

Эффективные алгоритмы и сложность вычислений

Н. Н. Кузюрин

С. А. Фомин

12 июня 2011 г.

Оглавление

Предисловие	6
Введение	8
1 Алгоритмы и их сложность	15
1.1 Примеры задач и алгоритмов	15
1.1.1 Теоретико-числовые задачи: «НОД», «факториал», «возвведение в степень», «дискретный логарифм»	15
1.1.2 Задачи на графах: «Коммивояжер», «Кратчайшие пути», «Остовные деревья»	22
1.1.3 Приближенные алгоритмы: «Составление расписаний»	40
1.1.4 «Сортировка слиянием»	44
1.1.5 «Быстрая сортировка»	47
1.2 Формально об алгоритмах. Несложно о сложности	54
1.2.1 «RAM»: машины с произвольным доступом	54
1.2.2 Сложность в худшем случае	61

1.2.3	Сложность в среднем	63
1.2.4	Полиномиальные алгоритмы	63
1.2.5	Полиномиальность и эффективность	67
2	Аппроксимация с гарантированной точностью	70
2.1	Алгоритмы с оценками точности	70
2.1.1	Жадные алгоритмы для «Покрытия множеств»	72
2.1.2	Приближенные алгоритмы для «Вершинного покрытия»	77
2.1.3	Жадный алгоритм для «Рюкзака»	88
2.1.4	Алгоритм Кристофида	93
2.2	Аппроксимация с заданной точностью	100
2.2.1	«Рюкзак»: динамическое программирование	100
2.2.2	Полностью полиномиальная приближенная схема для «Рюкзака»	110
3	Вероятностный анализ детерминированных алгоритмов	118
3.1	Сложность и полиномиальность в среднем	118
3.2	Задача упаковки	121
3.3	Выполнимость КНФ	128
3.4	Точность алгоритма для почти всех входов	139
3.5	«Рюкзак»: полиномиальность в среднем	145
4	Вероятностные алгоритмы и их анализ	157
4.1	Вероятностная проверка тождеств	157
4.2	Вероятностные методы в перечислительных задачах	162

4.3	Вероятностные методы в параллельных вычислениях	169
4.3.1	Максимальное по включению независимое множество в графе	169
4.3.2	Протокол византийского соглашения	180
4.4	Вероятностное округление	184
4.4.1	Вероятностное округление для задачи «MAX-SAT»	184
4.4.2	Максимальный разрез в графе	191
5	Методы дерандомизации	203
5.1	Метод условных вероятностей	203
5.2	Метод малых вероятностных пространств	210
5.3	Полиномиальная проверка простоты	215
6	Основы теории сложности вычислений	228
6.1	Сложность вычислений	228
6.1.1	Машины Тьюринга и вычислимость	228
6.1.2	Классы $DTIME$, $DSPACE$	248
6.2	Полиномиальные сводимости и \mathcal{NP} -полнота	252
6.2.1	Сводимость по Куку	253
6.2.2	Недетерминированные алгоритмы	255
6.2.3	Сводимость по Карпу	261
6.3	Вероятностные вычисления	271
6.3.1	Классы $\mathcal{RP}/co\mathcal{RP}$. «Односторонние ошибки»	275
6.3.2	Класс \mathcal{BPP} . «Двусторонние ошибки»	282

6.3.3	Класс \mathcal{PP}	288
6.3.4	Класс \mathcal{ZPP} . «Алгоритмы без ошибок»	291
6.3.5	Вероятностно проверяемые доказательства	293
6.3.6	\mathcal{PCP} и неаппроксимируемость	302
6.3.7	Класс \mathcal{APX} . Сводимости, сохраняющие аппроксимации	309
6.4	Схемы и схемная сложность	320
6.5	Коммуникационная сложность	329
6.6	Диаграмма классов сложности	334
7	Приложения	337
7.1	Введение в Python	337
7.2	Глоссарий	343
Предметный указатель	345	
Списки иллюстраций	353	
Список алгоритмов	356	

Предисловие

Настоящее учебное пособие написано по материалам двух спецкурсов, читавшихся авторами в течение нескольких лет для студентов 4-го и 6-го курсов Московского физико-технического института — «Сложность комбинаторных алгоритмов» и «Эффективные алгоритмы». Основная цель пособия заключается в ознакомлении читателей как с классическими результатами в разработке эффективных алгоритмов для решения вычислительно трудных задач, полученными еще в 1960–1970-х годах, так и с новыми результатами, полученными в последние годы. Именно в рассмотрении вычислительно трудных задач и современных подходов к их решению и заключается основное отличие данного пособия от традиционных книг по разработке и анализу эффективных алгоритмов (например: [АХУ79; КЛР99]).

Пособие можно разбить на три основные части соответственно уровням сложности изложения материала. В первой (глава 1) популярно излагаются примеры алгоритмов, и анализируется их сложность для ряда широко известных задач.

Вторая часть (главы 2–5) посвящена методам разработки и анализа алгоритмов решения конкретных задач. В главе 2 рассматриваются приближенные алгоритмы с гарантированными оценками точности для нескольких \mathcal{NP} -трудных задач. Глава 3 посвящена вероятностным методам анализа детерминированных алгоритмов. В ней демонстрируется, что для некоторых \mathcal{NP} -трудных задач существуют алгоритмы,

которые являются полиномиальными «в среднем» или точными «для почти всех входов». В главе 4 рассматриваются методы построения вероятностных приближенных алгоритмов. В главе 5 представлены методы дерандомизации — преобразования вероятностных алгоритмов в приближенные детерминированные. В этой части к результатам, которые не освещены в отечественной литературе, можно отнести полиномиальные в среднем алгоритмы для ряда \mathcal{NP} -трудных задач (например, для задачи о рюкзаке), вероятностные приближенные алгоритмы для ряда задач и методы их дерандомизации: метод условных вероятностей и метод малых вероятностных пространств.

Третья часть (глава 6) посвящена теории сложности, включающей в себя классические понятия сложности вычислений, классов сложности, теорию \mathcal{NP} -полноты. Из новых результатов, представленных в этой части, хочется отметить \mathcal{PCP} -теорему и ее следствия для доказательства неаппроксимируемости ряда задач, а также понятия сводимостей, сохраняющих аппроксимации.

Несколько слов о форме описания. Обычно при описании алгоритмов опускают вопросы конкретной программной реализации, и алгоритмы описываются либо неформально, либо на наиболее примитивном, алголоподобном языке (псевдокоде). Чтобы избежать неясностей, свойственных псевдоописаниям (т.к. псевдокод не «верифицирован» компьютером), мы реализовали часть алгоритмов на языке Python (см. раздел 7.1). Этот язык признан удачным выбором для преподавания информатики и курсов по алгоритмам.

Также в конце книги приведен глоссарий с элементарными определениями, используемыми в тексте книги. Эти определения обычно знакомы студентам старших курсов, поэтому мы не даем их в основном тексте книги, но иногда требуется освежить эти определения в памяти — для этого и предназначен глоссарий.

Введение

С понятием алгоритма были знакомы еще ученые древних цивилизаций. Алгоритм Евклида нахождения наибольшего общего делителя двух целых чисел был описан в книге VII «Начал» Евклида, датирующейся 330–320 гг. до н.э. Прообраз метода исключения Гаусса был описан в китайском источнике «Девять книг по арифметике» (202 г. до н.э. — 9 г.н.э.). Эратосфен (приблиз. 276–194 гг. до н.э.) предложил алгоритм проверки простоты числа, получивший впоследствии название решета Эратосфена.

Тем не менее, различные формализации понятия алгоритма были предложены только в середине 30-х годов прошлого столетия, когда и стала складываться современная теория алгоритмов. Ее возникновение связано с точным математическим определением такого, казалось бы, интуитивно ясного понятия как алгоритм. Потребность в этом определении была вызвана внутренними тенденциями развития математики, поскольку некоторые открытые проблемы заключались в выяснении существования (или несуществования) алгоритма для решения конкретных задач. Например, знаменитая 10-я проблема Гильберта заключалась в вопросе о существовании процедуры нахождения целочисленных корней произвольного многочлена от нескольких переменных с целыми коэффициентами¹.

Именно потенциальная возможность несуществования алгоритмов решения некоторых проблем по-

¹В 1970 году Ю.В. Матиясевич доказал, что эта проблема алгоритмически неразрешима.

требовала формального определения алгоритма. Почти одновременно в 30-х годах XX в. независимо (и в разных формах) рядом крупных математиков того времени — А. Тьюрингом ([Tur36]), А. Черчем ([Chu36]), Э. Постом ([Pos36]), К. Геделем, С. Клини и др. — было формализовано понятие вычислимости. Э. Пост и А. Тьюринг определили алгоритм как вычисление на абстрактных машинах. При этом вычислимость функции понималась как вычислимость на машинах Тьюринга. К. Гедель, А. Черч и С. Клини определили понятие вычислимости по-другому, на языке введенных ими арифметических функций, которые теперь называются частично рекурсивными функциями. Позднее А.А. Марков ввел понятие *нормального алгорифма*². Однако выяснилось, что все эти определения, совершенно различные по форме, описывают некое единое математическое понятие — *понятие алгоритма*.

Доказательство алгоритмической неразрешимости многих известных задач и получение ряда других отрицательных результатов послужило хорошим стимулом развития классической теории алгоритмов. По-видимому, не случайно эти исследования непосредственно предшествовали появлению первых компьютеров в 40-х годах прошлого века, поскольку теория алгоритмов явилась теоретическим фундаментом для создания и использования вычислительных машин. В рамках классической теории алгоритмов задаются вопросами о разрешимости различных задач, при этом вычислительная сложность алгоритмов принципиально не исследуется. Однако многие дискретные задачи, очевидно, алгоритмически разрешимы с помощью переборных алгоритмов. Такие прямые переборные алгоритмы уже при небольших исходных параметрах вынуждены просматривать огромное (обычно экспоненциальное) число вариантов. С практической же точки зрения часто нет никакой разницы между неразрешимой задачей и задачей, решаемой за экспоненциальное от длины входа время.

В связи с актуальными потребностями в анализе алгоритмов и задач с точки зрения вычислитель-

² Именно *алгорифмы Маркова*, а не *алгоритмы Маркова*. Так назвал их А.А.Марков, подчеркивая арабо-греческую этимологию слова. К тому же в дореволюционных российских энциклопедиях и учебниках использовалось слово *алгорифм*. В настоящее время слово *алгорифм* иногда используют математики так называемой «питерской школы».

ной сложности с 50-х годов XX века — с момента создания вычислительной техники — начала активно развиваться *теория сложности вычислений*. Параллельно и независимо от работ по теории сложности вычислений активно шли работы по построению и анализу конкретных алгоритмов для комбинаторных, теоретико-числовых и оптимизационных задач на графах и дискретных структурах. Постепенно сформировалось понятие «эффективного» алгоритма, под которым к 1970 г. стали понимать любой полиномиальный алгоритм, т. е. алгоритм, время выполнения которого ограничено некоторым полиномом от длины записи входных данных. Задачи, разрешимые такими алгоритмами, образуют класс, который стали обозначать через \mathcal{P} .

В начале 1970-х годов в работах С. Кука, Р. Карпа и Л. Левина была разработана математическая теория, строящаяся на основе фундаментальных понятий полиномиальной вычислимости и полиномиальной сводимости, которая объединила два достаточно независимых потока исследований: теоретико-сложностный и разработку и анализ алгоритмов для конкретных задач. В самом понятии сводимости не было ничего особенно нового, поскольку техника сводимостей является важным элементом классической теории алгоритмов. Новым было то, что полиномиально сводимыми друг к другу оказались многие известные практические задачи из различных областей.

Были введены недетерминированные вычисления, удачно моделирующие переборные алгоритмы, и определен класс \mathcal{NP} , в который попало большинство известных дискретных задач. Задачи, принадлежащие классу \mathcal{NP} , можно охарактеризовать как задачи, имеющие «короткое доказательство». Например, для знаменитой проблемы «Выполнимость», заключающейся в проверке наличия для формулы, заданной в виде к.н.ф., набора значений булевых переменных, на которых формула принимает значение «1», достаточно предъявить такой набор.

Было установлено существование «самых трудных» задач в классе \mathcal{NP} , названных \mathcal{NP} -полными. Это те задачи, к которым полиномиально сводится любая задача из класса \mathcal{NP} . Для таких задач существование точного эффективного алгоритма представляется крайне маловероятным, поскольку влечет равен-

ство $\mathcal{P} = \mathcal{NP}$. Классификация переборных задач на \mathcal{NP} -полные и те, которые поддаются решению с помощью точного эффективного алгоритма, оказалась весьма успешной — под нее попало подавляющее большинство дискретных задач.

Тем не менее, не все шло гладко в развитии теории \mathcal{NP} -полноты. До сих пор не удалось доказать, что в классе \mathcal{NP} есть задачи, которые не могут быть решены полиномиальными алгоритмами, или, иными словами, доказать несовпадение классов \mathcal{P} и \mathcal{NP} . В настоящее время это — одна из основных нерешенных проблем математики [Sma00]. Она связана с более общей проблемой получения *нижних оценок сложности вычислений*.

Здесь стоит отметить разницу между *сложностью алгоритма* и *сложностью алгоритмической задачи*. При анализе по худшему случаю получение *нижних оценок сложности конкретного алгоритма* почти всегда оказывается посильной задачей. Для этого достаточно подобрать набор входных данных, на которых алгоритм будет работать долго (именно так было доказано, что знаменитый симплекс-метод решения задачи линейного программирования не является полиномиальным алгоритмом).

