и вашим оператором является Verizon Wireless, то, скорее всего, вы скачивали и закачивали данные, используя частоты соответственно 746–756 МГц и 777–787 МГц. Во избежание помех часть спектра, зарезервированная для сотовых данных, не перекрывается с той, которая зарезервирована для наземного (то есть эфирного) телевидения.

Использование мобильных и беспроводных данных стремительно растет, увеличившись примерно на порядок только за последние пять лет. Для передачи большего объема данных требуется больше выделенных частот, и не все частоты полезны для беспроводной связи. (Например, при ограниченной мощности очень высокие частоты могут передавать сигналы только на короткие расстояния.) Спектр — это дефицитный ресурс, и современные технологии жаждут получить его.

---

[1] Сверхвысокочастотные (СВЧ, англ. аббр. UHF) каналы начинаются с 470 МГц и идут оттуда вверх в 6-мегагерцовых (МГц) блоках. Очень высокочастотные (ОВЧ, англ. аббр. VHF) каналы используют более низкие частоты, 174–216 МГц (для каналов 7–13) и 54–88 МГц (для каналов 2–6, а также 4 МГц для прочих применений, таких как открыватели гаражных ворот).

Телевидение все еще может быть большим, но наземное телевидение — нет. Примерно 85–90 % американских домохозяйств полагаются исключительно на кабельное телевидение (которое вообще не требует эфирного спектра) или спутниковое телевидение (которое использует гораздо более высокие частоты, чем обычные беспроводные приложения). Резервирование наиболее ценного спектрального имущества для эфирного телевидения имело смысл в середине XX века, но не в начале XXI.

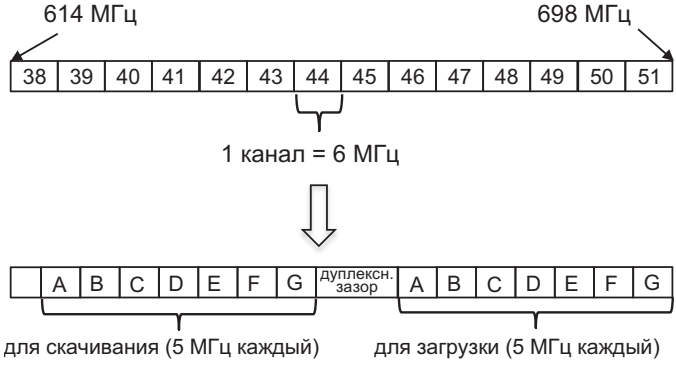
24.1.2. Недавнее перераспределение спектра

На момент написания этой книги основное перераспределение спектра почти завершено. После 13 июля 2020 года в США больше не будет телевизионных станций, вещающих в прямом эфире на самых высоких каналах, четырнадцати каналах между 38 и 51 (614–698 МГц). Каждая станция, которая вещала по одному из этих каналов, либо переключается на более низкий канал, либо прекращает все наземные передачи (хотя, возможно, все еще вещает по кабельному и спутниковому телевидению). Даже станции, которые уже вещали на каналах ниже 38, уходят из эфира или мигрируют на другие каналы, чтобы освободить место для своих товарищей, спускающихся с более высоких каналов. В общей сложности 175 станций отказываются от своих лицензий на вещание и примерно 1000 переключают каналы.[1]

Освобожденные 84 МГц спектра были реорганизованы и переданы телекоммуникационным компаниям, таким как T-Mobile, Dish и Comcast, которые, как ожидается, в ближайшие годы будут их использовать для создания беспроводных сетей нового поколения. (T-Mobile, например, уже переключила коммутатор на свою новую общенациональную сеть 5G.) Там, где раньше были каналы 38–51, теперь есть семь независимых пар блоков по 5 МГц. Например, первая пара содержит частоты 617–622 МГц (предназначенные для

[1] С момента перехода в 2009 году с аналогового на исключительно цифровое вещание наземного телевидения логический канал (отображаемый на телевизионной приставке) может быть переназначен на физический канал, отличный от того, который исторически ассоциировался с этим номером канала. Таким образом, станция может сохранять свой логический канал даже при переназначении физического канала.

скачивания на устройство) и 663–668 МГц (предназначенные для закачивания на станцию), вторая 622–627 МГц и 668–673 МГц, и так далее.[1]



Все это внешне должно выглядеть как большая запутанная операция. Какие станции должны уйти из эфира? Какие каналы следует переключить? Какими должны быть их новые каналы? В какой мере владельцы станций должны компенсировать свои потери? Какие телекоммуникационные компании должны быть вознаграждены вновь созданными парными блоками спектра? Сколько они должны за них платить? На все эти вопросы отвечает стимулирующий аукцион FCC — сложный алгоритм, который в значительной степени опирается на инструментарий для решения NP-трудных задач, описанных в этой книге.

24.2. Жадные эвристики для выкупа лицензий

Стимулирующий аукцион FCC состоял из двух частей: *обратного аукциона* для решения о том, какие телевизионные станции уйдут из эфира или переключат каналы, и соответствующей компенсации за них, и *прямого аукциона* для выбора получателей блоков недавно освобожденного спектра

[1] Существует также дуплексный зазор 11 МГц (652–663 МГц), разделяющий два типа блоков, и защитная полоса 3 МГц (614–617 МГц) во избежание помех на канале 37 (608–614 МГц), который уже давно зарезервирован для радиоастрономии и беспроводной медицинской телеметрии.

по определенным ценам. Правительство США (и многих других стран) уже 25 лет с успехом проводит предварительные аукционы по продаже лицензий на спектр, внося в них небольшие коррективы. Настоящее тематическое исследование фокусируется на обратном аукционе стимулирующего аукциона FCC, в котором заложена основная часть его инноваций.

24.2.1. Четыре временных упрощающих допущения

FCC предоставляет вещателю имущественные права с помощью *лицензии на вещание*, которая разрешает вещание по каналу в определенном географическом регионе. FCC берет на себя ответственность за устранение помех.[1]

Цель обратного аукциона состояла в возврате достаточного количества лицензий от телевизионных станций и освобождении целевого объема спектра (например, каналов 38–51). Чтобы получить первое представление об этой задаче, примем несколько упрощающих допущений, которые потом будут устранены.

ВРЕ́МЕННЫЕ УПРОЩАЮЩИЕ ДОПУЩЕНИЯ

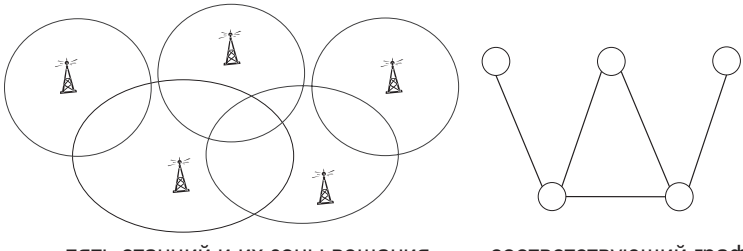
1. Все станции, которые останутся в эфире, будут транслироваться по одному каналу (скажем, каналу 14).
2. Две станции могут вещать на одном канале одновременно, если и только если их зоны вещания не перекрываются.
3. Существует известная ценность (стоимость) каждой станции.
4. Правительство может в одностороннем порядке решить, какие станции останутся в эфире.

[1] Для стимулирующего аукциона FCC закрепление канала за станцией не рассматривалось как часть имущественных прав владельца лицензии. Чтобы разрешить аукциону перераспределять каналы станций, потребовалось принятие закона Конгрессом. (Одного из восьми законопроектов, принятых в 2012 году, — «Закона о налоговых льготах для среднего класса и создании рабочих мест».)

Нужно, чтобы наиболее ценные станции сохранили свои лицензии. Для этого мы выявим множество не создающих помех станций с максимально возможной суммой стоимостей станций. Вы узнаёте эту задачу оптимизации?

24.2.2. Засада со стороны взвешенного независимого множества

Это в точности задача о взвешенном независимом множестве (с. 41)! Вершины соответствуют станциям, ребра — парам станций, создающих друг другу помехи, а стоимости станций — весам вершин:



пять станций и их зоны вещания соответствующий граф

Из следствия 22.3 мы знаем, что эта задача является NP-трудной, даже если каждая вершина имеет вес 1. Задача может быть решена за линейное время с помощью динамического программирования, когда входной граф представляет собой дерево (глава 16 *Третьей части*), но помеховые шаблоны телевизионных станций не являются древовидными. Например, все станции в одном городе создают друг другу помехи, приводя к клике в соответствующем графе.

Поиски внутри алгоритмического инструментария

Теперь, когда мы диагностировали задачу как NP-трудную, самое время начать искать лекарство в алгоритмическом инструментарии. (NP-трудность — это не смертный приговор!) Самое интересное, удастся ли решить эту задачу строго за приемлемое количество времени — скажем, менее чем за неделю?

Ответ на этот вопрос зависит от размера задачи. Если задействовать только тридцать станций, то исчерпывающий поиск будет работать просто на ура. Но реально-практическая задача состояла в тысячах участвующих станций и десятках тысяч помеховых ограничений — намного выше уровня расплаты за исчерпывающий поиск и методы динамического программирования из разделов 21.1–21.2.

Последней надеждой на точный алгоритм был решатель задач MIP (раздел 21.4). Задача о взвешенном независимом множестве легко кодируется как задача MIP (задача 21.9), и именно с этого FCC начала. К сожалению, задача оказалась слишком большой, и даже лучшие решатели на ней запнулись (точнее, на более реалистичной многоканальной версии задачи, описанной в разделе 24.2.4). Когда все возможности для абсолютно правильного алгоритма были исчерпаны, FCC пошла на компромисс в отношении правильности и обратилась к быстрым эвристическим алгоритмам.

24.2.3. Жадные эвристические алгоритмы

Для задачи о взвешенном независимом множестве, как и для многих других, жадные алгоритмы являются идеальным местом для запуска мозгового штурма в отношении быстрых эвристических алгоритмов.

Базовый жадный алгоритм

Возможно, наипростейший жадный подход к задаче о взвешенном независимом множестве состоит в имитации алгоритма Краскала для задачи о минимальном остовном дереве и выполнении одного прохода по вершинам (в порядке убывания веса) с постоянным добавлением по одной вершине к выходным данным, если это не нарушает допустимость.

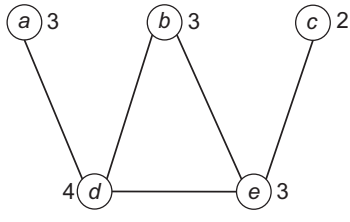
WISBASICGREEDY

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

Выход: независимое множество графа G.

$S := \emptyset$
Отсортировать вершины V от высшего к низшему весу
`// Главный цикл`
for each $v \in V$, в неубывающем порядке веса **do**
  if $S \cup \{v\}$ является допустимым **then** `// все несмежные`
    $S := S \cup \{v\}$
return S

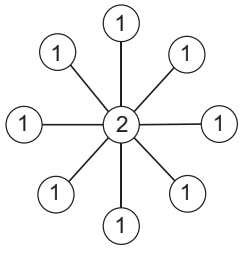
Например, в графе



вершины помечены их весами

базовый жадный алгоритм задачи о взвешенном независимом множестве `WISBasicGreedy` отбирает вершину d с наибольшим весом на первой итерации, пропускает вершины с весом 3 на второй, третьей и четвертой итерациях (поскольку каждая из них примыкает к d) и завершает отбором вершины c. Результирующее независимое множество имеет суммарный вес 6 и не является оптимальным (так как независимое множество $\{a, b, c\}$ имеет суммарный вес 8).

Поскольку задача о взвешенном независимом множестве является NP-трудной, а алгоритм `WISBasicGreedy` выполняется за полиномиальное время, результат предсказуем. Но вот более тревожный случай (с вершинами, помеченными их весами):



Алгоритм `WISBasicGreedy` обманом привязывается к центру звезды, не позволяя ему взять какой-либо из листьев. Как предотвратить подобные ловушки?