Совсем другая ситуация со сложностью задач. Для получения *нижних оценок сложности задачи* требуется доказать, что *не существует алгоритма ее решения со сложностью не выше заданной*. Ввиду необозримого множества алгоритмов эта задача находится вне пределов возможностей современной теории сложности (или, по крайней мере, для ее решения нужны кардинально новые идеи).

Сформировавшаяся теория \mathcal{NP} -полноты выработала и ряд прагматических рекомендаций для исследователей, занимающихся решением прикладных задач. В тех случаях, когда интересующая разработчика практических алгоритмов задача оказывается \mathcal{NP} -полной, имеет смысл попробовать построить эффективный алгоритм для какой-либо ее модификации или частного случая, приемлемых с практической точки зрения. Когда не удается найти и такую модификацию, имеет смысл попробовать построить для задачи *приближенный* эффективный алгоритм, который гарантирует нахождение решения, отличающегося от оптимального не более чем в заданное число раз. Такие алгоритмы часто используются на практике.

Построение эффективных приближенных алгоритмов для решения переборных задач оказалось магистральным направлением, развивающимся с 1970-х гг. Было разработано огромное число эффективных приближенных алгоритмов для различных задач и получены оценки их точности. Все возрастающая роль, которую вероятностные методы стали играть в дискретной математике во второй половине XX века, в полной мере проявилась и в теории алгоритмов. Предложенные концепции вероятностных алгоритмов, использующих в своей работе результат «подбрасывания монеты» и, возможно, допускающих неправильные ответы с ограниченной вероятностью (меньшей единицы), оказались чрезвычайно продуктивными. Как следствие, в большинстве предметных областей вероятностные алгоритмы дают наилучшие приближенные решения для многих \mathcal{NP} -полных задач.

Говоря о приближенных алгоритмах, следует выделить вопрос о качестве полученных оценок их точности — для многих задач он в течение долгого времени оставался открытым. Другими словами, как быть, если полученная оценка точности далека от желаемой, а построить полиномиальный алгоритм с лучшими оценками точности не удается на протяжении десятилетий? Такая ситуация имела место для многих известных \mathcal{NP} -трудных задач: задачи о покрытии, задачи о максимальном разрезе, о нахождении максимальной клики в графе и многих других.

Только к середине 1990-х годов было доказано, что для многих задач полиномиальных алгоритмов с лучшими, чем известные, оценками точности не существует при стандартной гипотезе $\mathcal{P} \neq \mathcal{NP}$ (или сходных с ней). Результат, позволивший доказывать подобные теоремы — одно из самых ярких достижений теории сложности, получил название \mathcal{PCP} -теоремы (*Probabilistically Checkable Proofs*).

Это теорема, доказанная в начале 1990-х годов, произвела настолько сильное впечатление, что о ней писала даже нематематическая пресса (например, газета «New York Times»).

В неформальном изложении она формулируется следующим образом: для каждого языка из \mathcal{NP} существует доказательство принадлежности слов языку, которое можно проверить (в вероятностном смысле), просматривая лишь фиксированное число битов доказательства, не зависящее от его длины, и обес-

печить вероятность правильного ответа $1 - o(1)$.

Еще одним заметным достижением 1990-х годов явилось то, что удалось также классифицировать задачи по трудности их аппроксимации на основе сводимостей, сохраняющих приближения (L -сводимостей, E -сводимостей, AP -сводимостей и т. п.). Исследования возможностей построения эффективных приближенных алгоритмов для различных задач имеют длинную историю. С самого начала стало ясно, что хотя все \mathcal{NP} -полные задачи и полиномиально сводятся друг к другу, они могут иметь различную сложность с точки зрения построения эффективных приближенных алгоритмов. Причина кроется в том, что полиномиальные сводимости плохо приспособлены к сохранению аппроксимаций, и нужны дополнительные ограничения, позволяющие их обеспечить.

На разработку сводимостей, сохраняющих аппроксимации, ушли десятилетия. Был определен класс \mathcal{APX} оптимизационных задач, для которых существуют полиномиальные приближенные алгоритмы с константной мультипликативной точностью (для некоторой константы), и было установлено существование «самых трудных» задач в классе \mathcal{APX} , названных \mathcal{APX} -полными. Это те задачи, к которым сводится (с помощью AP -сводимости) любая задача из класса \mathcal{APX} . Удалось доказать \mathcal{APX} -полноту ряда задач относительно AP -сводимостей («максимальная выполнимость», «максимальный разрез в графе», «вершинное покрытие», метрическая задача коммивояжера на минимум и т. д.). Для таких задач разработка полиномиального приближенного алгоритма с наперед заданной точностью $1 + \varepsilon$ означает совпадение классов \mathcal{P} и \mathcal{NP} .

Возрастающее влияние вероятностных методов стало проявляться и в методах анализа алгоритмов. Был разработан альтернативный подход к решению \mathcal{NP} -трудных задач, отличный от построения приближенных алгоритмов с гарантированными (в худшем случае) оценками точности получаемого решения. Этот подход заключался в переходе к анализу сложности в среднем. Как известно, большинство задач дискретной оптимизации является \mathcal{NP} -трудным, и для них существование полиномиальных алгоритмов маловероятно. Несмотря на это, для многих исходных данных такие задачи бывают легко разрешимы на

практике. Пожалуй, наиболее известный пример этого феномена — симплекс-метод решения задач линейного программирования, который, не являясь полиномиальным алгоритмом, поразительно хорошо зарекомендовал себя при решении практических задач. Таким образом, вся трудность задачи может заключаться в относительно небольшом подмножестве входов. Проблема нахождения таких входов важна для экспериментальной оценки эффективности алгоритмов и в то же время играет заметную роль в математической криптографии. Концепция сложности в среднем для таких задач представляется более адекватной, чем концепция сложности в худшем случае.

Естественным образом сформировалось понятие сложности в среднем (*average case complexity*), под которым, грубо говоря, понимается математическое ожидание времени работы алгоритма при заданном вероятностном распределении на исходных данных. Хотя для ряда \mathcal{NP} -полных задач удалось построить полиномиальные в среднем алгоритмы, однако достигнутые здесь успехи оказались гораздо скромнее результатов в теории сложности при анализе по худшему случаю. Тем не менее, именно в этом направлении в последнее время получен ряд интересных результатов для задач типа рюкзака с константным числом ограничений, введено новое понятие «сглаженной сложности» (*smoothed complexity*) как разновидность сложности в среднем при специальном выборе вероятностного распределения на исходных данных и доказано, что некоторые варианты симплекс-метода имеют полиномиальную «сглаженную сложность».

В целом же можно констатировать, что разработка и исследование алгоритмов с привлечением теоретико-вероятностных методов переживает период бурного развития, и мы уделяем этому направлению значительную часть нашей книги.

Глава 1

Алгоритмы и их сложность

1.1 Примеры задач и алгоритмов

1.1.1 Теоретико-числовые задачи: «НОД», «факториал», «возвведение в степень», «дискретный логарифм»

Мы начнем рассмотрение парадигмы сложности с простой задачи о натуральных числах.

Задача 1. Даны три натуральных числа x , n , и m . Требуется вычислить $y = x^n \bmod m$.

Легко придумать следующий тривиальный алгоритм (см. алгоритм 1), который в цикле от 1 до n выполняет следующее действие: $y \leftarrow y \cdot x \bmod m$.

Очевидный анализ показывает, что сложность этого алгоритма есть величина $O(n)$. Пока, на неформальном уровне, под сложностью мы будем понимать число элементарных операций.

Алгоритм 1 Тривиальное вычисление $y = x^n \bmod m$

```
def mod_exp(x, n, m):
    print x, n, m
    y = 1
    for i in range(n):
        y = y*x % m
        print y
    return y
```

13	16	47
13		
28		
35		
32		
40		
3		
39		
37		
11		
2		
26		
9		
23		
17		
33		
6		
y = 13^16 (mod 47) = 6		

Рассмотрим двоичное разложение $n = \sum_{i=0}^k a_i 2^i$, $a_i \in \{0, 1\}$. Заметим, что

$$x^n = x^{\sum_{i=0}^k a_i 2^i} = \prod_{i=0}^k x^{a_i 2^i} = \prod_{\{i: a_i > 0\}} x^{2^i}.$$

т. е. достаточно провести $l \leq (\log_2 n + 1)$ итераций по двоичному разложению n , чтобы на каждой i -й итерации, в зависимости от i -го бита в разложении, проводить умножение результата на сомножитель x^{2^i} , который легко получить из сомножителя предыдущей итерации (ключевое соотношение):

$$x^{2^i} \leftarrow \left(x^{2^{i-1}} \right)^2.$$

Значит, можно существенно улучшить алгоритм 1, использовав эти соображения (см. алгоритм 2). Анализ же алгоритма 2 показывает, что его сложность есть $O(\log n)$, т. е. разница в сложности первоначального прямолинейного алгоритма и его модификации, основанной на быстром возведении в степень, экспоненциальна!

Алгоритм 2 Разумное вычисление $y = x^n \bmod m$

```
def mod_exp(x, n, m):
    print x, n, m
    y = 1
    X = x
    N = n
    while N > 0:
        if N % 2 == 1:
            y = y * X % m
        X = X * X % m
        N = N / 2
    print X, N, y
return y
```

13	16	47
28	8	1
32	4	1
37	2	1
6	1	1
36	0	6
$y = 13^{16} \pmod{47} = 6$		

Заметим, что процедура расчета модульной экспоненты весьма распространена в различных сетевых протоколах, например, при установке защищенного интернет-соединения (всякий раз, когда в браузере адрес начинается с `https://`, при установке соединения работают алгоритмы криптографии, и вычисляется модульная экспонента). Если взять длину показателя экспоненты 128 бит, то получается простое сравнение количества умножений в обоих алгоритмах:

Длина показателя (бит)	Умножений в алг. 1	Умножений в алг. 2
56	$2^{56} \approx 7.2 \cdot 10^{16}$	≈ 100
128	$2^{128} \approx 3.4 \cdot 10^{38}$	≈ 130

Комментарии излишни.

Упражнение 1.1.1. Более внимательно взглянув на алгоритм быстрого возведения в степень, видно, что достаточно $2(\log_2 n + 1)$ умножений. На самом деле, как видно из изложенного, для чисел вида $n = 2^k$ достаточно $k = \log_2 n + 1$ умножений. Можно ли получить такую же асимптотическую оценку $\sim \log_2 n$ для чисел вида $n = 2^k + 2^{k-1} + 2^{k-2} + \dots + 2^{k-t}$? Для произвольных чисел?

Теперь рассмотрим примитивный алгоритм точного вычисления факториала по модулю m (см. алгоритм 3).

Нетрудно видеть, что его сложность есть $O(n)$. Можно ли ее существенно улучшить, как в случае с возведением степень? Оказывается, на сегодняшний день это является известной открытой проблемой (см. следствия из «Problem 4» в [Sma00]).

Рассмотрим еще одну очень важную задачу, связанную с возведением в степень по модулю.

Задача 2. «Дискретный логарифм» (*Discrete logarithm*) Даны натуральные a, b , и p — нечетное простое число. Найти минимальный x , такой, что

$$a^x \equiv b \pmod{p}.$$

По сути, эта обратная задача к задаче возведения в степень по модулю. Прямолинейный алгоритм ее решения переборного типа, конечно, существует. Достаточно для каждого x в интервале $[1, p]$ вычислить

Алгоритм 3 Вычисление факториала $y = n! \bmod m$

```
def factorial(n, m):
    print n, m
    y = 1
    for i in range(1, n+1):
        y = y * i % m
    print y
return y
```

13	131
1	
2	
6	
24	
120	
65	
62	
103	
10	
100	
52	
100	
121	
y = 13! mod 131 = 121	

$a^x \bmod p$ и проверить требуемое равенство. Возвведение в степень делается эффективно, как мы уже видели. Проблема в том, что эту эффективную процедуру придется проделать огромное число раз (например, при p порядка 2^{100}).

Можем ли мы, как и в задаче о возведении в степень, ожидать существования существенно более эффективного алгоритма решения? Оказывается, это весьма маловероятно, вычисление дискретного логарифма является очень сложной задачей, и самые быстрые из известных алгоритмов требуют сверхполиномиального (по длине двоичной записи p) времени. На этом важном свойстве (свойстве односторонней эффективной вычислимости модульной экспоненты) основано множество протоколов современной криптографии.

Еще одним примером взаимно обратных преобразований подобного рода являются операции произведения натуральных чисел и разложение на множители. Пусть p и q — два простых числа. Тогда не

составляет особого труда вычислить их произведение $n = pq$ (применив, например, школьный алгоритм умножения «столбиком»). Однако если мы знаем лишь n и требуется найти p и q , то это уже труднорешаемая задача, для которой неизвестны эффективные алгоритмы решения. На трудности задачи факторизации (разложения на множители) основана одна из самых известных криптосистем RSA.

Рассмотрим одну из классических задач.

Задача 3. «Наибольший общий делитель»

Для заданных натуральных a и b найти максимальное q , для которого $a \vdots q$ и $b \vdots q$ (a и b кратны q).

Проанализируем алгоритм Евклида (алгоритм 4) для решения этой задачи (для определенности ниже будем считать, что $a \leq b$). Базовое соотношение, лежащее в основе алгоритма Евклида, таково:

$$\text{НОД}(a, b) = \text{НОД}(a, r), \quad b = at + r, \quad 0 \leq r < a.$$

Алгоритм 4 Алгоритм Евклида

```
def gcd(a, b):
    print a, b
    if a == 0:
        return b
    return gcd(b % a, a)
```

```
Calculating gcd(123456, 6122256):
123456 6122256
72912 123456
50544 72912
22368 50544
5808 22368
4944 5808
864 4944
624 864
240 624
144 240
96 144
48 96
0 48
gcd(123456, 6122256) = 48
```

Лемма 1. Время работы алгоритма 4 «НОД» составляет $O(\log a + \log b)$ арифметических операций над натуральными числами.