Вершинноспецифичные множители

Во избежание ошибок алгоритма `WISBasicGreedy` мы можем поставить в худшие условия вершины с соседями. Например, признаем, что выбор вершины v начисляет выгоду от w_v, и проигнорируем $1 + deg(v)$ вершин (где $deg(v)$ обозначает степень вершины v). Один проход алгоритма будет осуществляться в убывающем порядке отношения отдачи к вложенным средствам $w_v/(1 + deg(v))$, а не веса w_v.[1] (Этот жадный алгоритм возвращает независимое множество с максимальным весом в двух примерах.) В более общем случае алгоритм вычислит вершинноспецифичные множители на шаге предварительной обработки, после чего перейдет к своему единственному проходу над вершинами.

WISGENERALGREEDY
вычислить β_v для каждого $v \in V$ `// напр.:` $\beta_v = 1 + deg(v)$
$S := \emptyset$
отсортировать вершины V от высшего к низшему значению w_v/β_v
for each $v \in V$, в неуменьшающемся порядке w_v/β_v **do**
  if $S \cup \{v\}$ является допустимым **then** `// все несмежные`
    $S := S \cup \{v\}$
return S

Каков наилучший вариант выбора для вершинноспецифичных множителей? Независимо от того, насколько хитроумна формула для каждого параметра β_v, обязательно будут примеры, в которых алгоритм `WISGeneralGreedy` возвращает неоптимальное независимое множество (исходя из того, что параметры β_v могут вычисляться за полиномиальное время, и истинности предположе-

[1] Вы, возможно, узнали эту идею из задачи 20.3 и жадного эвристического алгоритма для задачи о рюкзаке, который сортирует элементы в убывающем порядке отношений стоимости к размеру.

ния, что $P \neq NP$). Наилучший вариант выбора зависит от экземпляров задачи в приложении и должен быть определен эмпирически с использованием репрезентативных экземпляров.[1]

Станционноспецифичные параметры в стимулирующем аукционе FCC

Репрезентативные экземпляры задачи о взвешенном независимом множестве и более общей многоканальной задачи, описанной в следующем разделе, были легко найдены в фазе разработки обратного аукциона. Граф, произведенный из участвующих станций и их районов вещания, был в полной мере известен заранее. Основываясь на исторических данных, можно выдвинуть обоснованные догадки о диапазоне вероятных весов вершин (стоимостей станций). С тщательно настроенным выбором вершинноспецифичных множителей алгоритм `WISGeneralGreedy` (и многоканальное обобщение `FCCGreedy`, описанное в следующем разделе) рутинно возвращал решения репрезентативных экземпляров с суммарным весом, превышающим 90 % от максимально возможного.[2, 3]

24.2.4. Многоканальный случай

Пора отказаться от первого упрощающего допущения из раздела 24.2.1 и разрешить назначать остающимся в эфире станциям любой из k каналов.

---

[1] Общий совет для обуздания NP-трудных задач в реальном приложении: используйте как можно больше знаний о предметной области!

[2] Как были рассчитаны параметры в фактическом стимулирующем аукционе FCC? Посредством формулы $\beta_v = \sqrt{\deg(v)} \times \sqrt{\text{pop}(v)}$, где $\deg(v)$ и $\text{pop}(v)$ обозначают соответственно число станций, перекрывающихся со станцией v, и население, обслуживаемое станцией v. Член $\sqrt{\deg(v)}$ ставил в худшие условия станции, которые блокируют много других станций из тех, что остались в эфире. Смысл члена $\sqrt{\text{pop}(v)}$ был тоньше (и более спорным). Его эффект заключался в уменьшении компенсации, выплачиваемой правительством малым телевизионным станциям, которые, скорее всего, все равно уйдут из эфира.

[3] FCC также была способна получать высококачественные решения в разумные сроки путем ранней остановки современного решателя задач MIP до того, как найти оптимальное решение (см. с. 167). Жадный подход в конечном счете победил благодаря его легкому переводу в прозрачный формат аукциона (раздел 24.4).

Алгоритм `WISGeneralGreedy`, по всей видимости, легко распространяется на многоканальную версию задачи.[1]

FCCGREEDY
вычислить β_v для каждой станции v
$S := \emptyset$
отсортировать станции от высшего до низшего значения w_v/β_v
for each станция v, в невозрастающем порядке w_v/β_v **do**
  if $S \cup \{v\}$ является допустимым **then** `// настроить на k-каналах`
    $S := S \cup \{v\}$
return S

Выглядит как другие (полиномиально-временные) жадные алгоритмы, не правда ли? Но перейдем к итерации главного цикла, который отвечает за проверку возможности добавить текущую станцию v к текущему решению S, не разрушая допустимость.

Что делает подмножество станций допустимым? Допустимость означает, что все станции могут находиться в эфире одновременно, без помех. То есть должно быть закрепление станций в $S \cup \{v\}$ за k каналами, чтобы две станции с перекрывающимися областями вещания не закреплялись за одним каналом. Узнаёте задачу?

24.2.5. Засада со стороны раскраски графа

Это в точности задача о раскраске графа (с. 169)! Вершины соответствуют станциям, ребра — парам станций с перекрывающимися областями вещания, и k цветов — k имеющимся каналам.

[1] Подумайте о $k = 23$, соответствующем каналам 14–36. Стимулирующий аукцион FCC также позволил СВЧ-станциям опуститься до ОВЧ-диапазона (каналы 2–13), но основная часть действий происходила в СВЧ-диапазоне.

Как мы знаем из задачи 22.11, задача о раскраске графа является NP-трудной даже при $k = 3$.[1] Хуже того, алгоритм `FCCGreedy` должен решить *много* экземпляров задачи о раскраске графа, по одному на каждой итерации главного цикла. Как эти экземпляры связаны между собой?

ТЕСТОВОЕ ЗАДАНИЕ 24.1

Рассмотрим ряд экземпляров задачи о проверке допустимости, которая возникает в алгоритме `FCCGreedy`. Какое из следующих утверждений является истинным? (Правильных ответов может быть несколько.)

а) Если экземпляр на одной итерации является допустимым, то таким же является и экземпляр на следующей итерации.

б) Если экземпляр на одной итерации является недопустимым, то таким же является и экземпляр на следующей итерации.

в) Экземпляр на данной итерации имеет на одну станцию больше, чем экземпляр на предыдущей итерации.

г) Экземпляр на данной итерации имеет на одну станцию больше, чем самый недавний допустимый экземпляр.

(Ответ и анализ решения см. в разделе 24.2.6.)

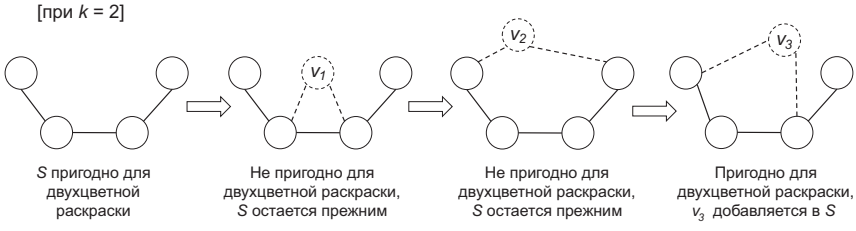
И что теперь? Разве диагностика нашей задачи о проверке допустимости как NP-трудной задачи о раскраске графа исключает использование жадного эвристического алгоритма для приближенной максимизации суммарной стоимости станций, остающихся в эфире?

24.2.6. Решение к тестовому заданию 24.1

Правильный ответ: (г). Ответ (а) явно неправильный: первый экземпляр является всегда допустимым, тогда как некоторые экземпляры ближе к концу

[1] Проверка допустимости в частном случае одиночного канала (раздел 24.2.2) соответствует задаче проверки на одноцветность или, что то же самое, проверки, является ли множество вершин независимым множеством.

алгоритма могут такими и не быть. Ответ (б) также неправильный. Например, текущее решение может блокировать всех новичков в северо-восточном регионе США, оставив западное побережье широко открытым. Ответ (в) неправильный, а (г) является правильным, так как текущее решение S изменяется только на итерации, в которой множество станций $S \cup \{v\}$ является допустимым. Например, в терминах раскраски графа:



24.3. Проверка допустимости

Появись у нас волшебный ящик для проверки допустимости, мы выполнили бы алгоритм `FCCGreedy`, чтобы после настройки станционноспецифичных множителей гарантированно вычислить допустимые решения с суммарной стоимостью, близкой к максимально возможной. Наши мечты о волшебных ящиках уже однажды были сорваны, и первая задача о максимизации стоимости оказалась слишком трудной для самых лучших решателей задач MIP (раздел 24.2.2). Почему мы продолжаем верить в волшебство?

24.3.1. Кодирование в качестве задачи выполнимости

Подпрограмма, требуемая алгоритмом `FCCGreedy`, отвечает только за проверку допустимости (узнает, пригоден ли подграф для k-цветной раскраски), а не за оптимизацию (не ищет самый дорогой и пригодный для k-цветной раскраски подграф). Это укрепляет надежды на волшебный ящик, который решает более «легкую» (если не NP-трудную) задачу о проверке допустимости, даже если для задачи оптимизации ее не существует. Разворот от оптимизации в сторону проверки допустимости подразумевает начало экспериментов с логическим языком и решателями задач SAT.

Формулировка задачи о раскраске графа как задачи SAT из раздела 21.5.3 здесь актуальна. Напомню, для каждой вершины v входного графа и разрешимого цвета $i \in \{1, 2, ..., k\}$ существует булева переменная x_{vi}. Для каждого ребра (u, v) входного графа и цвета i существует ограничение

$$\neg x_{ui} \vee \neg x_{vi}, \tag{24.1}$$

которое исключает присвоение цвета i как вершине u, так и вершине v. Для каждой вершины v входного графа существует ограничение

$$x_{v1} \vee x_{v2} \vee ... \vee x_{vk}, \tag{24.2}$$

которое исключает возможность оставить v неокрашенной.

Опционально для каждой вершины v и разных цветов $i, j \in \{1, 2, ..., k\}$, ограничение

$$\neg x_{vi} \vee \neg x_{vj} \tag{24.3}$$

может использоваться, чтобы исключить присвоение вершине v как цвета i, так и цвета j.[1]

24.3.2. Встраивание реберных ограничений

Фактический стимулирующий аукцион FCC использовал формулировку несколько более сложную, чем в (24.1)–(24.3). Станции с перекрывающимися областями вещания мешают закреплению одного канала и, в зависимости от нескольких факторов, могут мешать его соседям. Отдельная команда в FCC заранее определила для каждой пары станций пары закреплений каналов, создающие помехи. Этот список запрещенных попарных закреплений каналов, хотя и трудно компилируемый, был легко встроен в формулировку выполнимости с одним ограничением формы

$$\neg x_{uc} \vee \neg x_{vc'} \tag{24.4}$$

---

[1] Вершины могут получать многочисленные цвета, если эти ограничения опущены, но каждый способ выбора среди присвоенных цветов приводит к k-цветной раскраске.

для каждой пары u, v станций и закреплений за ними запрещенных каналов c, c'. Например, ограничение: $\neg x_{u14} \vee \neg x_{v15}$ будет препятствовать тому, чтобы за станциями u и v закреплялись каналы 14 и 15. Этот список помеховых ограничений устраняет второе упрощающее допущение из раздела 24.2.1.

Еще одна загвоздка заключалась в том, что не все станции имели право на закрепление всех каналов. Например, станции, граничащие с Мексикой, не могут быть отнесены к каналу, который будет мешать станции на мексиканской стороне границы. Чтобы отразить дополнительные ограничения, переменная x_{vi} решения опускалась всякий раз, когда станция v запрещалась каналом i.