Доказательство. Имеем

$$b \geq a + (b \bmod a) \geq 2(b \bmod a) \equiv 2r.$$

Отсюда получаем, что $r \leq \frac{b}{2}$. Таким образом, $ar \leq ab/2$, т. е. произведение ab уменьшается на каждой итерации вдвое, и после $\lceil \log(ab) \rceil$ итераций станет меньше 1, т. е. равно нулю. А это означает, что $a = 0$ (т. е. НОД уже найден). \square

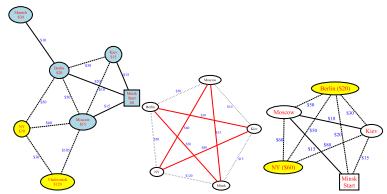
Алгоритм 5 Неэффективный алгоритм для «НОД»

```
def gcd_non_effective(a, b):
    print a, b
    if a == 0:
        return b
    return gcd( max(a, b) - min(a, b), min(a, b) )
```

Длина записи исходных данных в этой задаче есть $O(\log a + \log b)$ и по порядку совпадает с числом операций алгоритма Евклида, т. е. алгоритм является эффективным. Отметим, что алгоритм 5 «Неэффективный НОД» (модификация алгоритма 4 «НОД», где деление с остатком заменено на вычитание меньшего числа из большего) не является столь же эффективным — ведь число операций вычитания пропорционально $\frac{b}{a}$, а не сумме их логарифмов, и на входных данных, где b много больше a , алгоритм будет показывать экспоненциальную сложность.

Таким образом, мы увидели, что, несмотря на внешнюю похожесть, сложность подобных задач может существенно отличаться. Для обнаружения и понимания такой разницы и нужен анализ сложности, являющийся предметом нашего курса.

1.1.2 Задачи на графах: «Коммивояжер», «Кратчайшие пути», «Остовные деревья»



Классические задачи на графах. Различия сложности простых алгоритмов, решающих эти задачи.

Рассмотрим несколько классических задач на графах.

Задача 4. «Коммивояжер», «TSP¹». Заданы неориентированный граф из n вершин-городов, и $d_{ij} \equiv d(v_i, v_j)$ — положительные целые расстояния между городами.

Чему равна наименьшая возможная длина гамильтонова цикла (кольцевого маршрута, проходящего по одному разу через все города)? т. е. нужно найти минимально возможное значение суммы

$$\min_{p \in \begin{pmatrix} 1 & 2 & \dots & n \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}} \sum_{i=1}^{n-1} d_{p_i, p_{i+1}} + d_{p_n, p_1}, \quad (1.1)$$

где минимум берется по всем перестановкам p чисел $1, \dots, n$.

Переборный алгоритм для задачи о коммивояжере просто перебирает все возможные перестановки городов с фиксированным «стартовым» городом (см. алгоритм 6). При анализе его сложности видно, что вычисление индивидуальной суммы (1.1) не представляет особых трудностей и требует Cn операций, где C — некоторая константа. Проблема состоит в том, что этот процесс придется повторить $(n - 1)!$ раз, что дает общую сложность алгоритма $\Omega(n!)$. Некоторые значения факториала приведены в таблице 1.1.

¹В англоязычной литературе — Traveling Salesman Problem.

Алгоритм 6 Переборный алгоритм для «TSP»

```
def TSP_BruteForce(G):
    def get_path_length(path):
        path_length = 0
        for i, v1 in enumerate(path):
            v2 = path[ (i+1) % len(path) ]
            if not G.has_edge(v1, v2):
                return INFINITY
            path_length += G[v1][v2][“weight”]
    return path_length

min_path = min_path_length = None
# перебор всех перестановок с фиксированным первым узлом
for path in all_vertices_permutations(G):
    path_length = get_path_length(path)
    if not min_path or min_path_length > path_length:
        min_path, min_path_length = path, path_length
return min_path, min_path_length
```

Иллюстрация работы показана на рис. 1.1.

Таблица 1.1: Значения функции $n!$

n	5	8	10	13	15	30
$n!$	120	40320	$3.6 \cdot 10^6$	$6.2 \cdot 10^9$	$1.3 \cdot 10^{12}$	$2.7 \cdot 10^{32}$

Видно, что при $n = 5$ расчет всех вариантов согласно переборному алгоритму может быть произведен вручную. При $n = 8$ для его проведения в разумный отрезок времени нужно привлечь программируемый калькулятор, а при $n = 10$ — уже более быстродействующую вычислительную технику. Когда число городов дойдет до 13, потребуется суперкомпьютер, а случай $n = 15$ выходит за пределы возможностей любой современной вычислительной техники.

Число возможных вариантов при $n = 30$ превышает количество атомов на Земле².

Теперь рассмотрим задачу 5, на первый взгляд — очень похожую на задачу 4 «TSP».

Задача 5. «Кратчайший путь в графе»³.

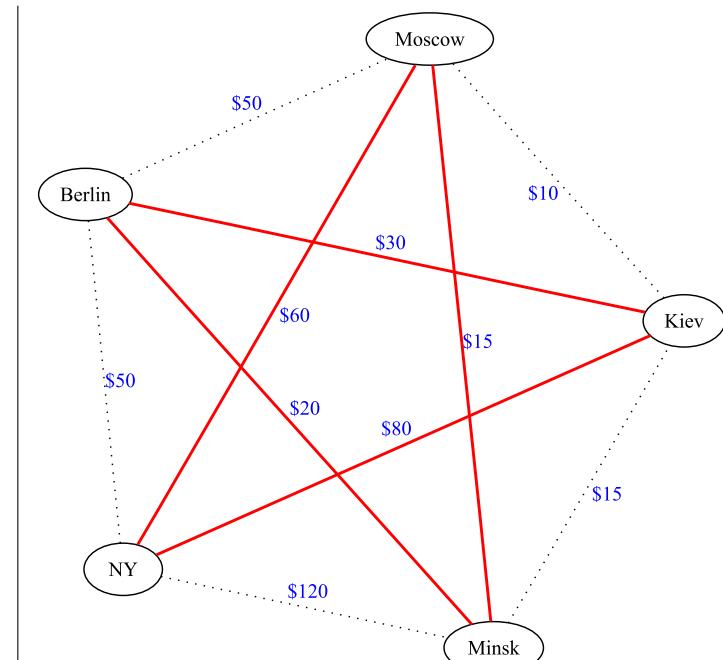
Заданы n вершин графа (узлов сети) v_1, v_2, \dots, v_n и положительные целые длины дуг $d_{ij} \equiv d(v_i, v_j)$ между ними.

²Конечно, разработаны различные методы сокращения перебора, кроме того, переборные задачи допускают эффективное распараллеливание для вычисления на многопроцессорных машинах или сети компьютеров. В частности, в 2004 году для задачи 4 «TSP» на 24978 городах Швеции было найдено оптимальное решение на многопроцессорном кластере (96 dual processor Intel Xeon 2.8 GHz). Если измерять вычислительное время относительно одного процессора «single Intel Xeon 2.8 GHz processor», то было потрачено 85 лет. См. отчет <http://www.tsp.gatech.edu/sweden/index.html>. Но это не отменяет высказанных соображений о непрактичности использования экспоненциальных алгоритмов перебора, кроме случаев входных данных ограниченного размера.

³В англоязычной литературе — Shortest Path Problem.

Рис. 1.1: Работа алгоритма 6 «TSP-перебор»

```
[‘NY’, ‘Moscow’, ‘Minsk’, ‘Berlin’, ‘Kiev’] 205  
[‘NY’, ‘Moscow’, ‘Minsk’, ‘Kiev’, ‘Berlin’] 170  
[‘NY’, ‘Moscow’, ‘Berlin’, ‘Minsk’, ‘Kiev’] 225  
[‘NY’, ‘Moscow’, ‘Berlin’, ‘Kiev’, ‘Minsk’] 275  
[‘NY’, ‘Moscow’, ‘Kiev’, ‘Minsk’, ‘Berlin’] 155  
[‘NY’, ‘Moscow’, ‘Kiev’, ‘Berlin’, ‘Minsk’] 240  
[‘NY’, ‘Minsk’, ‘Moscow’, ‘Berlin’, ‘Kiev’] 295  
[‘NY’, ‘Minsk’, ‘Moscow’, ‘Kiev’, ‘Berlin’] 225  
[‘NY’, ‘Minsk’, ‘Berlin’, ‘Moscow’, ‘Kiev’] 280  
[‘NY’, ‘Minsk’, ‘Berlin’, ‘Kiev’, ‘Moscow’] 240  
[‘NY’, ‘Minsk’, ‘Kiev’, ‘Moscow’, ‘Berlin’] 245  
[‘NY’, ‘Minsk’, ‘Kiev’, ‘Berlin’, ‘Moscow’] 275  
[‘NY’, ‘Berlin’, ‘Moscow’, ‘Minsk’, ‘Kiev’] 210  
[‘NY’, ‘Berlin’, ‘Moscow’, ‘Kiev’, ‘Minsk’] 245  
[‘NY’, ‘Berlin’, ‘Minsk’, ‘Moscow’, ‘Kiev’] 175  
[‘NY’, ‘Berlin’, ‘Minsk’, ‘Kiev’, ‘Moscow’] 155  
[‘NY’, ‘Berlin’, ‘Kiev’, ‘Moscow’, ‘Minsk’] 225  
[‘NY’, ‘Berlin’, ‘Kiev’, ‘Minsk’, ‘Moscow’] 170  
[‘NY’, ‘Kiev’, ‘Moscow’, ‘Minsk’, ‘Berlin’] 175  
[‘NY’, ‘Kiev’, ‘Moscow’, ‘Berlin’, ‘Minsk’] 280  
[‘NY’, ‘Kiev’, ‘Minsk’, ‘Moscow’, ‘Berlin’] 210  
[‘NY’, ‘Kiev’, ‘Minsk’, ‘Berlin’, ‘Moscow’] 225  
[‘NY’, ‘Kiev’, ‘Berlin’, ‘Moscow’, ‘Minsk’] 295  
[‘NY’, ‘Kiev’, ‘Berlin’, ‘Minsk’, ‘Moscow’] 205  
Оптимальный путь: [‘NY’, ‘Moscow’, ‘Kiev’, ‘Minsk’, ‘Berlin’] => 155
```

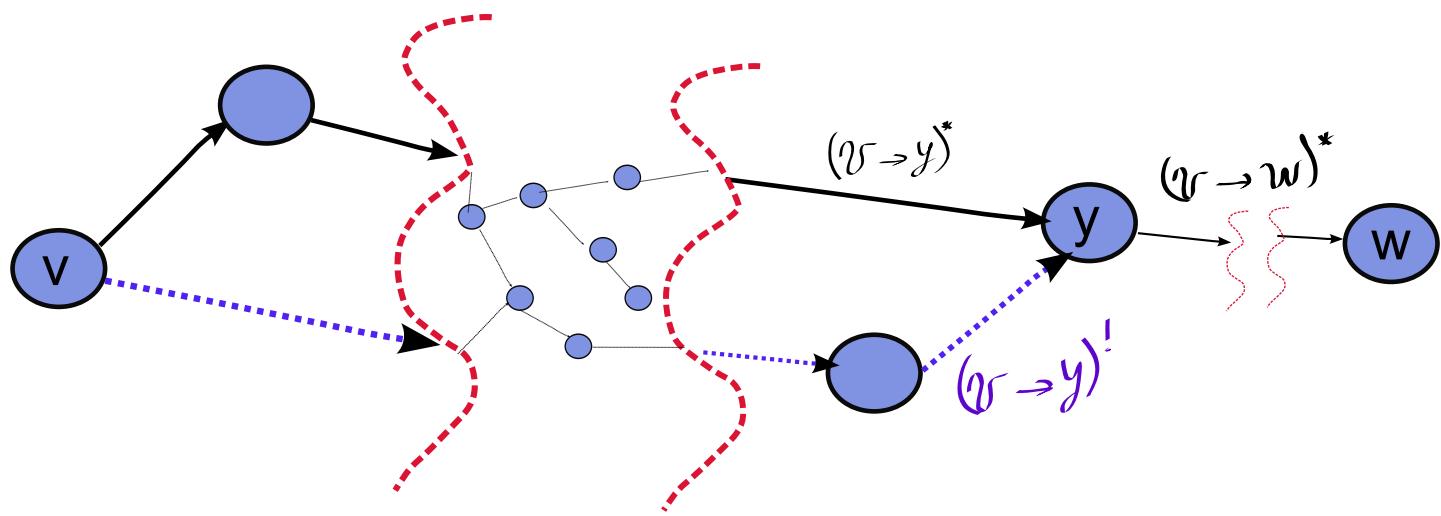


Нужно для всех $k \in (2 \dots n)$ найти минимальную длину пути из v_1 в v_k .

Разумеется, и эту задачу можно решать переборным алгоритмом, аналогичным алгоритму 6 «TSP-перебор», но интересно, можно ли разработать точный эффективный алгоритм, исключающий (или, по меньшей мере, минимизирующий) непосредственный перебор вариантов.

Оказывается, в данном случае можно. Здесь важным фактом является то, что если у нас есть кратчайший путь от v до w , проходящий через вершину y , назовем его $(v \rightarrow w)^*$, то его первая часть от v до y , $(v \rightarrow y)^*$ тоже будет кратчайшим путем.

Действительно, если бы это было не так, т. е. существовал бы путь $(v \rightarrow y)!$ длины меньшей, чем $(v \rightarrow y)^*$, то можно было бы улучшить оптимальный путь $(v \rightarrow w)^*$, заменив в нем $(v \rightarrow y)^*$ на $(v \rightarrow y)!$.



Задачи с подобными свойствами, когда оптимальное решение можно легко получить из оптимальных решений подзадач, обычно хорошо решаются так называемыми «жадными алгоритмами», примером которых может служить алгоритм⁴ 7 для задачи 5 «SPP». В алгоритме 7 мы итерационно поддерживаем два множества вершин:

- $Visited$ — множество вершин, до которых мы уже нашли кратчайший путь, ассоциированных со стоимостями кратчайших путей от стартовой вершины до них.
- $ToVisit$ — множество вершин, которые достижимы одной дугой из множества вершин $Visited$, ассоциированных с верхними оценками стоимости пути до них.

На каждой итерации мы выбираем из достижимых вершин вершину v , самую ближнюю к стартовой вершине s , и переносим ее из множества $ToVisit$ в множество $Visited$, увеличиваем множество «кандидатов» $ToVisit$ ее соседями и пересчитываем верхнюю оценку удаленности вершин из $ToVisit$ до вершины s .

В иллюстрации выполнения алгоритма 7 мы показываем изменение на каждой итерации хэш-таблиц $Visited$ и $ToVisit$, а в конце изображен входной граф, где сплошными линиями нарисованы найденные кратчайшие пути, а пунктиром — имеющиеся ребра с ассоциированными длинами.

Лемма 2. Алгоритм 7 «Дейкстры» корректен, т. е. все найденные им пути оптимальны.

Доказательство. Обозначим через $\delta(v, w)$ длину оптимального пути из v в w . Допустим, алгоритм некорректен — это означает, что алгоритм построит неоптимальный путь до какой-то вершины. Пусть вершина u будет первой вершиной, до которой «неоптимально» доберется наш алгоритм, построив путь $(s \rightarrow u)^*$,

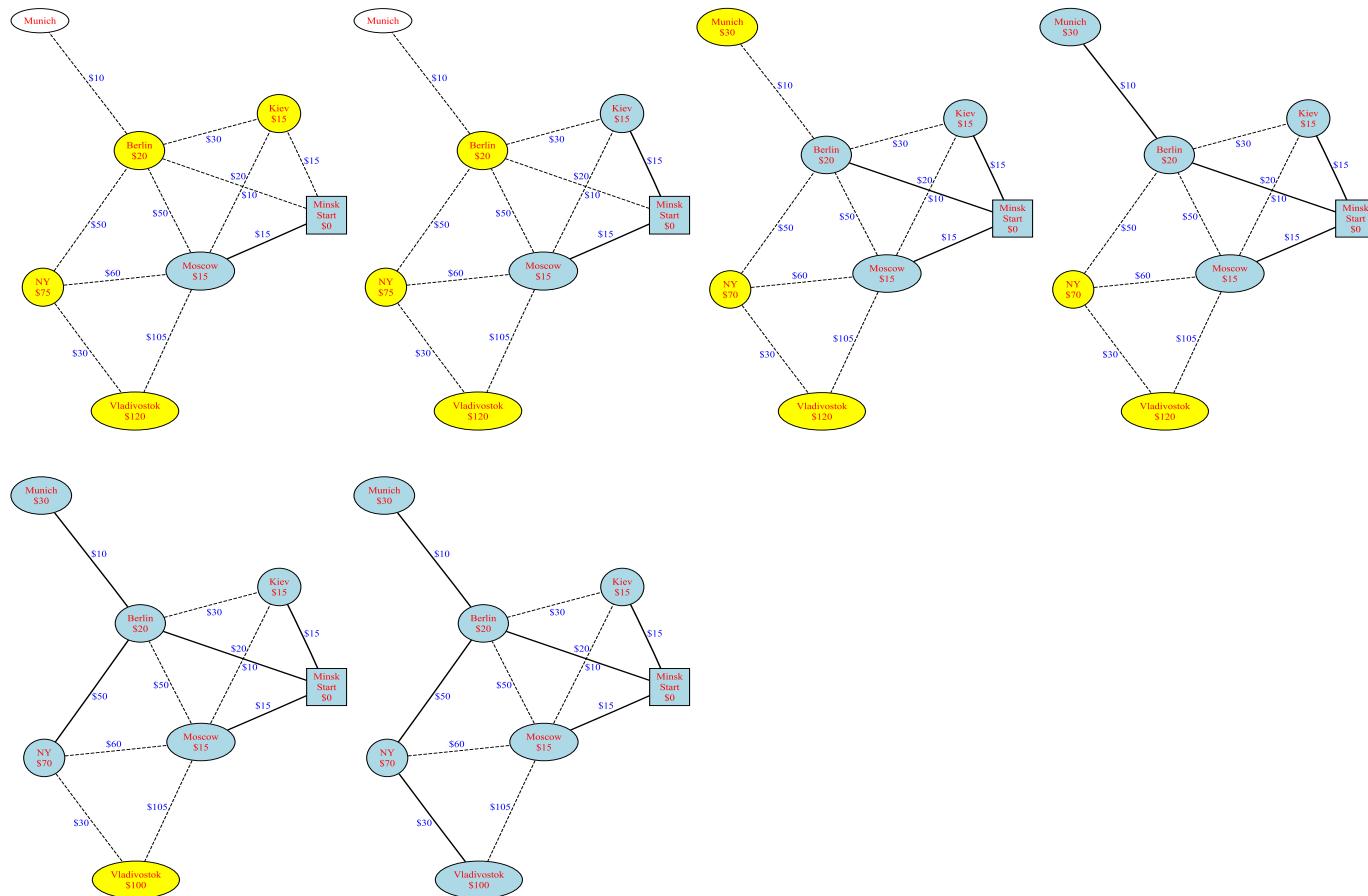
⁴Различные варианты этого алгоритма используются при маршрутизации интернет-трафика. Например, см. стандарты OSPF, Open Shortest Path First Routing Protocol, RFC 2740.

Алгоритм 7 Алгоритм Дейкстры

```
def dijkstra(G, startNode):
    Visited = {}      # хэш посещенных вершин (узел -> стоимость)
    ToVisit = {startNode: 0} # соседи посещенных (уз. -> стоим.)
    Paths = {startNode:[startNode]} # узел -> кратчайший путь
    while ToVisit: # пока есть куда стремиться
        v = argmin(ToVisit)          # выбираем ближайшую
        Visited[v] = ToVisit[v]      # фиксируем откуда пришли в v
        del ToVisit[v]              # кратчайший путь к v найден
        for w in G.neighbors(v): # для всех соседей вершины v
            if w not in Visited: # еще не нашли кратчайший путь
                # обновляем кратчайшие пути
                vwLength = Visited[v] + G[v][w]["weight"]
                if w not in ToVisit or vwLength < ToVisit[w]:
                    ToVisit[w] = vwLength # обновляем цену
                    Paths[w] = Paths[v] + [w] # посещения соседа
    return Visited, Paths
```

Иллюстрация работы показана на рис. 1.2.

Рис. 1.2: Работа алгоритма 7 «Дейкстры»



т. е. $Visited[u] > \delta(s, u)$, но все его подпути $(s \rightarrow w)$ еще будут оптимальными и $Visited[w] = \delta(s, w)$. Далее будем рассматривать действия алгоритма в той итерации, в которой он выбирает u .

Рассмотрим оптимальный путь $(s \rightarrow u)!$ (он, как мы выше предположили, не совпадает с $(s \rightarrow u)^*$). Он должен содержать в себе предпоследнюю вершину $y \notin Visited$. В противном случае, если бы предпоследняя вершина z из $(s \rightarrow u)!$ в момент выбора u принадлежала $Visited$, то был бы построен оптимальный кратчайший путь, т.к. $Visited[z] = \delta(s, z)$ и

$$ToVisit[u] \leq Visited[z] + \delta(z, u) = \delta(s, u),$$

что приводит нас к противоречию с высказанным допущением о неоптимальности выбранного алгоритмом пути.

Значит, в момент выбора алгоритмом вершины u должны существовать вершины x, y , идущие подряд в оптимальном пути $(s \rightarrow u)!$, причем $y \notin Visited$, а $x \in Visited$.

Следовательно, $y \in ToVisit$ и выполняется неравенство

$$ToVisit[y] \leq Visited[x] + \delta(x, y) = \delta(s, x) + \delta(x, y) = \delta(s, y).$$

Далее, из неравенств

- $\delta(s, y) < \delta(s, u)$ (в силу положительности весов, т. е. длин дуг),
- $Visited[u] \leq ToVisit[y] = \delta(s, y)$ (т.к. алгоритм выбрал на этой итерации вершину u , а не y)

следует $Visited[u] < \delta(s, u)$. Противоречие. □

Лемма 3. Трудоемкость алгоритма 7 «Дейкстры» составляет $O(n^2)$ операций, где n — число вершин.

Доказательство. Внешний цикл по вершинам дает множитель n . Внутренний цикл содержит выбор минимального элемента — $O(n)$ плюс пересчет оценок длин для соседей выбранного узла — $O(n)$. Итого — $O(n^2)$. \square

Вообще, используя специальные структуры данных (очереди приоритетов на основе фибоначчиевой кучи), можно добиться оценки сложности $O(n \log n + m)$, где n — число вершин, а m — число ребер графа.

Упражнение 1.1.2. Приведите пример графа с отрицательными весами, но без циклов отрицательной длины, для которого алгоритм Дейкстры даст неправильный ответ.

Упражнение 1.1.3. Неориентированный граф $G = (V, E)$ представляет телекоммуникационную сеть, причем с каждым ребром $e \in E$ ассоциирована «надежность», $0 \leq p_e \leq 1$, т. е. вероятность успешного прохождения сигнала (в обе стороны) по ребру e . Считая события успешного или неуспешного прохождения сигнала по различным ребрам независимыми, придумайте алгоритм нахождения наиболее надежных маршрутов между данным узлом и всеми остальными.

Для поиска кратчайших путей между любыми двумя парами вершин можно использовать $|V|$ запусков алгоритма 7 «Дейкстры», что даст эффективное решение этой задачи алгоритмом со сложностью $O(n^3)$. Также алгоритм 7 «Дейкстры» будет работать на ориентированных графах с неотрицательными весами дуг.

Однако такой подход не будет работать на задаче о кратчайших путях на ориентированном графе, дуги которого могут иметь отрицательные веса (с запретом на наличие в графе циклов отрицательной длины, т.к. очевидно, что при их наличии оптимального решения быть не может).

Задача 6. «Кратчайшие пути с отрицательными расстояниями».

Задан ориентированный граф $G = (V, E)$ и весовая функция на дугах $w_e : e \rightarrow \mathbb{Z}$, отображающая ребра в целые числа, такая, что в графе нет цикла отрицательной длины. Найти минимальные длины путей между всеми парами вершин.

Алгоритм 8 Алгоритм Флойда-Уоршолла

```

for k in range(N): # $N$---- размер матрицы
    DD = array(D) #Сохраняем $D_{\{k-1\}}$ в $DD$
    for v1 in range(N):
        for v2 in range(N):
            D[v1, v2] = min(DD[v1, v2], DD[v1, k] + DD[k, v2])

```

Иллюстрация работы показана на рис. 1.3.

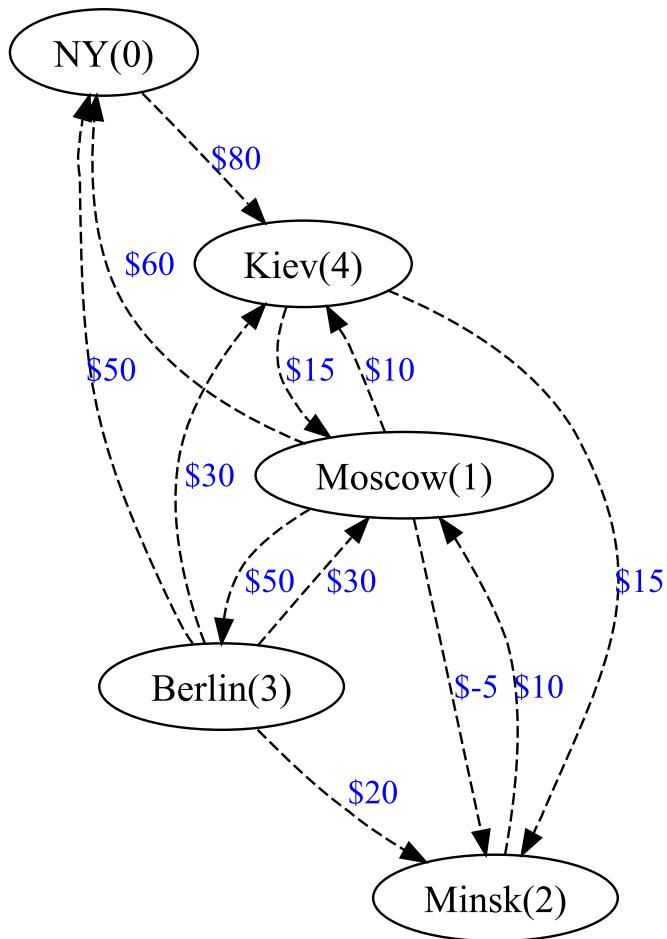
Для решения этой задачи применяется алгоритм 8, использующий методы динамического программирования. В этом алгоритме последовательно выполняются $n = |V|$ итераций, улучшая матрицу D_{ij} минимальных стоимостей пути из вершины i в вершину j , с возможным использованием (для k -й итерации) промежуточных вершин из множества $1, \dots, k$. Вычислять эту матрицу очень легко, изначально она определяется весовой функцией дуг, $D_{ij}^0 = w_{ij}$, для тех i и j , для которых есть дуга (i, j) , и $+\infty$ для остальных. Обновление этой матрицы на k -й итерации происходит по очевидной формуле:

$$D_{ij}^k = \min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1}),$$

где D^{k-1} — значение этой матрицы на предыдущей итерации. Таким образом, очевидна и корректность (то, что алгоритм находит требуемое решение) алгоритма 8, и оценка его сложности $O(n^3)$. В иллюстрации работы алгоритма 8 приведен вывод промежуточных структур и изображен входной граф.