Эти изменения в исходной формулировке задачи SAT (24.1)–(24.3) иллюстрируют общую мощь решателей задач MIP и SAT относительно специфичных алгоритмов: они хорошо приспосабливаются ко всем видам ограничений и требуют минимальных изменений базовой формулировки.

24.3.3. Задача о переупаковке

Задача о проверке допустимости на обратном аукционе стимулирующего аукциона FCC почти представляла собой задачу о раскраске графа, но не совсем (из-за ограничений из раздела 24.3.2), поэтому назовем ее *задачей о переупаковке*.

ЗАДАЧА: ЗАДАЧА О ПЕРЕУПАКОВКЕ

Заранее известны: список V телевизионных станций, разрешимые каналы C_v для каждой станции $v \in V$ и разрешимые пары каналов P_{uv} для каждой пары станций $u, v \in V$.

Вход: подмножество $S \subseteq V$ телевизионных станций.

Выход: закрепление за каждой станцией $v \in S$ канала в C_v чтобы за каждой парой станций $u, v \in S$ закреплялась пара каналов в P_{uv}. (Либо объявление, что такого закрепления не существует.)

Назовем подмножество станций *пригодным для упаковки*, если соответствующий экземпляр переупаковки имеет допустимое решение, и *непригодным для упаковки* в противном случае.

Алгоритмические устремления FCC были амбициозными: надежно решить задачу о переупаковке за минуту или меньше! (В разделе 24.4 мы увидим, почему временной бюджет был таким малым.) Экземпляры переупаковки имели тысячи станций, десятки тысяч пар перекрывающихся станций и десятки каналов. После преобразования в задачу SAT (как в разделах 24.3.1–24.3.2) результирующие экземпляры имели десятки тысяч переменных принятия решений и более миллиона ограничений.

Это реально много! Но почему бы не впустить туда самые лучшие решатели задач SAT и не посмотреть, как они справятся? К сожалению, при применении «прямо с полки» этим решателям часто требовалось десять минут или больше на то, чтобы решить репрезентативные экземпляры задачи о переупаковке. Более скорые результаты требовали, чтобы в задаче задействовались все имеющиеся ресурсы и никак иначе.

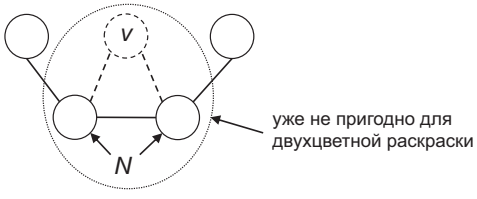
24.3.4. Трюк № 1: предварительные решатели (в поисках легкого выхода)

Стимулирующий аукцион FCC использовал *предварительные решатели*, чтобы быстро выуживать экземпляры, пригодные для упаковки, с помощью вложенной структуры экземпляров переупаковки в алгоритме `FCCGreedy` (см. тестовое задание 24.1). Каждый экземпляр принимал форму $S \cup \{v\}$ для пригодного для упаковки множества станций S и новой станции v.

Например, аукцион провел два быстрых и черновых локальных теста, которые рассмотрели только (относительно малую) окрестность станции v. Формально назовем две станции *соседями*, если они появляются вместе хотя бы в одном помеховом ограничении (24.4), и обозначим через $N \subseteq S$ соседей станции v в S.

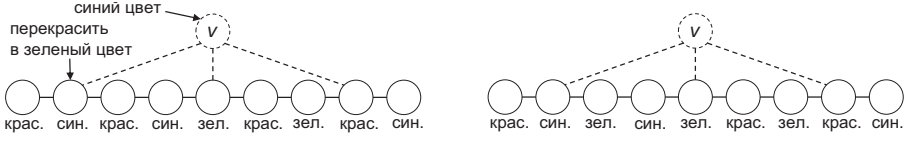
1. Проверка, пригодно или нет $N \cup \{v\}$ для упаковки. Если нет, то остановка и сообщение «не пригодно для упаковки». (Надмножества непригодных для упаковки множеств станций сами являются непригодными для упаковки.)

Аналогом в экземпляре раскраски графа была бы проверка, образует ли вершина v и ее соседи подграф, пригодный для k-цветной раскраски. Например (при $k = 2$):



2. Наследование ранее вычисленных допустимых закреплений каналов для пригодных для упаковки станций S. Закрепления всех станций в $S - N$ фиксированы. Проверка, есть ли закрепления каналов для станций в $N \cup \{v\}$, допускающие комбинированные закрепления. Если да, то сообщение «пригодное для упаковки» и возврат комбинированных закреплений каналов.

Успешность завершения этого шага, как правило, зависит от унаследованных закреплений каналов для станций в $S - N$. Например (при $k = 3$):



локальное перекрашивание для получения трехцветной раскраски

локальное перекрашивание не работает

Размер окрестности N обычно выражался однозначными или двузначными цифрами, поэтому каждый из этих шагов можно было выполнить быстро с помощью решателя задач SAT. Неоднозначность сохранялась для экземпляров переупаковки, прошедших через оба шага. Такой экземпляр мог бы быть пригодным для упаковки вследствие допустимого закрепления каналов, отклоняющегося от ограниченной формы, рассмотренной на шаге 2. Либо он мог быть непригодным для упаковки, без упаковки $N \cup \{v\}$ на шаге 1, расширяемой до одной из всех $S \cup \{v\}$.

24.3.5. Трюк № 2: предварительная обработка и упрощение

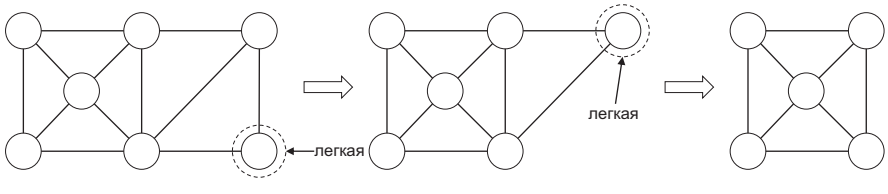
Каждый экземпляр переупаковки, который пережил предварительные решатели, подвергался шагу предварительной обработки, созданному для снижения размера.[1]

Удаление легких станций

Назовем станцию u из $S \cup \{v\}$ легкой, если, независимо от закрепления каналов за другими станциями, за станцией u может быть закреплен канал из C_u, который избегает помех с соседями. (Аналогом в экземпляре раскраски графа будет вершина, степень которой меньше числа k цветов.)

3. Итеративное удаление легких станций включает: (1) инициализацию $X := S \cup \{v\}$, (2) до тех пор пока X содержит легкую станцию u, $X := X - \{u\}$.

Например, в контексте раскраски графа (при $k = 3$):



ТЕСТОВОЕ ЗАДАНИЕ 24.2

Итеративное удаление легких станций из множества $S \cup \{v\}$...

а) ...могло бы изменить статус множества с пригодного для упаковки на непригодное или наоборот.

---

[1] Эта идея близка по духу к «бесплатным примитивам», подчеркиваемым во всех книгах этой серии. Если у вас есть ослепительно быстрый примитив (например, сортировка, вычисление связных компонент и т. д.), который может упростить вашу задачу, то почему бы им не воспользоваться?

б) ...могло бы изменить статус множества с пригодного для упаковки на непригодное, но не наоборот.

в) ...могло бы изменить статус множества с непригодного для упаковки на пригодное, но не наоборот.

г) ...не могло бы изменить статус пригодности множества для упаковки.

(Ответ и анализ решения см. в разделе 24.3.8.)

Декомпозиция задачи

Следующим шагом была декомпозиция задачи на более мелкие независимые подзадачи. (Аналогом в экземпляре раскраски графа было бы вычисление k-цветной раскраски отдельно для каждой связной компоненты.)

4. При наличии множества (нелегких) станций:

   а) Формирование графа H с вершинами, соответствующими станциям, и ребрами к соседним станциям.

   б) Вычисление связных компонентов графа H.

   в) Для каждой связной компоненты решение соответствующей задачи о переупаковке.

   г) Если по меньшей мере одна подзадача не пригодна для упаковки, сообщение «непригодна для упаковки». В противном случае — сообщение «пригодна для упаковки» и возврат объединения закреплений каналов, вычисленных в подзадачах.

Поскольку станции создают помехи только соседним станциям, разные подзадачи никак не взаимодействуют. Следовательно, станции в X пригодны для упаковки, если и только если все независимые подзадачи пригодны для упаковки.

Почему помогло разложение задачи? Стимулирующий аукцион FCC был в щекотливой ситуации, отвечая за решение всех подзадач, совокупный размер которых был таким же, как и у исходной задачи. Но алгоритм, который

выполняется за сверхлинейное время (как у решателя задач SAT), быстрее решает экземпляр по частям, чем весь сразу.[1]

24.3.6. Трюк № 3: портфель решателей задач SAT

Самые сложные экземпляры переупаковки прошли предварительные решатели и предварительную обработку и дожидались более изощренных инструментов. Хотя даже самый современный решатель задач SAT имел успех в некоторых репрезентативных экземплярах, ни один из них не соответствовал желанию FCC надежно решить экземпляры за минуту или меньше. Что же дальше?

Разработчики обратного аукциона на стимулирующем аукционе FCC воспользовались двумя преимуществами: (1) эмпирическим наблюдением, что разные решатели задач SAT с трудом справляются на разных экземплярах, и (2) современными компьютерными процессорами. Вместо того чтобы складывать все свои яйца в одну корзину с одним решателем задач SAT, аукцион использовал портфель из *восьми* тщательно настроенных решателей, работающих параллельно на 8-ядерном рабочем месте.[2, 3] Такая мера, наконец, предоставила достаточную алгоритмическую огневую мощь, чтобы решить более 99 % экземпляров переупаковки, с которыми аукцион сталкивался в пределах поставленной цели, равной одной минуте каждая. Довольно

---

[1] Например, рассмотрим квадратично-временной алгоритм, работающий за время cn^2 на n-размерных экземплярах для некоторой константы $c > 0$.Тогда решение двух $(n/2)$-размерных экземпляров занимает время, равное $2 \times c(n/2)^2 = cn^2/2$, что в 2 раза ускоряет решение одного n-размерного экземпляра.

[2] И как эти восемь решателей были отобраны? С жадным эвристическим алгоритмом, аналогичным алгоритмам для задач о максимальном охвате (раздел 20.2) и максимизации влияния (раздел 20.3)! Решатели выбирались последовательно, причем каждый решатель максимизировал предельное улучшение времени выполнения на репрезентативных экземплярах по сравнению с решателями, уже находящимися в портфеле.

[3] Для фанатов локального поиска (разделы 20.4–20.5), расстроенных его очевидным отсутствием в этом тематическом исследовании: несколько решателей задач SAT в этом портфеле были локально-поисковыми алгоритмами; подумайте о более жадных и сильно параметризованных версиях рандомизированного алгоритма выполнимости, описанного в задаче 21.13.

впечатляюще для экземпляров задачи SAT с десятками тысяч переменных и более чем миллионом ограничений!

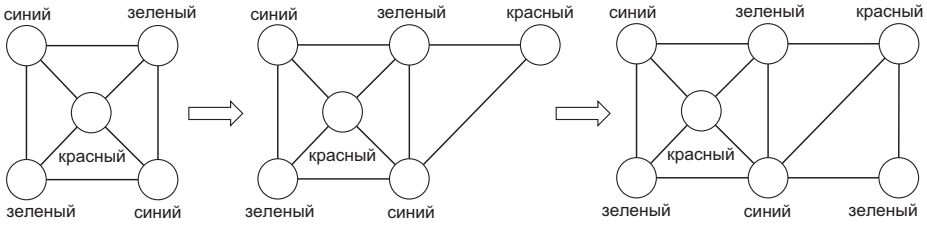
24.3.7. Терпимость к отказам

Более 99 % звучит прекрасно, но что происходило в оставшийся 1 % времени? Неужели стимулирующий аукцион FCC беспомощно пробуксовывал, в то время как восемь решателей задач SAT возились вокруг, отчаянно нуждаясь в удовлетворяющем закреплении канала за станцией?