Таким образом, использование «жадных алгоритмов» и «динамического программирования», позволили построить **эффективные алгоритмы** для задач, очевидным решением которых был полный пе-

Рис. 1.3: Иллюстрация работы алгоритма 8 «Флойда – Уоршолла»



k=0 (NY)	NY	Moscow	Minsk	Berlin	Kiev
NY	0	∞	∞	∞	80
Moscow	60	0	-5	50	10
Minsk	∞	10	0	∞	∞
Berlin	50	30	20	0	30
Kiev	∞	15	15	∞	0

k=1 (Moscow)	NY	Moscow	Minsk	Berlin	Kiev
NY	0	∞	∞	∞	80
Moscow	60	0	-5	50	10
Minsk	$\infty \Rightarrow 70$	10	0	$\infty \Rightarrow 60$	$\infty \Rightarrow 20$
Berlin	50	30	20	0	30
Kiev	$\infty \Rightarrow 75$	15	$15 \Rightarrow 10$	$\infty \Rightarrow 65$	0

k=2 (Minsk)	NY	Moscow	Minsk	Berlin	Kiev
NY	0	∞	∞	∞	80
Moscow	60	0	-5	50	10
Minsk	70	10	0	60	20
Berlin	50	30	20	0	30
Kiev	75	15	10	65	0

k=3 (Berlin)	NY	Moscow	Minsk	Berlin	Kiev
NY	0	∞	∞	∞	80
Moscow	60	0	-5	50	10
Minsk	70	10	0	60	20
Berlin	50	30	20	0	30
Kiev	75	15	10	65	0

k=4 (Kiev)	NY	Moscow	Minsk	Berlin	Kiev
NY	0	$\infty \Rightarrow 95$	$\infty \Rightarrow 90$	$\infty \Rightarrow 145$	80
Moscow	60	0	-5	50	10
Minsk	70	10	0	60	20
Berlin	50	30	20	0	30
Kiev	75	15	10	65	0

ребор, требующий экспоненциального времени. Об этих алгоритмах и понятии эффективности будет более подробно рассказано в следующих разделах.

Задача 4 «TSP» и задачи о кратчайших путях чрезвычайно похожи по своей структуре, и мы постарались подчеркнуть это сходство в их формулировках.

Сравнительно успешно устранив перебор для задач о кратчайших путях, естественно задаться аналогичным вопросом для задачи 4 «TSP». Довольно быстро выясняется, впрочем, что ни один из «естественных» методов сокращения перебора к последней задаче неприменим. Таким образом, встает законный вопрос: а можно ли вообще решить задачу 4 «TSP» с помощью точного алгоритма, существенно более эффективного, нежели переборный?

Одним из главных достижений теории сложности вычислений является теория \mathcal{NP} -полноты, позволяющая в 99% случаев дать вполне удовлетворительный ответ на этот вопрос.

Эта теория будет рассмотрена далее в разделе 6.2, пока же термин \mathcal{NP} -полнная задача (или \mathcal{NP} -трудная задача) можно понимать неформально в смысле труднорешаемая переборная задача, для которой существование алгоритма, намного более эффективного, нежели простой перебор вариантов, крайне маловероятно.

В частности, в одном из первых исследований было показано, что задача 4 «TSP» \mathcal{NP} -трудна, и, тем самым, на возможность построения для нее точного эффективного алгоритма рассчитывать не приходится.

Поэтому следующий вопрос, на который пытается ответить теория сложности вычислений: какие рекомендации можно дать практическому разработчику алгоритмов в такой ситуации, т. е. в тех случаях, когда результаты диагностики интересующей его задачи на существование для нее точных эффективных алгоритмов столь же неутешительны, как и в случае задачи 4 «TSP»?

Одна из таких рекомендаций состоит в следующем: попытаться проанализировать постановку задачи и понять, нельзя ли видоизменить ее формулировку так, чтобы, с одной стороны, новая формулировка

все еще была бы приемлема с точки зрения практических приложений, а с другой стороны, чтобы в этой формулировке задача уже допускала эффективный алгоритм. Кстати, в качестве побочного продукта в ряде случаев такой сложностной анализ позволяет лучше понять природу задачи уже безотносительно к ее вычислительной сложности.

Например, пусть некий начинающий проектировщик сетей задумал спроектировать оптимальную компьютерную сеть, соединяющую n корпусов общежитий. Он только что изучил кольцевые топологии сети и вознамерился проложить кольцевой сетевой маршрут через все корпуса общежитий. Стоимость прокладки кабеля между любыми двумя корпусами известна (если между какими-то корпусами кабель проложить нельзя, например, из-за постоянных работ по ремонту теплотрасс, то стоимость полагается равной $+\infty$). Формулировка этой задачи в чистом виде совпадает с задачей 4 «TSP». Как мы уже видели раньше, если число корпусов больше 10, то проектировщику потребуется доступ к дорогой вычислительной технике, если еще увеличить размер задачи, то можно не получить оптимальное решение за время человеческой жизни.

Что же делать начинающему проектировщику сети?

Почитав дальше книгу по проектированию сетей, он решает построить *минимальную связную сеть*, используя минимальные связные (из $n - 1$ дуги) подграфы исходного потенциального графа связности общежитий, так называемые *остовные деревья*⁵.

Задача 7. «Минимальное остовное дерево» (*Minimum Spanning Tree*)

Задан связный неориентированный граф $G = (V, E)$, где $V = \{v_1, \dots, v_n\}$ — множество вершин, $|V| = n$, E — множество ребер между ними, и весовая функция $w : E \rightarrow Z^+$ ⁶

⁵ В англоязычной литературе — spanning trees.

⁶ Можно вводить положительные целые веса на ребрах, как $w_{ij} \equiv w(v_i, v_j)$.

Требуется найти наименьший возможный вес оставного дерева, т. е.

$$\min \sum_{(i,j) \in T} w(v_i, v_j), \quad (1.2)$$

где минимум берется по всем оставным деревьям на n вершинах (по всем множествам T из $(n - 1)$ дуг, связывающим все n вершин в единую сеть).

На первый взгляд, переход от задачи 4 «TSP» к задаче 7 «Minimum Spanning Tree» только увеличивает трудности, стоящие перед нашим проектировщиком: перебор по множеству всех замкнутых путей заменяется перебором по еще более «необозримому» множеству произвольных оставных деревьев. Тем не менее, эта интуиция в данном случае в корне ошибочна, поскольку так же, как и в случае с кратчайшими путями, существуют эффективные алгоритмы для задачи 7 «Minimum Spanning Tree».

Опишем один из них, так называемый *алгоритм Прима*⁷. В этом алгоритме минимальный остов строится постепенно: сначала выбирается произвольная вершина, которая включается в остов, затем на каждой итерации к текущему остову добавляется наиболее дешевое ребро (u, v) , соединяющее какую-либо вершину из остова u с какой-либо вершиной v не из остова.

Алгоритм Прима и иллюстрация его работы представлены как алгоритм 9. Видно, что этот алгоритм похож на алгоритм 7 «Дейкстры».

Упражнение 1.1.4. Для алгоритма 9 «MST Прима» докажите корректность, т. е., что алгоритм действительно находит оптимальное решение.

Видно, что сложность алгоритма 9 «MST Прима», гарантирующего оптимальное решение, равна $O(n^3)$, а не экспоненциальна, как в алгоритме для задачи 4 «TSP».

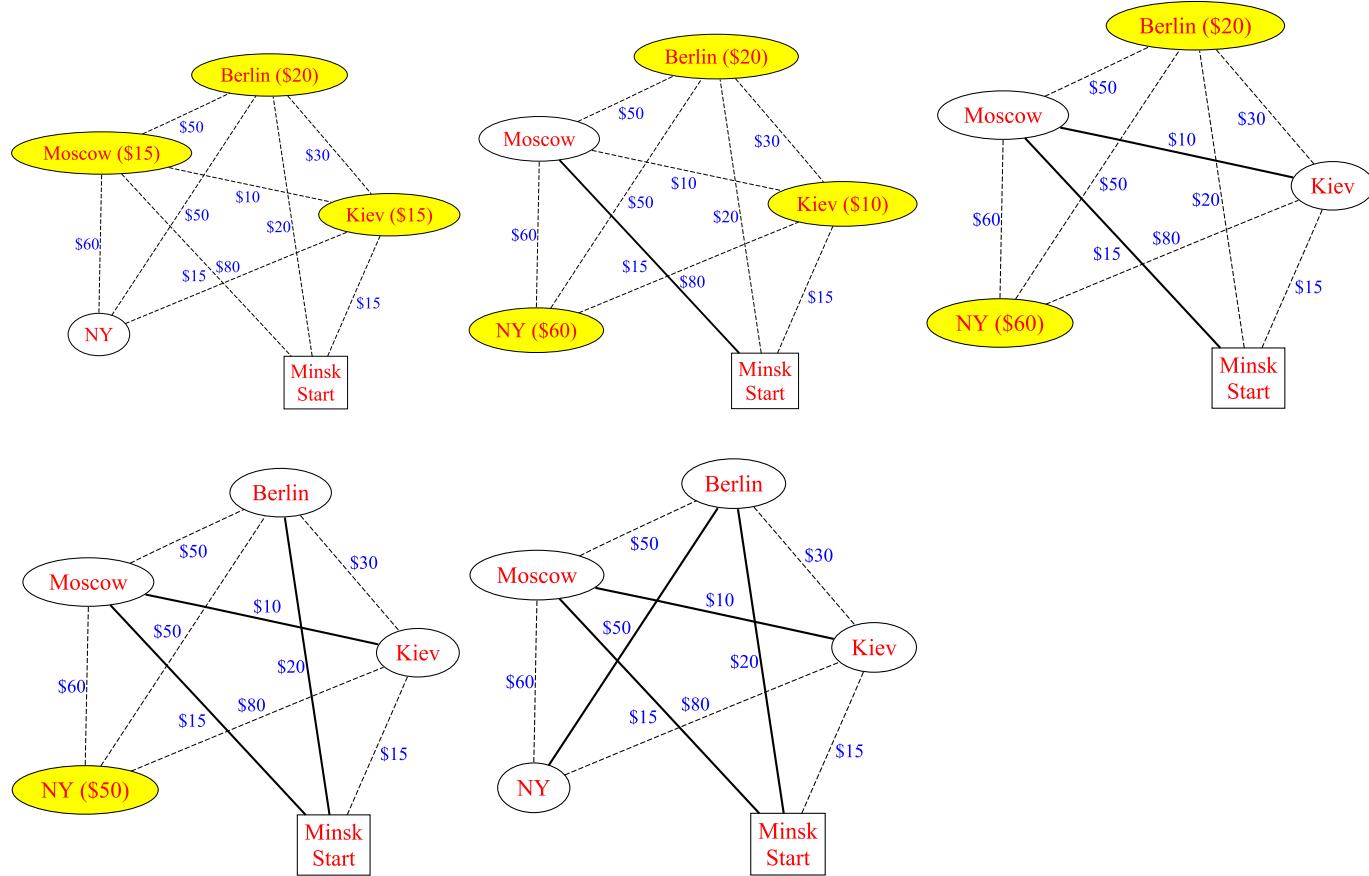
⁷Впервые опубликован в [Pri57]. Для этой задачи широко известен также алгоритм Краскала [КЛР99].

Алгоритм 9 Алгоритм Прима

```
def MST_Prim(G, s):
    MST = {} #хэш (узел: предшественник)
    ToVisit = {s: 0} # граничащие с MST, (узел: стоимость)
    Ancestor = {s: s} #хэш: вершины из которых включают другие
    while ToVisit: # пока есть непосещенные вершины
        v = argmin(ToVisit) # ближайшая достижимая вершина
        MST[v] = Ancestor[v] # запоминаем, откуда пришли
        del ToVisit[v]
        del Ancestor[v] # больше не посещать
        for w in G.neighbors(v): # для всех соседей вершины v
            if w not in MST: # которые еще не в MST
                # обновляем стоимость включения в MST
                if (w not in ToVisit
                    or G[v][w]["weight"] < ToVisit[w]):
                    ToVisit[w] = G[v][w]["weight"]
                    Ancestor[w] = v
    return MST
```

Иллюстрация работы показана на рис. 1.4.

Рис. 1.4: Иллюстрация работы алгоритма 9 «MST Прима»



Упражнение 1.1.5. Докажите, что жадный алгоритм в задаче 4 «TSP», т. е. алгоритм стартующий с некоторой вершины и пытающийся добавлять наиболее дешевые ребра к еще непосещенным вершинам, не гарантирует нахождение оптимального решения.

Этот поучительный пример еще раз наглядно демонстрирует, что при устраниении перебора внутренняя структура задачи имеет гораздо большее значение, нежели размер полного перебора вариантов, а внешность задачи во многих случаях бывает крайне обманчивой, и никакие правдоподобные рассуждения «по аналогии» не могут заменить настоящего сложностного анализа. Последний в ряде случаев приводит к довольно неожиданным выводам, зачастую противоречащим обыкновенной интуиции.

Перечислим основные выводы из вышеизложенного:

1. Для подавляющего большинства задач дискретной оптимизации существуют тривиальные алгоритмы решения, основанные на переборе всех вариантов⁸. Переборные алгоритмы становятся нереализуемыми с практической точки зрения уже при сравнительно небольшом размере входных данных (см. таблицу 1.1).
2. Теория сложности вычислений занимается изучением возможности построения и анализом эффективных алгоритмов — в данном случае алгоритмов, в которых перебор вариантов устранен или, по крайней мере, сокращен до приемлемого уровня.
3. Основное внимание концентрируется на сравнительно небольшом числе «модельных», классических алгоритмических задач, каждая из которых описывает огромное число самых разнообразных приложений.

⁸Пример задач, которых невозможно решить даже конечным перебором — задачи выполнимости целочисленных нелинейных неравенств.

4. Наиболее предпочтительным решением является построение точного эффективного алгоритма для рассматриваемой задачи.
5. Имеется обширный класс \mathcal{NP} -полных задач, для которых существование точного эффективного алгоритма представляется крайне маловероятным. Классификация переборных задач на \mathcal{NP} -полные и те, которые поддаются решению с помощью точного эффективного алгоритма, оказывается весьма успешной — под нее подпадает подавляющее большинство переборных задач.
6. В тех случаях, когда интересующая разработчика практических алгоритмов задача оказывается \mathcal{NP} -полней, имеет смысл попробовать построить эффективный алгоритм для какой-либо ее модификации (т. е. близкой модельной задачи) либо частного случая, приемлемых с практической точки зрения.
7. В тех случаях, когда такую модификацию найти не удается, имеет смысл попробовать построить для решения задачи *приближенный* эффективный алгоритм, который гарантирует нахождение решения, отличающегося от оптимального не более чем в заданное число раз.

1.1.3 Приближенные алгоритмы: «Составление расписаний»

Очень часто алгоритмы, используемые на практике, не находят точное решение, а довольствуются нахождением приближенного (в некотором смысле) решения. Такие алгоритмы называются *приближенными*. Особенно часто приближенные алгоритмы применяют для решения \mathcal{NP} -трудных задач, причем актуальной задачей является анализ точности получаемого решения.

Рассмотрим для иллюстрации одну из простейших задач составления расписаний.

Задача 8. «Составление расписаний»

Имеется m одинаковых машин и n независимых работ с длительностями исполнения t_1, \dots, t_n . Распределить эти работы по машинам так, чтобы минимизировать максимальную загрузку (загрузка машины равна сумме длительностей работ, приписанных данной машине).

Несмотря на простоту постановки, эта задача трудна с вычислительной точки зрения (\mathcal{NP} -трудна). Даже для случая $m = 2$ она остается \mathcal{NP} -трудной, поскольку к ней сводится \mathcal{NP} -трудная задача о камнях: для заданного множества из n камней с весами t_1, \dots, t_n выяснить, можно ли разбить это множество на два так, чтобы суммы весов в них были равны. Это условие можно записать в виде булева уравнения:

$$\sum_{i=1}^n x_i t_i = A/2,$$

где $A = \sum_{i=1}^n t_i$, $x_i \in \{0, 1\}$ и необходимо, чтобы A было четным.

Определение 1.1.1. Эвристикой, в теории алгоритмов обычно называют некоторый интуитивно-понятный алгоритм (или принцип построения алгоритмов). Эвристика может не гарантировать какую-либо точность решения, и не иметь никаких оценок времени работы, но часто применяется из-за хороших практических результатов.

Традиционный подход к задачам такого рода состоит в использовании простых эвристик (см. определение 1.1.1), одну из которых мы сейчас проанализируем.

Эвристика: Берется произвольная работа и помещается на машину, имеющую наименьшую загрузку (загрузка равна сумме длин работ на данной машине).

Эта эвристика обладает следующим очевидным свойством.

Лемма 4. В любой момент работы этой эвристики разница в загрузке между наиболее и наименее загруженными машинами не превосходит $t_{max} = \max_i t_i$.

Теперь мы можем доказать утверждение о качестве работы нашей эвристики.

Лемма 5. *Построенное расписание отличается от оптимального (по критерию минимизации максимальной загрузки) не более чем в два раза.*

Доказательство. Пусть T^* — длина оптимального расписания (сумма длин работ на наиболее загруженной машине), T^A — длина расписания, которое построено нашей эвристикой, T_{min}^A — сумма длительностей работ на наименее загруженной машине (в расписании, построенном нашей эвристикой). Имеют место очевидные неравенства:

$$\begin{aligned} T^* &\geq T_{min}^A, \\ T^* &\geq t_{max}. \end{aligned}$$

А теперь мы легко можем оценить качество получаемого расписания:

$$\frac{T^A}{T^*} \leq \frac{T_{min}^A + t_{max}}{T^*} \leq \frac{T^* + t_{max}}{T^*} \leq 1 + \frac{t_{max}}{T^*} = 2.$$

□

Супремум отношения $\frac{T^A}{T^*}$ по всем входным данным фиксированного размера, показывающий, в какое максимально возможное число раз алгоритм A может ошибиться при поиске решения, иногда называют *мультипликативной ошибкой* алгоритма A . Таким образом, мы доказали, что мультипликативная ошибка рассмотренного алгоритма не превосходит 2.

Более того, если предположить, что $\frac{t_{max}}{T^*} \rightarrow 0$, т. е. если мы интересуемся *асимптотической ошибкой*, то из доказанного очевидно вытекает, что $\frac{T^A}{T^*} \leq 1 + \varepsilon$, где $\varepsilon \rightarrow 0$. Таким образом, асимптотическая мультипликативная ошибка нашего алгоритма стремится к единице, что является весьма желательным, но далеко не всегда достижимым эффектом.

Следует подчеркнуть, что эвристики очень часто используются для решения задач на практике. В рассмотренном примере эвристика обладала одним дополнительным свойством: она применима и в случае, когда работы поступают одна за другой, поскольку решение о назначении машины для данной работы принимается только на основании информации о состоянии машин и не изменяется с приходом следующих работ. Такие алгоритмы называются *онлайновыми*.

Вдохновленные столь успешным подходом к решению рассмотренной выше \mathcal{NP} -трудной задачи о составлении расписания, мы можем попытаться построить эффективные приближенные алгоритмы для других интересующих нас \mathcal{NP} -трудных задач. Сразу скажем, что хотя это направление исследований является магистральным, ожидать здесь только радужных положительных результатов не приходится. Как выяснилось, многие \mathcal{NP} -трудные задачи остаются трудными и с точки зрения нахождения нетривиальных приближенных решений (см. раздел 6.3.6).

Простая же иллюстрация может быть сделана сейчас на примере уже рассмотренной ранее задачи 4 «TSP». Действительно, предположим, что существует эффективный алгоритм A , который для любого входа задачи строит гамильтонов цикл веса не более чем в K раз превосходящего оптимум, причем K является функцией только от размера входа, т. е. $K = f(n)$, где n — число вершин графа. Подадим на вход алгоритма A набор весов специального вида. Пусть дан произвольный граф $G = (V, E)$, $|V| = n$. Тогда вес ребра e в полном графе зададим формулой: $w(e) = 1$, если $e \in E$, и $w(e) = Kn$, если $e \notin E$.

Посмотрим, что должен выдать алгоритм A для таких входов. Нетрудно видеть, что если в графе $G = (V, E)$ есть гамильтонов цикл, то алгоритм должен выдать число, не превосходящее n . Если же гамильтонова цикла в G нет, то алгоритм выдает число, не меньшее $nK + n - 1 > n$. Таким образом, применяя алгоритм A , мы можем решить задачу о существовании гамильтонова цикла в графе G , которая, как известно, является \mathcal{NP} -полной. Значит, такого алгоритма A не должно существовать в предположении, что $\mathcal{P} \neq \mathcal{NP}$.

Еще одна простая, но общая идея, которая повсеместно применяется при решении \mathcal{NP} -трудных за-

дач — это стремление учитывать все дополнительные ограничения на исходные данные, которые упрощают задачу, т. е. рассматривать разрешимые частные случаи исходных проблем. Так, в нашем примере веса ребер были произвольными неотрицательными числами. Если же потребовать, чтобы веса удовлетворяли дополнительно неравенству треугольника (что естественно для длин путей), то получим так называемую *метрическую задачу коммивояжера*. Она все еще остается \mathcal{NP} -трудной, но для нее уже удается построить эффективные алгоритмы, гарантирующие нахождение решения, отличающегося от оптимума не более чем в константу раз (с константой $3/2$ — см. раздел 2.1.4). Кстати, важную роль в разработке приближенных алгоритмов для метрической задачи коммивояжера играет использование сходной, но эффективно решаемой задачи о минимальном остовном дереве.

Несколько забегая вперед, можно сказать, что в классе \mathcal{APX} задач, разрешимых полиномиальными алгоритмами с константной мультиплекативной ошибкой, имеется класс \mathcal{APX} -полных задач, для которых существование эффективного алгоритма нахождения решений с любой наперед заданной точностью (с мультиплекативной ошибкой, не превосходящей $1+\varepsilon$) представляется крайне маловероятным (см. раздел 6.3.7).

1.1.4 «Сортировка слиянием»

Рассмотрим довольно простые задачи сортировки или упорядочивания, алгоритмы для которых имеют не более чем квадратичную сложность.

На примере этих задач мы изложим некоторые общие приемы построения и анализа эффективных алгоритмов, такие, как «разделяй и властвуй», анализ сложности в среднем вместо худшего случая, использование вероятностных алгоритмов вместо детерминированных.

Задача 9. «Сортировка» (*Sorting*)

Имеется произвольный массив A : a_1, \dots, a_n .

Требуется путем сравнений отсортировать этот массив таким образом, чтобы элементы расположились в порядке возрастания (или убывания), то есть $a_{i1} \leq a_{i2} \leq \dots \leq a_{in}$.

Слияние — это объединение двух или более упорядоченных массивов в один упорядоченный. Пусть требуется упорядочить массив по убыванию. Для этого сравниваются два наименьших элемента обоих массивов, и наименьший из них выводится как наименьший элемент суммарного массива, затем процедура повторяется (см. процедуру «merge» в алгоритме 10 «Mergesort»). Нетрудно видеть, что слияние двух упорядоченных последовательностей A и B длин $|A|$ и $|B|$ происходит за $O(|A| + |B|)$ сравнений.

Упражнение 1.1.6. Найдите такие упорядоченные последовательности A и B (без ограничения их длины), на которых процедура «merge» в алгоритме 10 «Mergesort» выполнит $|A| + 1$ сравнений.

Упражнение 1.1.7. Найдите такие упорядоченные последовательности A и B , на которых процедура «merge» в алгоритме 10 «Mergesort» выполнит $|A| + |B| - 1$ сравнений.

Рассмотрим сортировку слиянием (алгоритм 10 «Mergesort»). Исходный массив A длины n делится пополам. Затем производится рекурсивная сортировка каждой половинки и их слияние. Пусть $T(n)$ — число сравнений, достаточное для сортировки слиянием.

Можем записать рекуррентное соотношение:

$$T(n) = 2T(n/2) + O(n).$$

Чтобы решить это уравнение, применим метод подстановки, и просто проверим, что $T(n) = cn \log_2 n$ является решением при подходящей константе c .

Алгоритм 10 Сортировка слиянием

```

def mergesort(L):
    M = len(L) / 2 # середина
    if M == 0:
        # если список короче двух
        return L # сортировать нечего
    return merge(mergesort(L[:M]),
                mergesort(L[M:]))


def merge(A, B):
    C = []
    while A or B: # пока оба списка непусты
        if not B or (A and A[0] < B[0]):
            # A_0 переносим из A в C
            C.append( A.pop(0) )
        else:
            # B_0 переносим из B в C
            C.append( B.pop(0) )
    return C

```

```

mergesort: [3, 4, 1, 5, 9, 2, 6, 3, 5]
mergesort: [3, 4, 1, 5]
mergesort: [3, 4]
mergesort: [3]
mergesort: [4]
    merge: [3] [4] -> [3, 4]
mergesort: [1, 5]
mergesort: [1]
mergesort: [5]
    merge: [1] [5] -> [1, 5]
    merge: [3, 4] [1, 5] -> [1, 3, 4, 5]
mergesort: [9, 2, 6, 3, 5]
mergesort: [9, 2]
mergesort: [9]
mergesort: [2]
    merge: [9] [2] -> [2, 9]
mergesort: [6, 3, 5]
mergesort: [6]
mergesort: [3, 5]
mergesort: [3]
mergesort: [5]
    merge: [3] [5] -> [3, 5]
    merge: [6] [3, 5] -> [3, 5, 6]
    merge: [2, 9] [3, 5, 6] -> [2, 3, 5, 6, 9]
    merge: [1, 3, 4, 5] [2, 3, 5, 6, 9] -> [1, 2, 3, 3, 4, 5, 5, 6, 9]
0тсортирано: [1, 2, 3, 3, 4, 5, 5, 6, 9]

```

Несмотря на асимптотически оптимальное поведение алгоритма (алгоритм сортирует входной массив за время $O(n \log n)$ для любых входных данных), алгоритм сортировки слиянием используется в основном в области СУБД, где требуется сортировать огромные массивы информации, во много раз превосходящие объем оперативной памяти (и следовательно, которых невозможно загрузить в память целиком) — тогда весьма кстати, то, что в каждой итерации слияния к массивам нужен только последовательный доступ — а известно, что последовательное чтение данных с диска на несколько порядков эффективней произвольного доступа (*random access*).

Все остальные алгоритмы сортировки требуют произвольного доступа к сортируемым данных, и поэтому, не используются для сортировки больших массивов СУБД. Хотя последние успехи в области дешевой памяти (SSD-устройства хранения, и т. п.), возможно изменят архитектуру СУБД и используемые там алгоритмы.