Еще одной особенностью алгоритма `FCCGreedy` является его терпимость к отказам благодаря подпрограмме проверки допустимости. Предположим, что при проверке допустимости множества $S \cup \{v\}$ подпрограмма берет тайм-аут и сообщает «я не знаю». Без обеспечения гарантии допустимости (что является железным ограничением) алгоритм не может рисковать добавлением v в свое решение и должен его пропускать, потенциально отказываясь от некоторой стоимости, которую он мог бы получить в противном случае. Но алгоритм всегда завершается через предсказуемый промежуток времени с допустимым решением, и потеря стоимости от тайм-аутов была скромной, при условии что они происходили редко.

24.3.8. Решение к тестовому заданию 24.2

Правильный ответ: (г). Если финальное множество X не пригодно для упаковки, то таким же является и надмножество $S \cup \{v\}$. Если X пригодно для упаковки, то каждое допустимое закрепление каналов за станциями множества X может быть расширено до всех из $S \cup \{v\}$, по одной легкой станции за раз (в обратном порядке удаления):



24.4. Реализация в виде нисходящего тактового аукциона

И где же тут «аукцион» в стимулирующем аукционе FCC? Разве алгоритм `FCCGreedy` в разделе 24.2 наряду с подпрограммой переупаковки в разделе 24.3 уже не решает задачу о максимизации стоимости, почти достигая оптимальности? Имея не более нескольких тысяч проверок допустимости (по одной на участвующую станцию) и одну минуту, затрачиваемую на проверку допустимости, алгоритм будет завершен в течение нескольких дней. Не пора ли объявить о победе?

Нет. Вместо этого самое время пересмотреть и устранить последние два упрощающих допущения из раздела 24.2.1. Станции не удалялись из эфира принудительно: они добровольно отказывались от своих лицензий (в обмен на компенсацию). Так почему бы не выполнить алгоритм `FCCGreedy`, чтобы выяснить, какие станции должны оставаться в эфире, и выкупить другие станции по любой цене, которую они будут готовы принять? Потому что стоимость станции, определяемая здесь как минимальная компенсация, которую ее владелец примет за уход из эфира, не была известна заранее. (Владельцы, вероятно, преувеличат стоимость своих станций.) Как реализовать алгоритм `FCCGreedy` без предварительного знания о стоимостях станций?

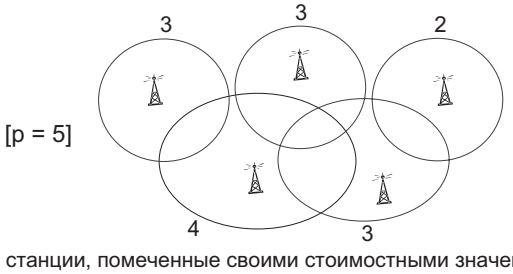
24.4.1. Аукционы и алгоритмы

Вспомните аукционы, которые вы видели в кино или в реальной жизни — например, на распродаже недвижимости, в аукционном доме или на школьном благотворительном вечере. Аукционист задает вопросы по форме: «Кто готов купить этот теннисный мяч, подписанный Роджером Федерером, за сто долларов?», и желающие совершить такую операцию покупатели поднимают руки. На обратном аукционе стимулирующего аукциона FCC «аукционист» (правительство) покупал, а не продавал, поэтому вопросы имели форму «кто готов продать свою лицензию на вещание за миллион долларов?». Ответ на этот вопрос со стороны станции с предложенной компенсацией p показывал, была ли ее стоимость — минимальная приемлемая компенсация — выше или ниже p.

Алгоритм FCCGreedy начинается с сортировки станций в неувеличивающемся порядке значений w_v/β_v, где w_v — стоимость станции v, и β_v — станционно-специфичный параметр, кажущийся безнадежным, когда стоимости станций неизвестны.[1] Можно ли переосмыслить алгоритм так, чтобы станции сортировали себя, используя только удобные для аукциона операции формы «верно ли, что $w_v \leq p$»?

24.4.2. Пример

Чтобы увидеть, как это может работать, допустим, что стоимости станций являются положительными целыми числами между 1 и известной верхней границей W и есть только один свободный канал ($k = 1$), а $\beta_v = 1$ для каждой станции v. Например, есть пять станций и $W = 5$:

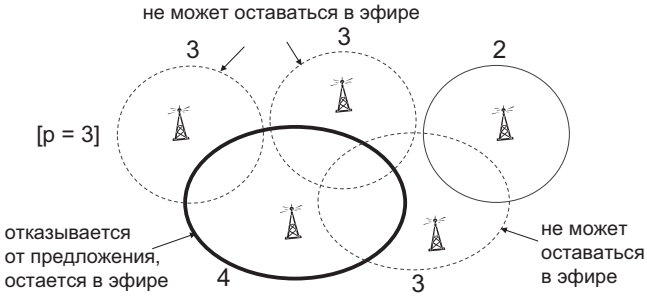


станции, помеченные своими стоимостными значениями

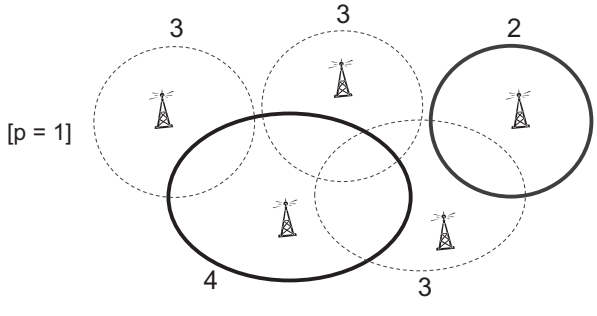
Начнем с самой высокой мыслимой компенсации ($p = W$) и будем двигаться вниз. Множество S станций, которые останутся в эфире, изначально является пустым. На первой итерации алгоритма стоимость каждой станции сравнивается с первым значением p (то есть с 5). Таким же образом каждому вещателю задается вопрос, согласится ли он принять компенсацию в размере 5 в обмен на свою лицензию. Все участники соглашаются, и алгоритм уменьшает p и переходит к следующей итерации. Все участники снова принимают предложение о сниженной компенсации (при $p = 4$). На следующей итерации

[1] В стимулирующем аукционе FCC станционноспецифичные параметры β_v *были* известны заранее, поскольку они зависели только от обслуживаемого станцией населения и помеховых ограничений станции (см. сноску 2 на с. 260).

станция со стоимостью 4 отказывается от предложения при $p = 3$, и алгоритм отвечает, добавляя ее в множество S эфирных станций:



Теперь, когда станция со стоимостью 4 вернулась в эфир и имеется только один канал, три перекрывающиеся станции заблокированы и должны оставаться вне эфира. На последующих итерациях алгоритм делает убывающие предложения компенсации единственной станции, судьба которой остается нерешенной, — станции со стоимостью 2. Эта станция отказывается от предложения при $p = 1$, в результате чего она добавляется в S, и алгоритмы останавливаются:



В этом примере и в целом этот итеративный процесс воссоздает траекторию алгоритма `WISBasicGreedy` на соответствующем экземпляре взвешенного независимого множества (раздел 24.2.3): станции выпадают из эфира (и возвращаются в эфир) в неубывающем порядке стоимости при условии допустимости. Можем ли мы расширить эту идею, чтобы запечатлеть полноценный алгоритм `FCCGreedy`?

24.4.3. Переосмысление жадного алгоритма FCCGreedy

Станционноспецифичные параметры β_v в алгоритме `FCCGreedy` могут быть эмулированы с помощью станционноспецифичных предложений, причем компенсация $\beta_v \times p$ будет предложена станции v на итерации с базовой ценой p. Станция v со стоимостью w_v выпадет, когда базовая цена p падает ниже w_v/β_v. По мере постепенного уменьшения p результирующий процесс точно симулирует жадный алгоритм `FCCGreedy`: станции выпадают из эфира (и возвращаются в эфир) в неубывающем порядке значения w_v/β_v при условии допустимости. Результирующий алгоритм называется *нисходящим тактовым аукционом*,[1] и это именно тот алгоритм, который используется в обратном аукционе стимулирующего аукциона FCC (при $\epsilon = 0{,}05$ и β_v, определенном как в сноске 2 на с. 260).

FCCDESCENDINGCLOCK

Вход: множество V станций, параметр $\beta_v > 0$ для каждого $v \in V$, параметр $\epsilon \in (0, 1)$.

Выход: пригодное для переупаковки подмножество $S \subseteq V$.

$p :=$ КРУПНОЕ ЧИСЛО `// максимизировать участие`
$S := \emptyset$ `// станции, остающиеся в эфире`
$X := \emptyset$ `// станции, уходящие из эфира`
while $S \cup X \neq V$ **do** `// станции все еще в подвешенном состоянии`
  for each станция $v \notin S \cup X$, в произвольном порядке **do**
    `// вызвать подпрограмму проверки допустимости (раздел 24.3)`

---

[1] Нисходящий тактовый аукцион (descending clock auction) начинается с высокого уровня поддержки, который постепенно снижается. Участники торгов выбывают по мере того, как уровни поддержки становятся для них неприемлемо низкими, до тех пор пока оставшихся участников торгов не будет достаточно, чтобы заполнить объем аукциониста. В нисходящем тактовом аукционе, таким образом, цена последнего выбывшего участника торгов (самая низкая отклоненная заявка) будет определять цену для всех вознаграждаемых проектов. — *Примеч. пер.*

```
    if S ∪ {v} пригодно для упаковки then // все еще есть
                                          место для v
      предложить компенсацию β_v · p каналу v
      if предложение отклонено then // поскольку p < w_v/β_v
        S := S ∪ {v}   // v возвращается в эфир
    else                    // нет места для v (или тайм-аута)
      X := X ∪ {v}              // v должен остаться вне эфира
  p := (1 − ϵ) · p   // более низкие предложения на следующем
                     раунде
return S
```

Внешний цикл алгоритма нисходящего тактового аукциона `FCCDescendingClock` управляет значением p «тактовых часов». Эта базовая цена уменьшается на небольшую величину на каждой итерации (именуемой *раундом*), до тех пор пока судьба всех станций не будет решена. В течение раунда внутренний цикл выполняет один проход над остальными станциями в произвольном порядке с намерением сделать более низкое предложение для каждой. Но прежде чем сделать предложение станции v, алгоритм обращается к подпрограмме проверки допустимости из раздела 24.3, чтобы убедиться, что v может быть размещена в эфире, если она отклонит предложение.[1] Когда подпрограмма проверки допустимости обнаруживает, что $S \cup \{v\}$ не пригодна для упаковки или ее тайм-аут истекает, то алгоритм не рискует отказом v и берет на себя обязательство держать v вне эфира. Когда подпрограмма проверки допустимости возвращается с допустимой упаковкой станций в $S \cup \{v\}$, то алгоритм безопасно продолжает работу с более низким предложением к v. Он возвращает окончательное множество станций, которые останутся в эфире, вместе с закреплениями каналов, вычисленными для них подпрограммой проверки допустимости.

[1] Оригинальный алгоритм `FCCGreedy` из раздела 24.2.4 вызывает подпрограмму проверки допустимости только один раз в расчете на участвующую станцию. Переосмысленная версия, алгоритм `FCCDescendingClock`, требует новой серии проверок допустимости в каждом раунде. Обратный аукцион стимулирующего аукциона FCC проходил в течение десятков раундов, требуя в общей сложности около ста тысяч проверок допустимости. Вот почему FCC предоставила только одну минуту на проверку допустимости. (И даже с одной минутой тайм-аута аукциону потребовалось много месяцев на то, чтобы довести свою работу до конца.)

24.4.4. Пора получить компенсацию

Алгоритм `FCCDescendingClock` определяет станции, остающиеся в эфире, и их новые закрепления каналов. У него есть еще одна обязанность: вычислять цены, уплачиваемые уходящим вещателям в обмен на их лицензии. (Вещатели, которые остаются в эфире, не получают никакой компенсации.)[1]

Прежде всего, каковой является начальная базовая цена p? На стимулирующем аукционе FCC эта стоимость выбиралась таким образом, чтобы первые предложения были абсурдно прибыльными и побуждали многие станции принимать участие (участие было добровольным). Например, первое предложение для WCBS, филиала CBS в Нью-Йорке, составляло 900 миллионов долларов![2] Каждая вещательная компания, которая участвовала в аукционе, была по контракту обязана продать свою лицензию по первому предложению, если правительство попросит об этом, и схожим образом для каждого последующего (более низкого) предложения, принятого в ходе аукциона. Правительство, естественно, оплачивало самую низкую согласованную цену.

> **КОМПЕНСАЦИЯ НА СТИМУЛИРУЮЩЕМ АУКЦИОНЕ FCC**
>
> Каждому вещателю, уходящему из эфира, выплачивалось самое последнее (и отсюда самое низкое) предложение, которое он принимал на аукционе.

Несмотря на всю свою сложность под капотом, обратный аукцион стимулирующего аукциона FCC был чрезвычайно прост для участников. Первое предложение за лицензию было известно заранее, и каждое последующее предложение автоматически составляло 95 % от предыдущего. До тех пор пока текущее предложение превышало стоимость, ожидаемую вещателем, очевидным шагом было его принятие (так вещатель мог отклонять более

---

[1] Технически станции, вынужденные переключить каналы после проведения стимулирующего аукциона FCC, получали скромную сумму денег — гораздо меньшую, чем цена продажи лицензии, — чтобы покрыть расходы на переключение.

[2] И помните, что продажа лицензии означала только отказ от наземного вещания — мелкой рыбешки по сравнению с кабельным и спутниковым телевидением.

низкие предложения). Как только текущее предложение падало ниже стоимости лицензии, очевидным ответом было отклонить предложение и вернуться в эфир (поскольку любые последующие предложения были еще хуже).

Эффективность подпрограммы проверки допустимости (раздел 24.3) оказала первостепенное влияние на расходы правительства. Никакого вреда, никакого фола, когда тайм-аут подпрограммы истекал с непригодным для упаковки множеством станций, — алгоритм `FCCDescendingClock` действовал так, как и должен был. Но всякий раз, когда тайм-аут подпрограммы проверки допустимости истекал на пригодном для упаковки множестве станций $S \cup \{v\}$, на столе оставалась куча денег — часто в миллионах долларов.[1] Аукцион мог бы сделать станции v более низкое предложение, если бы не отказ подпрограммы проверки его допустимости. Теперь вы понимаете, почему создатели аукциона хотели добиться успеха подпрограммы, как можно более близкого к 100 %![2]

24.5. Окончательный результат

Стимулирующий аукцион FCC проходил примерно год, с марта 2016 по март 2017 года. Было задействовано почти три тысячи телевизионных станций, 175 из которых решили уйти из эфира в обмен на суммарную компенсацию примерно в десять миллиардов долларов (в среднем около пятидесяти миллионов долларов за лицензию, с большой разницей в разных регионах страны)[3]. Примерно тысяча станций переназначили свои каналы.

Между тем 84 МГц освобожденного спектра были реорганизованы в семь пар блоков по 5 МГц (один блок для закачивания данных на станцию, другой для скачивания на приставку). Каждая из всего пула лицензий, выставленных на продажу на прямом аукционе стимулирующего аукциона FCC, соответствовала одной из семи пар и одному из 416 регионов США (именуемых

---

[1] Почти 50 % тайм-аутов приходились на пригодное для упаковки множество станций.

[2] Существовала ли когда-нибудь более прямая связь между временем работы алгоритма и огромными суммами денег?

[3] Полный список результатов по адресу https://auctiondata.fcc.gov.

частичными экономическими зонами). И какой же была выручка от этого прямого аукциона? Двадцать миллиардов долларов![1] Основная часть прибыли использована для сокращения дефицита бюджета США.[2]

Стимулирующий аукцион FCC имел ошеломляющий успех, и он никогда бы не состоялся без передового алгоритмического инструментария для решения NP-трудных задач — того самого инструментария, который вы можете теперь, после того как проявили настойчивость, прочитав эту книгу до конца, заявить, как свой собственный.

ВЫВОДЫ

- ✶ Стимулирующий аукцион FCC представлял собой сложный алгоритм, который закупал 84 МГц беспроводного спектра, используемого для наземного телевидения, и выкупал его для беспроводных сетей следующего поколения.

---

[1] Хорошо, что выручка от прямых аукционов превысила закупочные расходы на обратных аукционах — неужели правительству просто повезло? Это относится к еще одному вопросу: кто решил, что 84 МГц — это идеальная величина спектра для очистки? Фактически стимулирующий аукцион FCC имел дополнительный внешний цикл, который осуществлял нисходящий поиск идеального числа каналов для очистки (это одна из причин, почему аукцион занял так много времени). На своей первой итерации (именуемой стадией) аукцион амбициозно попытался освободить двадцать один канал (126 МГц), достаточный для создания десяти парных лицензий в расчете на регион для продажи на прямом аукционе. (Среди двадцати одного канала это были 30–36 и 38–51; как отмечается в сноске на с. 254, канал 37 был закрыт.) Эта стадия потерпела провал, причем закупочные расходы составили примерно 86 млрд долларов, а выручка от прямых аукционов — только около 23 млрд долларов. Аукцион перешел ко второй стадии с уменьшенной целью очистки девятнадцати каналов (114 МГц, достаточной для девяти парных лицензий в расчете на регион), возобновив обратный и прямой аукционы там, где они остановились на первой стадии. Аукцион в конечном счете остановился после четвертой стадии (очистив четырнадцать каналов, как описано в этой главе) — первой стадии, на которой его выручка покрыла его расходы.

[2] Сокращение дефицита было запланировано с самого начала — вероятно, это было одной из главных причин, по которой законопроект удалось принять Конгрессу (см. сноску на с. 255).

- ✶ Обратный аукцион решал то, какие телевизионные станции уйдут из эфира или переключат каналы, и вычислял размер компенсации для них.
- ✶ Даже при наличии только одного канала задача поиска наиболее ценных станций, не создающих друг другу помех и остающихся в эфире, сводится к NP-трудной задаче о взвешенном независимом множестве.
- ✶ При наличии многочисленных каналов простая проверка, может ли множество станций оставаться в эфире без помех, сводится к NP-трудной задаче о раскраске графа.
- ✶ На репрезентативных экземплярах тщательно настроенный жадный эвристический алгоритм надежно возвращал решения с почти оптимальной суммарной стоимостью.
- ✶ Каждая итерация жадного алгоритма вызывала подпрограмму проверки допустимости, чтобы узнать, есть ли для текущей станции место в эфире.
- ✶ Для реализации этого алгоритма использовался нисходящий тактовый аукцион, при котором с течением времени величина предложений компенсации падала, и станции выпадали из торгов.
- ✶ Используя предварительные решатели, предварительную обработку и портфель из восьми современных решателей задач SAT, FCC менее чем за минуту решила более 99 % экземпляров проверки допустимости.
- ✶ Стимулирующий аукцион FCC продолжался в течение года, удалил 175 станций из эфира и получил почти 10 млрд долларов прибыли.

Задачи на закрепление материала

Задача 24.1 *(S)*. Какой из алгоритмических инструментов, описанных в главах 20 и 21, не сыграл никакой роли в стимулирующем аукционе FCC?

а) Жадные эвристические алгоритмы.

б) Локальный поиск.

в) Динамическое программирование.

г) Решатели задач MIP и SAT.

Задача 24.2 *(S)*. В каждом раунде алгоритма `FCCDescendingClock` в разделе 24.4.3 станции, все еще находящиеся в подвешенном состоянии, обрабатывались в произвольном порядке. Было ли множество S станций, возвращаемых алгоритмом, независимым от порядка, используемого в каждом раунде? (Правильных ответов может быть несколько.)

а) Да, при условии что никакие две стоимости w_v станций не совпадали и все параметры β_v были установлены равными 1.

б) Да, при условии что все параметры β_v были установлены равными 1 и были достаточно малы.

в) Да, при условии что никакие две стоимости w_v станций не совпадали, все параметры β_v были установлены равными 1 и были достаточно малы.

г) Да, при условии что никакие два соотношения w_v/β_v не были одинаковыми и достаточно малыми.

Задача 24.3 *(S)*. Прежде чем сделать станции v более низкое по стоимости предложение, алгоритм `FCCDescendingClock` проверяет, годится или нет множество станций $S \cup \{v\}$ для упаковки, где S обозначает уже находящиеся в эфире станции. Предположим, что мы изменили порядок этих двух шагов:

```
предложить компенсацию β_v × p станции v
if предложение отклонено then  // потому что p < w_v/β_v
    if S ∪ {v} packable then   // место для v
        S := S ∪ {v}           // v возвращается в эфир
    else                       // нет места для v
        X := X ∪ {v}           // v должен оставаться вне эфира
```

Представьте, что мы предлагаем уходящим вещателям компенсацию, как на с. 277, причем каждому оплачивается сумма в соответствии с последним предложением, которое он принял (предпоследнее предложение, которое он получил). Правда ли, что вещатель должен принимать каждое предложение выше его стоимости и отклонять первое предложение ниже его стоимости?

Задача повышенной сложности

Задача 24.4 *(H)*. Эта задача расследует качество решения, достигаемое жадными эвристическими алгоритмами `WISBasicGreedy` и `WISGeneralGreedy` из раздела 24.2.3 для частного случая задачи о взвешенном независимом множестве, в которой степень $\deg(v)$ каждой вершины v равна самое большее Δ (где Δ — это неотрицательное целое число, такое как 3 или 4).

а) Докажите, что независимое множество, возвращаемое базовым жадным алгоритмом `WISBasicGreedy`, всегда имеет суммарный вес по меньшей мере в $1/(\Delta + 1)$ раз больше суммарного веса всех вершин входного графа.

б) Докажите, что та же гарантия справедлива для общего жадного алгоритма `WISGeneralGreedy` с β_v, установленным равным $1 + \deg(v)$ для каждой вершины $v \in V$.

в) Покажите на примерах, что для каждого неотрицательного целого числа утверждения в (а) и (б) становятся ложными, если $1/(\Delta + 1)$ заменяется любым более крупным числом.

Задача по программированию

Задача 24.5. Попробуйте один или несколько решателей задач SAT на коллекции экземпляров задачи о раскраске графов, используя формулировку (24.1)–(24.3). (Поэкспериментируйте как с ограничениями, описанными в (24.3), так и без них.) Например, расследуйте случайные графы, где каждое ребро присутствует независимо с некоторой вероятностью $p \in (0, 1)$. Либо, что еще лучше, произведите граф из фактических помеховых ограничений, используемых в стимулирующем аукционе FCC.[1] Насколько большим будет размер входных данных, который решатель способен надежно обработать менее чем за минуту или менее чем за час? Насколько сильно ответ зависит от решателя?

[1] Доступны по адресу https://data.fcc.gov/download/incentive-auctions/Constraint_Files/.