1.1.5 «Быстрая сортировка»

«Быстрая сортировка». Анализ «в среднем» и вероятностная версия для «худшего случая».

Алгоритм быстрой сортировки⁹ в худшем случае (на некоторых входных массивах) использует время $\Omega(n^2)$, что, например, хуже, чем сложность в наихудшем случае алгоритма 10 «Mergesort». Однако он является очень популярным алгоритмом, т.к. анализ «в среднем» и опыт реального использования показали его эффективность.

Ключевая идея алгоритма 12 «Quicksort» заключается в процедуре «partition» (см. алгоритм 11), которая за линейное время от размера массива осуществляет такую перестановку элементов относительно некоторой «оси» — заданного значения, равного одному из значений сортируемого интервала массива, что переставленный массив состоит из трех интервалов, идущих по порядку:

⁹«QuickSort», разработан Хоаром (C.A.R. Hoare), см. [Hoa62].

Алгоритм 11 Процедура «partition» в «Quicksort»

```

def partition(A, L, H, pivot):
    # Перестановка в интервале [L \dots H] массива A.
    # Возникают 3 интервала: элементы меньшие, равные и большие pivot.
    i = L
    while i <= H:
        if A[i] < pivot:    # Если элемент меньше оси
            A[i], A[L] = A[L], A[i]  # переставляем A_i \leftrightarrow A_L
            L = L + 1      # и сужаем область слева
            i = i + 1
        elif A[i] > pivot: # Если элемент больше оси
            A[i], A[H] = A[H], A[i]  # переставляем A_i \leftrightarrow A_H
            H = H - 1      # и сужаем область справа
        else:
            i = i + 1          # оставляем на месте и сужаем слева.
    return L, H  # \forall i: L \leq i \leq H: A_i=pivot

```

1. элементы, меньшие «оси»;
2. элементы, равные «оси»;
3. элементы, большие «оси».

Первый и последний из упомянутых интервалов могут остаться неупорядоченными, поэтому далее они рекурсивно сортируются процедурой «QSortInterval».

Алгоритм 12 Быстрая сортировка/«Quicksort»

```
def pivotTrivial(A, left, right):
    return A[left] # ось - первый элемент в интервале

def pivotRandom(A, left, right): #случайный выбор оси
    return A[LOG.random_source.randrange(left, right)]

def quicksort(A, pivotFunc):
    def QSortInterval(L, H):
        if L < H: # Если в интервале [$A_L$ \dots $A_H$] хотя бы два элемента
            newL, newH = partition(A, L, H, pivotFunc(A, L, H))
            QSortInterval(L, newL - 1)
            QSortInterval(newH + 1, H)

    QSortInterval(0, len(A) - 1)
    return A
```

Упражнение 1.1.8. Модифицируйте алгоритм 12 «Quicksort», чтобы остался только один рекурсивный вызов.

Эффективность алгоритма существенно зависит от функции выбора «оси» (параметр «pivotFunc» в алгоритме 12 «Quicksort»). Например, если назначать «осью» значение, которое больше или меньше всех элементов, то алгоритм вовсе не будет работать — разбиение никак не будет уменьшать сортируемый интервал и алгоритм зациклится (хотя это конечно противоречит определению «оси», как одного из эле-

ментов текущего интервала, и этот пример приведен только для иллюстрации важности выбора «оси»).

Одним из тривиальных решений может стать выбор осью первого элемента из сортируемого интервала (т. е. использование функции «trivialPivot» из алгоритма 12 «Quicksort»). Для некоторых последовательностей это может приводить к неоптимальному поведению, когда при разбиении один из крайних интервалов намного меньше другого или вовсе пустой. Пример таких «плохих» данных (для выбора оси «trivialPivot») показан на рис. 1.5. Видно, что на каждой рекурсии сортируемый интервал уменьшается только на один осевой элемент, и алгоритм 12 «Quicksort» на «плохих» входных данных выполняет $\Omega(N)$ рекурсий, а общее количество операций (с учетом операций в процедуре «partition») будет $\Omega(\sum_{i=1}^N i) = \Omega(n^2)$.

Упражнение 1.1.9. Сколько памяти может использовать алгоритм 12 «Quicksort» в наихудшем случае?

Осталось выяснить, часто ли встречаются наихудшие (для «pivotTrivial») случаи на множестве входных данных.

Допустим, входные данные случайны, и их распределение таково, что в каждом интервале длины N , встречающемся в процедуре «partition», первый элемент с равной вероятностью может быть k -м ($1 \leq k \leq N$) по величине, «разбивая» тем самым входной интервал на интервалы длины $k - 1$ и $N - k$. Тогда можно записать следующую рекурсивную оценку математического ожидания сложности алгоритма:

$$\begin{aligned} T(N) &= O(N) + \frac{1}{N} \left(\sum_{k=1}^N (T(k-1) + T(N-k)) \right) = \\ &= O(N) + \frac{2}{N} \sum_{k=1}^{N-1} T(k). \quad (1.3) \end{aligned}$$

Рис. 1.5: Работа «Quicksort» с разными методами выбора оси

```
Sorting (pivotFunc=pivotTrivial): [3, 1, 4, 5, 2, 6, 3, 7]
partition [3, 1, 4, 5, 2, 6, 3, 7] *3*-> [1, 2] [3, 3] [6, 5, 7, 4]
partition [1, 2] *1*-> [] [1] [2]
partition [6, 5, 7, 4] *6*-> [5, 4] [6] [7]
partition [5, 4] *5*-> [4] [5] []
Sorted: [1, 2, 3, 3, 4, 5, 6, 7]
```

Плохой набор для выбора осью первого элемента.

```
Sorting (pivotFunc=pivotTrivial): [1, 6, 2, 5, 3, 7, 4]
partition [1, 6, 2, 5, 3, 7, 4] *1*-> [] [1] [2, 5, 3, 7, 4, 6]
partition [2, 5, 3, 7, 4, 6] *2*-> [] [2] [3, 7, 4, 6, 5]
partition [3, 7, 4, 6, 5] *3*-> [] [3] [4, 6, 5, 7]
partition [4, 6, 5, 7] *4*-> [] [4] [5, 7, 6]
partition [5, 7, 6] *5*-> [] [5] [6, 7]
partition [6, 7] *6*-> [] [6] [7]
Sorted: [1, 2, 3, 4, 5, 6, 7]
```

Эффективно сортируется вероятностной осью.

```
Sorting (pivotFunc=pivotRandom): [1, 2, 3, 4, 5, 6, 7]
partition [1, 2, 3, 4, 5, 6, 7] *1*-> [] [1] [3, 4, 5, 6, 7, 2]
partition [3, 4, 5, 6, 7, 2] *4*-> [3, 2] [4] [7, 6, 5]
partition [3, 2] *3*-> [2] [3] []
partition [7, 6, 5] *7*-> [6, 5] [7] []
partition [6, 5] *6*-> [5] [6] []
Sorted: [1, 2, 3, 4, 5, 6, 7]
```

Проверим разумную гипотезу, что

$$T(N) = O(N \log N), \quad (1.4)$$

т. е. что $\exists C : T(N) \leq CN \log N$.

Действительно (логарифм двоичный):

$$\sum_{k=1}^{N-1} k \log k \leq \log N \cdot \frac{N^2}{2} \quad (1.5)$$

Подставляя (1.5) в (1.3), убеждаемся, что $T(N) = CN \log N$ для некоторой константы C .

Однако допущенное нами предположение о равномерности распределения «осей» относительно интервалов может не выполняться на практике. На практике возможна ситуация, когда для выбранной функции «pivotFunc» будут попадаться исключительно «плохие» входные данные. В таком случае, когда нельзя «сделать случайными» входные данные, можно внести элемент случайности непосредственно в сам алгоритм. В частности, чтобы сохранилась оценка математического ожидания (1.4), достаточно сделать вероятностным выбор осевого элемента из разделяемого интервала — см. использование функции «pivotRandom» на рис. 1.5.

Другие интересные примеры анализа поведения алгоритмов на случайных данных будут рассмотрены в главе 3.

В завершение раздела приведем небольшую «карту памяти» для запоминания рассмотренных в разделе понятий.



Рис. 1.6: Карта-памятка раздела 1.1

1.2 Формально об алгоритмах. Несложно о сложности

1.2.1 «RAM»: машины с произвольным доступом

При написании данного раздела использован обзор [КР96].

В предыдущих разделах мы довольствовались качественным, интуитивным понятием «эффективного» алгоритма. Для построения же математической теории сложности алгоритмов, разумеется, необходимо строгое количественное определение меры эффективности.

Опыт, накопленный в теории сложности вычислений, свидетельствует, что наиболее удобным и адекватным способом сравнения эффективности разнородных алгоритмов является понятие *асимптотической сложности*, рассмотрению которого и посвящен настоящий параграф.

Первое, о чем следует договориться, — это выбор вычислительной модели, в которой конструируют-ся наши алгоритмы. Оказывается, что как раз этот вопрос не имеет слишком принципиального значения для теории сложности вычислений, и тот уровень строгости, на котором мы работали в разделе 1.1 (число выполненных операторов на языке Python), оказывается почти приемлемым. Главная причина такого легкомысленного отношения к выбору модели состоит в том, что существуют весьма эффективные способы моделирования (или *трансляции программ* в более привычных терминах) одних естественных вычислительных моделей с помощью других. При этих моделях сохраняется класс эффективных алгоритмов и, как правило, алгоритмы более эффективные в одних моделях оказываются более эффективными и в других.

Сначала рассмотрим модель, наиболее напоминающую современный компьютер, программируемый непосредственно в терминах инструкций процессора (или на языке Assembler): *random access machines (RAM)*¹⁰, т. е. «машины с произвольным доступом к памяти».

¹⁰ В теории сложности вычислений под машинами традиционно понимают single-purpose machines, т. е. машины, каждая из

RAM-машину составляют следующие компоненты (рис. 1.7).

- Конечная входная *read-only* лента, на которую записываются входные данные.
- Полубесконечная¹¹ выходная *write-only* лента, куда записывается результат работы машины.
- Бесконечное число регистров r_0, r_1, r_2, \dots , каждый из которых может хранить произвольное целое число (изначально везде записаны нули). Регистр r_0 является выделенным и называется «сумматором» — этот регистр используется при арифметических операциях как накопитель, т. е. как второй operand и место хранения результата.
- Программа, состоящая из конечного числа инструкций, каждая из которых содержит адрес и команду с операндом. Список команд приведен в таблице 1.2. Существенно, что в качестве операнда можно использовать как произвольный регистр, так и регистр, номер которого хранится в другом регистре — это так называемая косвенная адресация.
- Регистр-счетчик «PC» — указатель текущей команды.

Машина последовательно, такт за тактом, выполняет команды, на которые указывает регистр «PC», читает входную ленту, изменяет значения регистров и записывает результат на выходную ленту.

Легко видеть, что высокоуровневые конструкции языков программирования, типа циклов, легко моделируются на ассемблере RAM-машин (см. рис. 1.8).

Переменные других типов (булевы, строки, структуры) тоже можно моделировать с помощью целых чисел и косвенной адресации.

которых создана для решения какой-либо одной фиксированной задачи. В привычных терминах это скорее программы.

¹¹Ограниченнная с одной стороны и неограниченная с другой.