Эпилог: полевое руководство по разработке алгоритмов

Прочитав серию «Совершенный алгоритм», вы обладаете богатым алгоритмическим инструментарием, годящимся для решения широкого спектра вычислительных задач. Он на самом деле богат, но огромное число алгоритмов, структур данных и парадигм проектирования выглядит пугающим. Когда вы сталкиваетесь с новой задачей, какой будет самый эффективный способ заставить ваши инструменты работать? Чтобы дать вам отправную точку, я расскажу о типичном рецепте, который сам использую, когда нужно разобраться в незнакомой вычислительной задаче. По мере того как вы будете накапливать все больше алгоритмического опыта, разработайте свой индивидуальный рецепт.

1. 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 ис-

[1] Если вы перейдете к более глубокому изучению алгоритмов, то узнаете о более четких решениях задач, которые появляются в замаскированном виде. Несколько примеров включают быстрое преобразование Фурье, задачи о максимальном потоке и минимальном разрезе, двудольном сочетании и задачи линейного и выпуклого программирования.

пользуйте самый быстрый и простой алгоритм, достаточный для решения задачи.

2. Можно ли упростить задачу, предварительно обработав входные данные с помощью бесплатного примитива, такого как сортировка или вычисление связных компонент?

3. Если нужно разработать новый алгоритм с нуля, откалибруйте свою работу, определив черту на песке, проведенную «очевидным» решением (например, исчерпывающим поиском). Для входных данных, которые вас интересуют, является ли очевидное решение уже достаточно быстрым?

4. Если очевидного решения недостаточно, то проведите мозговой штурм, задействовав как можно большее число естественных жадных алгоритмов, и протестируйте их на малых примерах. Скорее всего, все провалится, но вы лучше поймете задачу.

5. Если есть очевидный способ разделить входные данные на меньшие подзадачи, то насколько легко будет объединить их решения? Если вы видите, как это сделать быстро, 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 малого числа способов? Можете ли вы сформулировать рекуррентное уравнение, чтобы быстро решить подзадачу, имея решения меньших подзадач?

7. Сможете ли вы улучшить очень хороший алгоритм посредством ловкого развертывания структур данных? Ищите важные вычисления, которые алгоритм выполняет снова и снова (например, операции поиска или вычисления минимума). Помните принцип бережливости: выбрать простейшую структуру данных, поддерживающую все операции, требуемые вашим алгоритмом.

8. Можете ли вы сделать свой алгоритм проще или быстрее, используя рандомизацию? Например, если ваш алгоритм должен выбирать один объект из многих, что произойдет, когда он будет делать выбор случайно?

9. Если все предыдущие шаги оказываются безуспешными, подумайте о непредвиденной, но реальной возможности того, что для вашей задачи не существует эффективного алгоритма. Какая NP-трудная задача

наиболее близка к вашей? Можно ли свести NP-трудную задачу к вашей? Что насчет задачи 3-SAT или других задач из книги Гэри и Джонсона (с. 193)?

10. Решите, хотите ли вы пойти на компромисс в отношении правильности или скорости. Если вы предпочитаете сохранять гарантированную скорость, повторите парадигмы проектирования алгоритмов в поисках возможности для быстрых эвристических алгоритмов. Парадигма дизайна жадных алгоритмов выделяется как наиболее часто используемая для этой цели.

11. Рассмотрите также парадигму локального поиска, как для приближенного решения задачи с нуля, так и для шага постобработки без недостатков, чтобы привязать ее к какому-то другому эвристическому алгоритму.

12. Если вы предпочитаете настаивать на гарантированной правильности, вернитесь к парадигме динамического программирования и отыщите точные алгоритмы лучше исчерпывающего поиска (но, вероятно, попрежнему экспоненциально-временные).

13. Если динамическое программирование неприменимо либо ваши алгоритмы динамического программирования являются слишком медленными, то скрестите пальцы и поэкспериментируйте с полунадежными волшебными ящиками. В случае задачи оптимизации попробуйте сформулировать ее как смешанную целочисленную программу и натравите на нее решатель задач MIP. В случае задачи о проверке допустимости вместо этого начните с формулировки выполнимости и натравите на нее решатель задач SAT.

Подсказки и решения

Решение задачи 19.1: (б), (в). Алгоритм динамического программирования для (NP-трудной) задачи о рюкзаке является хорошим примером того, почему (г) является неправильным.

Решение задачи 19.2: (в). Сноска на с. 21 показывает, почему (а) является неправильным. Все остовные деревья графа могут иметь разные суммарные стоимости (например, если реберные стоимости являются разными степенями 2), поэтому (б) также является неправильным. Логика в (г) ошибочна, так как задача о минимальном остовном дереве легко решается даже в графах с экспоненциальным числом остовных деревьев.

Решение задачи 19.3: (б), (г). Ответ (в) неверный, потому что полиномиально-временная решаемость не связана с решаемостью на практике. (Представьте алгоритм с временем выполнения, например, $O(n^{100})$ на n-размерных входных данных.)

Решение задачи 19.4: (а). Например, алгоритм динамического программирования для задачи о рюкзаке показывает, что (в) и (г) являются неправильными.

Решение задачи 19.5: (д). Для (а) и (б) редукция идет в неверном направлении. Ответ (в) неверный, потому что некоторые задачи (например, задача об остановке, упомянутая в сноске 1 на с. 38) строго труднее, чем другие (например, задача о минимальном остовном дереве). Ответ (г) является неправильным, когда, например, (а) и (б) являются задачами о кратчайшем пути с одним истоком и для всех пар вершин. Формальное доказательство для (д) напоминает решение к тестовому заданию 19.3.

Решение задачи 19.6: (а), (б), (в). В (а) вы можете без потери общности допустить, что вместимость рюкзака C составляет самое большее n^6 (почему?). Для (б) обратитесь к задаче 20.11. Для (в) задача является NP-трудной, даже когда входные данные содержат только положительные целые числа (с. 41).

Подсказка для задачи 19.7. Чтобы использовать подпрограмму для задачи коммивояжера и решить экземпляр задачи о пути коммивояжера, нужно добавить одну дополнительную вершину, соединенную нуль-стоимостным ребром с каждой из исходных вершин. Сначала следует разделить произвольную вершину v на две копии, v' и v'' (где каждая наследует реберные стоимости от v и $c_{v'v''} = +\infty$). Затем следует добавить две новые вершины x, y, каждая из которых соединена со всеми другими вершинами бесконечно-стоимостными ребрами, за исключением того, что $c_{xv'} = c_{yv''} = 0$.

Подсказка для задачи 19.8. Посетить вершины графа G в том же порядке, в каком поиск сначала в глубину (из произвольной стартовой вершины) посещал бы вершины тура T. Доказать, что суммарная стоимость результирующего тура равна $2\sum_{e \in E'} a_e$ и что ни один тур не может иметь меньшую суммарную стоимость.

Решение задачи 20.1: (б). Чтобы сфальсифицировать (а), рассмотрим десять машин, десять работ длиной 1, девяносто работ длиной 6/5 и одну работу длиной 2. Чтобы доказать (б), используйте допущения, показывающие, что максимальная длина работы не превышает 20 % от средней машинной загрузки, и подключите это в (20.3).

Решение задачи 20.2: (б). Чтобы сфальсифицировать (а), используйте 16-элементный вариант примера из тестового задания 20.5. Оптимальное решение должно использовать два подмножества жадного алгоритма пять (с разрывом связей для наихудшего случая). Для (б) первые k итераций жадного алгоритма совпадают с такими же итерациями алгоритма `GreedyCoverage` с бюджетом k. Из гарантии приближенной правильности для последнего алгоритма (теорема 20.7) следует, что этот первый пакет из k итераций охватывает по меньшей мере $1 - 1/e$ долю элементов в U. Следующий пакет из k итераций охватывает по меньшей мере $1 - 1/e$ долю элементов, которые не были охвачены в первом пакете (почему?). После t серий из k итераций каждая число все еще неохваченных элементов не превышает $(1/e)^t \times |U|$. Это число меньше 1 при $t > \ln |U|$, поэтому алгоритм завершается в течение $O(k \log |U|)$ итераций.

Решение задачи 20.3: (в), (д), (е). Чтобы сфальсифицировать (а) и (г), нужно взять $C = 100$ и рассмотреть десять предметов со значением 2 и размером 10 вместе с сотней предметов со значением 1 и размером 1. Чтобы сфальсифицировать (б), рассмотрите один предмет с размером и значением, равным 100, и второй предмет со значением 20 и размером 10. Чтобы доказать (в), позвольте второму жадному алгоритму обманным путем заполнить рюкзак полностью, используя долю одного дополнительного предмета (со значением, заработанным на пропорциональной основе). Примените аргумент с обменом, чтобы доказать, что суммарное значение этого мошеннического решения по крайней мере равно значению любого допустимого решения. Аргументируйте, что совокупное значение решений, возвращаемых первыми двумя жадными алгоритмами, по меньшей мере равно значению мошеннического решения (и, следовательно, лучшее из двух будет как минимум на 50 % таким же). Чтобы доказать (д) и (е), аргументируйте, что второй жадный алгоритм пропускает только наихудшие 10 % (с точки зрения отношения значения к размеру) мошеннического решения.

Решение задачи 20.4: (а). Каждая итерация главного цикла `while` алгоритма выбирает одно ребро входного графа. Обозначьте через M множество выбранных ребер. Тогда возвращаемое алгоритмом подмножество S будет содержать $2|M|$ вершин. Никакие два ребра M не имеют общей конечной точки (почему?), поэтому каждое допустимое решение должно включать в себя по меньшей мере $|M|$ вершин (одна конечная точка в расчете на ребро из M).

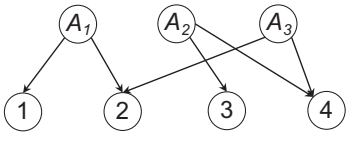
Решение задачи 20.5: (в). Алгоритм локального поиска в конечном итоге останавливается в локально оптимальном решении.

Подсказка для задачи 20.6. Храните в куче по одному объекту на машину с ключами, равными текущим машинным загрузкам. Каждое обновление машинной загрузки сводится к операции `ExtractMin` с последующей операцией `Insert` с обновленным значением ключа.

Подсказка для задачи 20.7. Для (а) можно игнорировать любые работы после работы j (почему?). Доказать, что, если $\ell_j > M^*/3$, за каждой машиной закрепляется одна или две первых j работ, причем самые длинные работы выполняются на их собственных машинах, а остальные оптимально объединяются в пары на остальных машинах. Для (б) используйте (20.3).

Подсказка для задачи 20.8. Для (а) используйте $k^{k-1} \times k^{k-1}$-решетку элементов и $2k - 1$ подмножеств. Для (б) замените каждый элемент группой из N его копий (каждая из которых принадлежит к тем же подмножествам, что и раньше). Устраните связи, добавив одну дополнительную копию в некоторые группы. Выбор N зависит от ϵ.

Подсказка для задачи 20.9. Например, имея экземпляр задачи о максимальном охвате с бюджетом k, универсальным множеством $U = \{1, 2, 3, 4\}$ и подмножествами $A_1 = \{1, 2\}$, $A_2 = \{3, 4\}$ и $A_3 = \{2, 4\}$, закодируйте его с использованием ориентированного графа



с активационной вероятностью $p = 1$ и тем же бюджетом k.

Подсказка для задачи 20.10. Для (а) проверьте свойства непосредственно для функций охвата, а затем используйте лемму 20.10. Для (б) первостепенным ингредиентом будет общая версия лемм 20.8 и 20.11, а именно неравенства (20.7) и (20.15). Обозначьте через S оптимальное решение, и через S_{j-1} — первые $j - 1$ объектов, выбранных жадным алгоритмом. Можно рассмотреть правые стороны этих неравенств как суммы поочередных предельных значений объектов в $S^* - S_{j-1}$, когда они добавляются в S_{j-1} один за другим в произвольном порядке. Левые части выражают сумму предельных значений объектов в $S^* - S_{j-1}$, когда каждый из них добавляется в S_{j-1} изолированно. Из субмодулярности следует, что каждый член в первой сумме не превышает член во второй. Где в доказательстве проявляются неотрицательность и монотонность?

Подсказка для задачи 20.11: Для (а) каждая подзадача вычисляет для некоторых $i \in \{0, 1, 2, ..., n\}$ и $x \in \{0, 1, 2, ..., n \times v_{max}\}$ минимальный суммарный размер подмножества первых i элементов, имеющих суммарное значение по меньшей мере x (либо $+\infty$, если такого подмножества не существует). Полное решение см. в бонусных видеороликах по адресу www.algorithmsilluminated.org.

Подсказка для задачи 20.12. Для (а) каждый тур можно рассматривать как гамильтонов путь (который, как остовное дерево, имеет суммарную стоимость,

не превышающую суммарную стоимость минимального остовного дерева) вместе с одним дополнительным ребром (которое, по принятому допущению, имеет неотрицательную стоимость). Для (б) используйте неравенство треугольника, чтобы доказать, что все реберные стоимости в построенном древовидном экземпляре задачи коммивояжера по меньшей мере так же велики, как и в данном метрическом экземпляре задачи коммивояжера. Используя решение задачи 19.8, заключите, что суммарная стоимость вычисленного тура не более чем в два раза превышает стоимость тура T минимального остовного дерева.

Подсказка для задачи 20.13. Представьте граф с использованием матрицы смежности (в которой записи кодируют реберные стоимости) и текущий тур — с использованием двусвязного списка.

Решение задачи 20.14. Для (а) значение целевой функции всегда является целым числом от 0 до $|E|$, которое увеличивается хотя бы на 1 на каждой итерации. Для (б) рассмотрите локальный максимум. Для каждой вершины $v \in S_i$ и группы S_j, где $j \neq i$, число ребер между v и вершинами группы S_i не превышает число между v и вершинами группы S_j (почему?). Сложение этих $|V| \times (k - 1)$ неравенств и перестановка завершают аргументацию.

Решение задачи 21.1: (в).

Решение задачи 21.2. Со столбцами, индексированными вершинами в $V - \{a\}$, и строками, индексированными подмножествами S, которые содержат a и, по меньшей мере, одну другую вершину:

$\{a, b\}$	1	N/A	N/A	N/A
$\{a, c\}$	N/A	4	N/A	N/A
$\{a, d\}$	N/A	N/A	5	N/A
$\{a, e\}$	N/A	N/A	N/A	10
$\{a, b, c\}$	6	3	N/A	N/A
$\{a, b, d\}$	11	N/A	7	N/A
$\{a, b, e\}$	13	N/A	N/A	4
$\{a, c, d\}$	N/A	12	11	N/A
$\{a, c, e\}$	N/A	18	N/A	12

$\{a, d, e\}$	N/A	N/A	19	14
$\{a, b, c, d\}$	14	13	10	N/A
$\{a, b, c, e\}$	15	12	N/A	9
$\{a, b, d, e\}$	17	N/A	13	14
$\{a, c, d, e\}$	N/A	22	21	20
$\{a, b, c, d, e\}$	23	19	18	17
	b	c	d	e

Решение задачи 21.3: (б). Добавление ребра (w, v) в самый дешевый путь P с $(i - 1)$ переходами из 1 в w создает цикл, если P уже посещает v.

Решение задачи 21.4: (а), (б), (в), (г), (д).

Решение задачи 21.5. Со столбцами, индексированными вершинами, и строками, индексированными непустыми подмножествами цветов:

$\{R\}$	0	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$\{G\}$	$+\infty$	$+\infty$	0	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$\{B\}$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0	0	$+\infty$	$+\infty$
$\{Y\}$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0	0
$\{R, G\}$	1	4	1	4	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$\{R, B\}$	2	6	$+\infty$	$+\infty$	6	2	$+\infty$	$+\infty$
$\{R, Y\}$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$\{G, B\}$	$+\infty$	$+\infty$	7	3	7	3	$+\infty$	$+\infty$
$\{G, Y\}$	$+\infty$	$+\infty$	8	5	$+\infty$	$+\infty$	5	8
$\{B, Y\}$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	9	10	9	10
$\{R, G, B\}$	5	7	3	5	8	3	$+\infty$	$+\infty$
$\{R, G, Y\}$	9	9	$+\infty$	$+\infty$	$+\infty$	$+\infty$	9	9
$\{R, B, Y\}$	12	15	$+\infty$	$+\infty$	$+\infty$	$+\infty$	15	12
$\{G, B, Y\}$	$+\infty$	$+\infty$	16	13	14	8	8	13
$\{R, G, B, Y\}$	10	17	13	19	15	11	10	11
	a	b	c	d	e	f	g	h

Подсказка для задачи 21.6. Любая вершина j, достигшая минимума в (21.6), появляется последней в некотором оптимальном туре. Вершина k, достигшая минимума в (21.4), непосредственно предшествует j на таком туре. Остальная часть тура может быть реконструирована схожим образом в обратном порядке. Чтобы достичь линейного времени выполнения, измените алгоритм `BellmanHeldKarp` так, чтобы он кэшировал для каждой подзадачи вершину, достигшую минимума, в рекуррентном уравнении (21.5), используемом для вычисления решения подзадачи.

Подсказка для задачи 21.7. Измените алгоритм `PanchromaticPath` так, чтобы он кэшировал для каждой подзадачи ребро (w, v), достигшее минимума в рекуррентном уравнении (21.7), используемом для вычисления решения подзадачи. Кроме того, кэшируйте вершину, достигшую минимума в последней строке псевдокода.

Подсказка для задачи 21.8. Выбросите решения s-размерных подзадач после вычисления всех решений $(s + 1)$-размерных подзадач. Используйте аппроксимацию Стирлинга (21.1) для оценивания $\binom{n}{n/2}$.

Решение задачи 21.9. (а) с x_v, указывающим, включена или нет вершина v в решение:

$$\begin{aligned} &\text{минимизировать} \sum_{v \in V} w_v x_v \\ &\text{при условии, что } x_u + x_v \le 1 \quad [\text{для каждого ребра } (u,v) \in E] \\ &\qquad\qquad x_v \in \{1, 0\} \quad [\text{для каждой вершины } v \in V] \end{aligned}$$

(б) с x_{ij}, указывающим, закреплена ли работа j за машиной i, и с M, обозначающим соответствующую производственную продолжительность плана:[1]

$$\begin{aligned} &\text{минимизировать } M \\ &\text{при условии, что } \sum_{j=1}^{n} \ell_j x_{ij} \le M \quad [\text{для каждой машины } i] \end{aligned}$$

---

[1] Если вас беспокоят ограничения с переменными решения с обеих сторон, то перепишите их как $\sum_{j=1}^{n} \ell_j x_{ij} - M \le 0$ для каждой машины i. Эти ограничения заставляют M быть не меньше максимальной машинной загрузки. В любом оптимальном решении задачи MIP равенство должно соблюдаться (почему?).

$$\sum_{i=1}^{m} x_{ij} = 1 \quad \text{[для каждого задания } j\text{]}$$
$$x_{ij} \in \{0,1\} \quad \text{[для каждой машины } i \text{ и задания } j\text{]}$$
$$M \in \mathbb{R}$$

(в) с x_i, указывающим, входит ли подмножество A_i в решение, и y_e, указывающим, принадлежит ли элемент e выбранному подмножеству:[1]

$$\text{минимизировать } \sum_{e \in U} y_e$$
$$\text{при условии, что } y_e \le \sum_{i : e \in A_i} x_i \quad \text{[для каждого элемента } e \in U\text{]}$$
$$\sum_{i=1}^{m} x_i = k$$
$$x_i, y_e \in \{0,1\} \quad \text{[для каждого подмножества } A_i \text{ и элемента } e\text{]}$$

Подсказка для задачи 21.10. Для (а) пустите тур в одном направлении и установите x_{ij} равным 1, если j является непосредственным преемником i, и в противном случае — 0. Для (б) покажите, что объединение двух (или более) непересекающихся ориентированных циклов, которые вместе посещают все вершины, транслируется в допустимое решение задачи MIP. Для (в), если ребро (i, j) является ℓ-м переходом тура (начиная с вершины 1), установите $y_{ij} = n - \ell$. Для (г) аргументируйте тем, что каждое допустимое решение имеет форму, построенную в (в).

Подсказка для задачи 21.11. Закодируйте ограничение $x_1 \vee \neg x_2 \vee x_3$ как $y_1 + (1 - y_2) + y_3 \ge 1$, где y_i — это переменные решения 0–1. (Используйте целевую функцию-местозаполнитель, например константу 0.)

Подсказка для задачи 21.12. Предварительно обработайте экземпляр задачи 2-SAT так, чтобы каждое ограничение имело ровно два литерала. (Один хак заключается в замене ограничения типа x_i двумя ограничениями, $x_i \vee z$ и $x_i \vee \neg z$, где z — это только что добавленная переменная решения. Более практичное итеративное устранение однолитеральных ограничений приводит к присвоению переменной значения, которое может быть распространено на

[1] Первое множество ограничений вынуждает $y_e = 0$ всякий раз, когда ни одно из подмножеств, содержащих e, не выбрано. (И если такое подмножество выбрано, то y_e равно 1 в каждом оптимальном решении.)

другие ограничения, связанные с ней.) Когда остаются только двухлитеральные ограничения, ключевой трюк заключается в вычислении сильно связных компонент соответствующего ориентированного графа (что можно сделать за линейное время; см. главу 8 *Второй части*). Вы находитесь на ориентированном пути, если ваш граф имеет $2n$ вершин (по одной на литерал) и $2m$ ориентированных ребер. Данный экземпляр допустим, если и только если каждый литерал находится в другой компоненте, чем его противоположность.

Подсказка для задачи 21.13. Для (б) вспомните (21.10). Для (в) используйте факт, что *ta** удовлетворяет ограничению, а *ta* — нет. Для (г) вспомните, что присвоение истинности и его противоположность одинаково вероятны. Для (е) используйте тот факт, что граница времени выполнения формы $O\left((\sqrt{3})^n n^d \ln\frac{1}{\delta}\right)$ для константы d также равна $O((1{,}74)^n \ln 1/\delta)$ (поскольку любая экспоненциальная функция растет быстрее любой полиномиальной функции).

Решение задачи 22.1: (г). Задача о неориентированном гамильтоновом пути сводится к каждой задаче в (а)–(в). Задача в (г) может быть решена за полиномиальное время с использованием вариации алгоритма кратчайшего пути Беллмана — Форда (см. главу 18 *Третьей части*).

Решение задачи 22.2: (а), (б). Задачи в (а) и (б) сводятся к задаче о кратчайшем пути для вершин без отрицательных циклов (для (б) после умножения всех длин ребер на –1), которая может быть решена за полиномиальное время с использованием алгоритма Флойда — Уоршелла (см. главу 18 *Третьей части*). Задача об ориентированном гамильтоновом пути сводится к задаче в (в), доказывая, что последняя (и более общая задача в (г)) является NP-трудной.

Решение задачи 22.3: (а), (б), (в), (г). Для (б), если предполагаемая подпрограмма для версии решения говорит «нет», то сообщите «решения нет». Если говорит «да», то используйте подпрограмму повторно, чтобы удалить исходящие ребра из s, но не удаляйте ребро, которое перевернет ее ответ на «нет». В итоге останется только одно исходящее ребро (s, v). Повторите процесс из v. Для (г) выполните двоичный поиск целевой суммарной стоимости C. Время выполнения будет полиномиальным по числу вершин и числу цифр, необходимых для представления реберных стоимостей, то есть полиномиальным по размеру входных данных; см. также обсуждение на с. 41.

Подсказка для задачи 22.4. Переключайте ребра, которые присутствуют или отсутствуют.

Подсказка для задачи 22.5. Подмножество вершин S является вершинным покрытием, если и только если его дополнение $V - S$ является независимым множеством.

Подсказка для задачи 22.6. Используйте одно подмножество на вершину, содержащее инцидентные ей ребра.

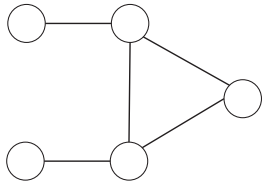
Подсказка для задачи 22.7. Используйте t в качестве вместимости рюкзака и a_i — в качестве как стоимости предмета, так и размера предмета i.

Решение задачи 22.8. Вызовите подпрограмму для задачи о максимальном охвате с заданной системой множеств и последовательно увеличивающимися бюджетами $k = 1, 2, ..., m$. В первый раз, когда подпрограмма возвращает k подмножеств, охватывающих все множество U, эти подмножества составляют оптимальное решение для данного экземпляра задачи о покрытии множества.

Подсказка для задачи 22.9. Чтобы свести неориентированную версию к ориентированной, замените каждое неориентированное ребро (v, w) двумя ориентированными ребрами (v, w) и (w, v). Для другого направления выполните следующую операцию на каждой вершине:

Подсказка для задачи 22.10. Для (а) добавьте во входные данные еще одно дополнительное число. Для (б) используйте значения a_i в качестве длин работ.

Подсказка для задачи 22.11. Начните с треугольника на вершинах, именуемых t («истинная»), f («ложная») и o («другая»). Добавьте еще две вершины v_i, w_i в расчете на переменную x_i в данный экземпляр задачи 3-SAT и соедините их треугольником с помощью o. В каждой трехцветной раскраске v_i и w_i имеют те же цвета, что t и f (значит, x_i := истина), либо те же цвета, что f и t (значит, x_i := ложь). Реализуйте дизъюнкцию двух литералов с помощью подграфа формы



где «входы» находятся слева и «выход» — справа. Объедините два подграфа, чтобы реализовать дизъюнкцию трех литералов.

Подсказка для задачи 22.12. Часть (б) вытекает из редукции в доказательстве теоремы 22.7. Для части (а) добавьте 1 в каждую реберную стоимость в этой редукции.

Решение задачи 23.1: (а), (б). Для (а), насколько нам известно, задача коммивояжера может быть решена за полиномиальное время. Для (б), насколько нам известно, $P \neq NP$, но гипотеза об экспоненциальном времени является ложной. Для (в) см. задачу 23.5. Для (г), если любая NP-полная задача поддается решению за полиномиальное время, то $P = NP$ и все такие задачи поддаются решению за полиномиальное время.

Решение задачи 23.2. Поскольку задача коммивояжера является NP-трудной (теорема 22.7), каждая задача в NP сводится к ней. Следовательно, если предположение Эдмондса является ложным и для задачи коммивояжера существует полиномиально-временной алгоритм, то и предположение, что $P \neq NP$, тоже является ложным. И наоборот, если $P = NP$, то поисковая версия задачи коммивояжера (которая принадлежит NP) будет поддаваться решению за полиномиальное время. Оптимизационная версия задачи коммивояжера сводится к поисковой версии бинарным поиском (задача 22.3) и тогда также будет поддаваться решению за полиномиальное время, опровергая предположение Эдмондса.

Решение задачи 23.3. Все варианты (убедитесь сами).

Подсказка для задачи 23.4. Для (а) составьте редукции. Для (в) соедините в цепь два препроцессора и два постпроцессора. Чтобы наложить границу на время выполнения, рассуждайте как в решении к тестовому заданию 19.3.

Решение задачи 23.5: Для (а) существует редукция Левина из задачи 3-SAT к ее дополненной версии (добавьте одну новую переменную и соответству-

ющее дополнение). Для (б) проверьте (за линейное время), дополнены ли входные данные, и если да, то примените исчерпывающий поиск, чтобы вычислить удовлетворяющее присвоение истинности или заключить, что его не существует. Поскольку размер N дополненного экземпляра равен по меньшей мере n^2, этот исчерпывающий поиск выполняется за время $2^{O(n)} = 2^{O(\sqrt{N})}$.

Подсказка для задачи 23.6. Попробуйте трюк из предыдущей задачи с суперполиномиальным, но субэкспоненциальным количеством дополнений. Покажите, что полиномиально-временной алгоритм для дополненной задачи опровергнет гипотезу об экспоненциальном времени. Используя тот факт, что дополненная задача может быть решена за субэкспоненциальное время (почему?), докажите, что редукция Кука из задачи 3-SAT к дополненной такой задаче также опровергнет гипотезу об экспоненциальном времени.

Подсказка для задачи 23.7. Для (а) выполните поиск сначала в ширину n раз, по одному разу для каждого варианта выбора стартовой вершины. Для (б) разделите n переменных экземпляра задачи k-SAT на две группы размером $n/2$ каждая. Введите по одной вершине для каждого из $2^{n/2}$ возможных присвоений истинности переменным в первой группе, а также для второй группы. Обозначьте два множества $2^{n/2}$ вершин через A и B. Введите по одной вершине для каждого из m ограничений вместе с двумя дополнительными вершинами s и t. Обозначьте это множество $m + 2$ вершин через C и определите $V = A \cup B \cup C$. (Вопрос: насколько большим может быть m в зависимости от n и k?) Включите ребра между каждой парой вершин в C, между s и каждой вершиной в A и между t и каждой вершиной в B. Завершите множество ребер E, соединив вершину v из A или B с вершиной w, соответствующей ограничению, если и только если ни одно из $n/2$ присвоений значений переменным, кодируемых вершиной v, не удовлетворяет ограничению, соответствующему вершине w. Докажите, что диаметр графа $G = (V, E)$ равен либо 3, либо 2, в зависимости от того, является ли экземпляр задачи k-SAT выполнимым или невыполнимым.

Решение задачи 24.1: (в).

Решение задачи 24.2: (в), (г). Если в каждом раунде алгоритма `FCCDescendingClock` есть одна станция v, все еще находящаяся в подвешенном состоянии, которая откажется от предложения этого раунда (потому что в первый раз w_v превышает $\beta_v \times p$), то порядок не влияет на то, какие станции остаются

в эфире (почему?).[1] Когда соотношения станций w_v/β_v различны, соблюдение этого условия можно обеспечить путем взятия достаточно малых значений ϵ. Поэтому ответы (в) и (г) являются правильными. Если две станции готовы выпасть в одном и том же раунде — из-за связи между станционными коэффициентами w_v/β_v или из-за того, что ϵ является недостаточно малым, — разные упорядочения приведут их на выходе к разным результатам (убедитесь сами). Поэтому ответы (а) и (б) неверные.

Решение задачи 24.3. Не обязательно, поскольку вещатель может в некоторых случаях разыграть систему, отклонив предложение выше его стоимости и получив бóльшую компенсацию, чем в противном случае. (Например, если владелец станции v, находящейся в подвешенном состоянии, узнает, что множество $S \cup \{v\}$ стало непригодным для упаковки, то он должен отклонить следующее предложение.)

Подсказка для задачи 24.4. Для (а) всякий раз, когда алгоритм включает v в свое текущее решение S, он выбивает из дальнейшего рассмотрения самое большее Δ других вершин, вес каждой из которых не превышает w_v. Поэтому $\sum_{v\in S} w_v \le \sum_{v\in S} \Delta \times w_v$, из чего вытекает заявленная граница. В (б) для $v \in S$ обозначьте через $X(v)$ вершины, выбываемые из дальнейшего рассмотрения включением станций v в S, то есть $u \in X(v)$, если v является первым соседом u, добавленным в S, или если u сам является v. В силу жадного критерия алгоритма, всякий раз, когда он включает v в S, $w_v \ge \sum_{u\in X(v)} w_u / (\deg(u)+1)$. Поскольку каждая вершина $u \in V$ принадлежат множеству $X(v)$ ровно для одной вершины $v \in S$ (почему?),

$$\sum_{v\in S} w_v \ge \sum_{v\in S} \sum_{u\in X(v)} \frac{w_u}{\deg(u)+1} = \sum_{u\in I} \frac{w_u}{\deg(u)+1} \ge \frac{\sum_{u\in I} w_u}{\Delta+1}.$$

[1] Хотя даже в этом случае компенсация, выплачиваемая станции, уходящей из эфира, может зависеть от порядка обработки (почему?).

Книги Тима Рафгардена

Начальный уровень

Совершенный алгоритм. Основы / Пер. с англ. А. Логунова. — СПб.: Питер, 2019. — 253 с.: ил., табл. — (Библиотека программиста).

Совершенный алгоритм. Графовые алгоритмы и структуры данных / Пер. с англ. А. Логунова. — СПб.: Питер, 2019. — 255 с.: ил., табл.

Совершенный алгоритм. Жадные алгоритмы и динамическое программирование. — СПб.: Питер, 2020. — 256 с.: ил.

Совершенный алгоритм. Алгоритмы для NP-трудных задач (Algorithms Illuminated, Part 4: Algorithms for NP-Hard Problems, Soundlikeyourself Publishing, 2020).

Промежуточный уровень

Двадцать лекций по алгоритмической теории игр (Twenty Lectures on Algorithmic Game Theory, Cambridge University Press, 2016).

Коммуникационная сложность (для дизайнеров алгоритмов) (Communication Complexity (for Algorithm Designers), NOW Publishers, 2016).

Продвинутый уровень

Эгоистичный маршрут и цена анархии (Selfish Routing and the Price of Anarchy, MIT Press, 2005).

Алгоритмическая теория игр (Algorithmic Game Theory (ed.), co-editors: Noam Nisan, Éva Tardos, and Vijay V. Vazirani, Cambridge University Press, 2007).

Теория сложности, теория игр и экономика: барбадосские лекции (Complexity Theory, Game Theory, and Economics: The Barbados Lectures, NOW Publishers, 2020).

За пределами анализа алгоритмов для наихудшего случая (Beyond the Worst-Case Analysis of Algorithms (ed.), Cambridge University Press, 2020).

Тим Рафгарден

Совершенный алгоритм. Алгоритмы для NP-трудных задач

Перевел с английского *А. Логунов*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*А. Руденко*
Художественный редактор	*В. Мостипан*
Корректоры	*М. Молчанова, Г. Шкатова*
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: 12.2020. Наименование: книжная продукция. Срок годности: не ограничен.

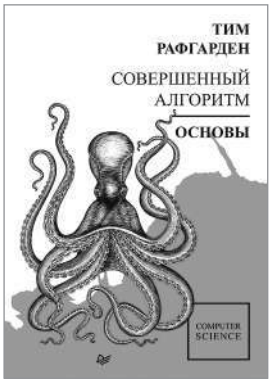
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

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 18.12.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 1000. Заказ 0000.

Тим Рафгарден

СОВЕРШЕННЫЙ АЛГОРИТМ. ОСНОВЫ



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 этой книге Тим Рафгарден — гуру алгоритмов — расскажет об асимптотическом анализе, нотации большое-О, алгоритмах «разделяй и властвуй», рандомизации, сортировки и отбора.

Книга «Совершенный алгоритм» адресована тем, у кого уже есть опыт программирования. 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.

Тим Рафгарден

СОВЕРШЕННЫЙ АЛГОРИТМ. ГРАФОВЫЕ АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Во второй книге Тим Рафгарден — гуру алгоритмов — расскажет о графовом поиске и его применении, алгоритме поиска кратчайшего пути, а также об использовании и реализации некоторых структур данных: куч, деревьев поиска, хеш-таблиц и фильтра Блума.

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.

КУПИТЬ

Тим Рафгарден

СОВЕРШЕННЫЙ АЛГОРИТМ. ЖАДНЫЕ АЛГОРИТМЫ И ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

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.

КУПИТЬ