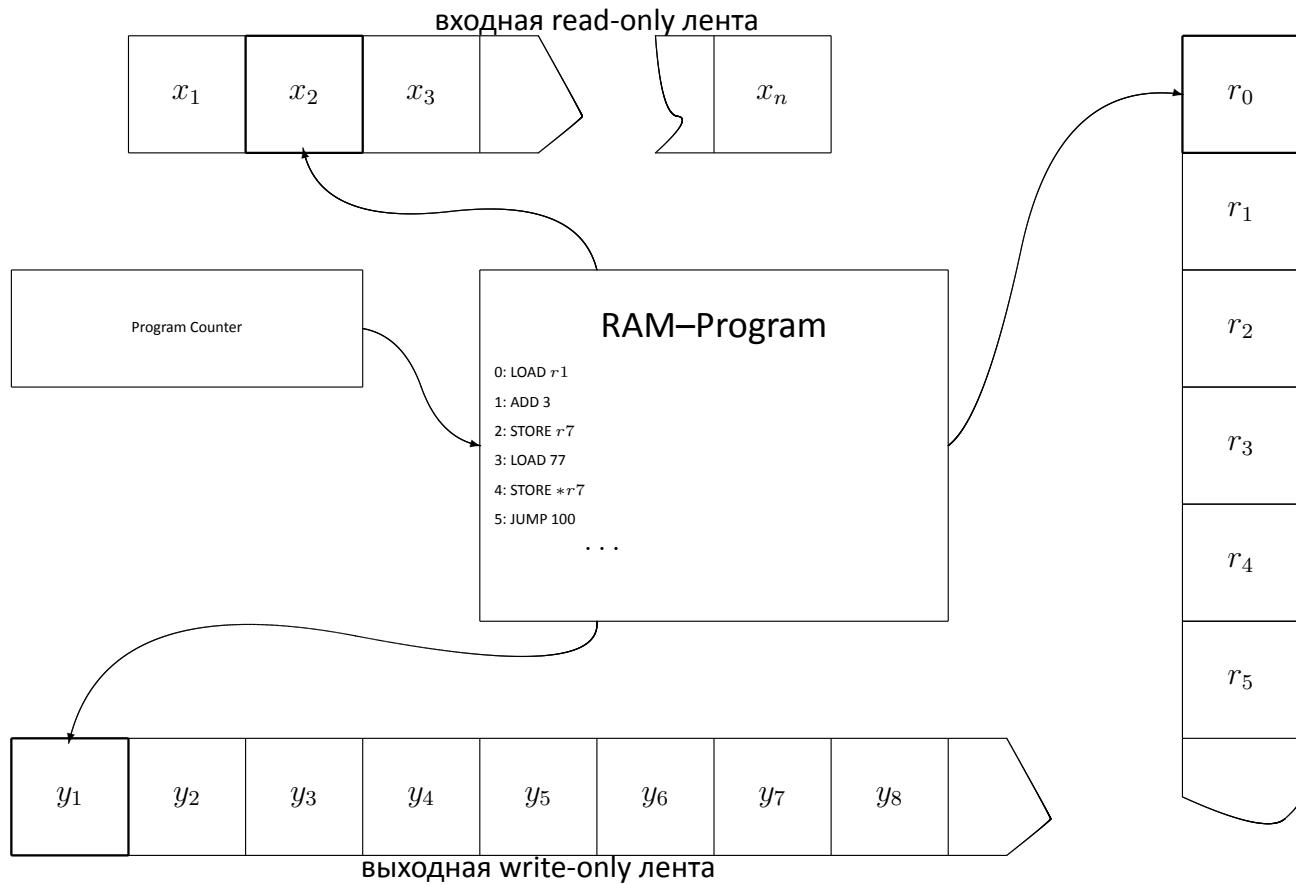


Таблица 1.2: RAM-машина: Список команд

LOAD OP	$r_0 \leftarrow OP$	Загрузить операнд в сумматор.
STORE OP	$r_{OP} \leftarrow r_0$	Сохранить сумматор в регистре.
ADD OP	$r_0 \leftarrow r_0 + OP$	Прибавить операнд к сумматору.
SUB OP	$r_0 \leftarrow r_0 - OP$	Вычесть операнд из сумматора.
READ OP	$r_{OP} \leftarrow input$	Загрузить ячейку из входной ленты в r_{OP} и перейти к следующей.
WRITE OP	$OP \rightarrow output$	Записать OP в текущую ячейку выходной ленты и сдвиг к следующей.
JUMP OP	$PC \leftarrow OP$	Установить счетчик команд в OP .
JGTZ OP	$PC \leftarrow OP : r_0 > 0$	Установить счетчик команд в OP , если $r_0 > 0$.
JZERO OP	$PC \leftarrow OP : r_0 = 0$	Установить счетчик команд в OP , если $r_0 = 0$.
HALT		Остановить работу.

Операнд OP может быть:

- целым числом, например, 7, -1917 ;
- регистром, например, r_{12}, r_{34} ;
- значением регистра, указанного в другом регистре. Например, операнд $*r_{14}$ означает значение регистра, номер которого указан в регистре r_{14} .

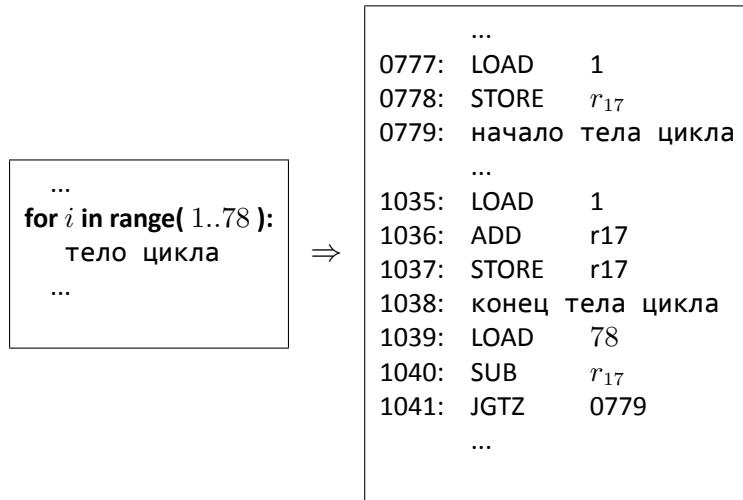


Рис. 1.8: Моделирование циклов для RAM

Обратите внимание, что, несмотря на «примитивный ассемблер», RAM-машины потенциально мощнее любых существующих компьютеров и физически нереализуемы, т.к. оперируют бесконечной памятью, где доступ к любой ячейке-регистру осуществляется мгновенно при выполнении соответствующей инструкции, и каждая ячейка этой памяти может содержать произвольное целое число (т. е. не ограниченна по размеру). Но эта модель уже дает возможность вводить более или менее формальные определения времени выполнения программы (как число тактов до остановки машины) и, соответственно, сложности алгоритма.

Если мы будем принимать в расчет только число выполненных команд, то определим так называемые

однородные меры сложности. В англоязычной литературе RAM с однородной мерой сложности называются *unit-cost RAM*.

Более реалистично было бы учитывать битовый размер операндов при выполнении каждой команды. Например, можно считать, что время выполнения каждой команды есть единица плюс величина пропорциональная сумме логарифмов значений operandov¹², если они есть (т. е. их суммарной битовой длине), а общее время работы программы на рассматриваемых входных данных — суммарное время выполнения всех индивидуальных команд. Таким образом, мы можем определить различные логарифмические меры сложности.

Очевидно, логарифмическая сложность всегда больше однородной. С другой стороны, обратите внимание, что определенная нами RAM-машина не включает в число основных операций умножение и деление, хотя в литературе такой вариант машин со случайным доступом часто рассматривается. Допустим, что однородное время работы некоторой машины с произвольным доступом к памяти без умножения и деления при работе на входных данных x_1, \dots, x_m , битовая длина каждого из которых не больше n , равно t . Так как при выполнении любого индивидуального оператора максимальная битовая длина может возрасти не более чем на 1, в ходе выполнения всей программы встречаются лишь числа битовой длины не более $(t + n)$, и, стало быть, логарифмическая сложность превышает однородную не более чем в $2(t + n)$ раз. В частности (см. обсуждение в разделе 1.1), с точки зрения эффективности однородная и логарифмическая сложности равносильны, и выбор одной из них в основном определяется внутренней спецификой рассматриваемой задачи.

Если же добавить умножение и деление к списку элементарных операций, то это исключит равносильность однородной и логарифмической сложности. Следующий известный способ быстро переполнить память карманного калькулятора (алгоритм 13) имеет однородную сложность порядка $(t + 1)$ и логарифмическую порядка экспоненты — 2^t .

¹²Более корректно логарифм брать от модуля операнда плюс 2, чтобы логарифм был положительным.

Алгоритм 13 Переполнение памяти умножением

```
def full_memory(loop):
    result = 2
    for i in range(loop):
        result = result * result
    return result
```

loop: 2 result: 16
loop: 3 result: 256
loop: 4 result: 65536
loop: 5 result: 4294967296
loop: 6 result: 18446744073709551616
loop: 7 result: 340282366920938463463374607431768211456
loop: 8 result: 115792089237316195423570985008687907853269984665640564039457

Алгоритм 14 Тривиальное вычисление $a \cdot b$ на RAM

```
def mult(a, b):
    mul = 0
    for i in range(b):
        mul = mul + a
    return mul
```

$2 \times 3 = 6$
$6 \times 2 = 12$
$183 \times 34 = 6222$

Как показывает обсуждение в разделе 1.1, этот алгоритм не может считаться эффективным с точки зрения логарифмической сложности (хотя и по совершенно другим причинам, нежели переборные алгоритмы), и, чтобы избежать неприятных эффектов такого рода, мы не включаем умножение (и тем более деление) в список основных операций.

В тех же случаях, когда умножение используется «в мирных целях», его в большинстве случаев можно промоделировать с помощью сложения очевидным образом (см. алгоритм 14 «Умножение»).

В заключение можно отметить, что некоторые частные случаи модели RAM описывают популярные модели вычислений — *неветвящиеся программы*, булевы схемы¹³ (см. раздел 6.4), а сама модель RAM является основой модели, используемой в теории параллельных вычислений — PRAM¹⁴ (см. раздел 4.3.1).

1.2.2 Сложность в худшем случае

Чаще всего для характеризации сложности алгоритма в качестве основных вычислительных ресурсов рассматривают *время*, затраченное алгоритмом на вычисление, и *использованную память*. Понятно, что потребление этих ресурсов может зависеть от размера входных данных.

Значит, сначала надо определить, что понимать под размером входных данных. По аналогии с однородными и логарифмическими мерами сложности возникают две различные возможности. Мы можем определить размер данных как *размерность* задачи, т. е. число ячеек RAM, занятых входными данными. Например, размерность задач 4 «TSP», 5 «SPP», рассмотренных в разделе 1.1, равна $\frac{n(n-1)}{2}$ (где n — число вершин в исходном графе). Либо мы можем определить размер входных данных как *суммарную битовую длину* записи всех входных параметров ($\sum_{1 \leq i < j \leq n} \lceil \log_2(|d_{ij}| + 1) \rceil$ для вышеупомянутых задач), т. е. число заполненных ячеек на входной ленте МТ.

Большинство алгоритмов, рассмотренных нами в разделе 1.1, обладает тем приятным свойством, что время их работы относительно однородной меры сложности зависит лишь от размерности задачи и не зависит от входных значений ячеек.

Более того, тем же свойством будет обладать любой алгоритм, все циклы в котором имеют вид:

for $i = 1$ **to** ограничитель **do**,

¹³Boolean circuits.

¹⁴Parallel RAM.

где ограничитель — некоторая явная функция, зависящая только от размерности задачи. Мы будем называть алгоритмы с этим свойством *однородными*.

Сложность однородного алгоритма — это целочисленная функция целого аргумента $t(n)$, равная времени его работы для входных данных размерности n .

К сожалению, эффективность и однородность — требования, зачастую плохо совместимые между собой, поэтому нам необходимо иметь меру сложности, пригодную также для неоднородных алгоритмов.

Естественная попытка определить сложность неоднородного алгоритма как функцию $t(R_1, \dots, R_m)$ от всего массива входных данных при ближайшем рассмотрении оказывается неудовлетворительной: эта функция несет в себе огромное количество избыточной информации, с трудом поддается вычислению или хотя бы оцениванию в явном виде и позволяет сравнивать между собой различные алгоритмы лишь в исключительных случаях. Поэтому, чтобы получить на самом деле работающее на практике определение сложности, нам необходимо каким-нибудь образом свести эту функцию к функции *одного* (или в крайнем случае небольшого числа) целого аргумента (как в случае однородных алгоритмов). Имеются два принципиально различных подхода к решению этой задачи.

Самый распространенный подход — это рассмотрение *сложности в наихудшем случае*. Мы полагаем $t(n)$ равным

$$\max \left\{ t(R_1, \dots, R_m) \mid \sum_{i=1}^m \lceil \log_2(|R_i| + 1) \rceil \leq n \right\}.$$

Иными словами, $t(n)$ — это то время работы, которое данный алгоритм может гарантировать *затруднительно*, если известно, что суммарная битовая длина входных данных не превышает n . При этом время работы алгоритма для некоторых (или даже для «большинства») таких входных данных может быть существенно меньше $t(n)$. Более формальное определение можно найти в разделе [6.1.2](#).

1.2.3 Сложность в среднем

Другая возможность заключается в рассмотрении *сложности в среднем*. В этой постановке вводится некоторое (например, равномерное) вероятностное распределение на массивах входных данных битовой длины $\leq n$, и сложностью $t(n)$ называется *математическое ожидание* времени работы нашего алгоритма на случайному входе, выбранном в соответствии с этим распределением. Пример анализа в среднем нами уже рассмотрен для алгоритма быстрой сортировки. Более интересные примеры такого анализа рассмотрены далее, где приводятся полиномиальные в среднем алгоритмы для ряда \mathcal{NP} -трудных задач (см. раздел 3.5).

Следует отметить, что, несмотря на свою внешнюю привлекательность, сложность в среднем в настоящее время не может конкурировать со сложностью в худшем случае по объему проводимых исследований, хотя интенсивность исследований в области вероятностного анализа поведения алгоритмов в последние годы заметно увеличивается. Сложность в среднем также естественным образом возникает в криптографии при анализе стойкости различных крипtosистем.

1.2.4 Полиномиальные алгоритмы

Итак, алгоритмы для одной и той же задачи целесообразно сравнивать по *асимптотическому* поведению их сложности $t(n)$. В теории сложности вычислений традиционно игнорируются мультипликативные константы в оценках сложности. Вызвано это тем, что в большинстве случаев эти константы привносятся некоторыми малоинтересными внешними факторами, и их игнорирование позволяет абстрагироваться от такого влияния. Приведем несколько примеров таких факторов.

Во-первых, при несущественных модификациях вычислительной модели сложность обычно изменяется не более чем на некоторую (как правило, довольно небольшую) мультипликативную константу. Например, если мы разрешим в машинах со случайным доступом оператор цикла **for**, то ввиду наличия

моделирования (см. рис. 1.8) сложность от этого может уменьшиться не более чем в два раза. Следовательно, если мы игнорируем мультиплекативные константы, можем свободно использовать в машинах со случайным доступом **for-do-endfor** циклы и другие алголоподобные конструкции, не оговаривая всякий раз это специально.

Во-вторых, при переходе от двоичной к восьмеричной или десятичной системе счисления длина битовой записи, а следовательно, и сложность полиномиальных алгоритмов изменяется не более чем на мультиплекативную константу. Таким образом, путем игнорирования таких констант также можно абстрагироваться от вопроса о том, в какой системе счисления мы работаем, и вообще не указывать явно в наших оценках основание логарифмов.

Кроме того, имеет место приблизительное равенство: «физическое время работы = среднее время выполнения одного оператора \times однородная сложность».

В этой формуле теория сложности вычислений отвечает за второй сомножитель, а первый зависит от того, насколько грамотно составлена программа, от качества транслятора на язык низшего уровня (что определяет, например, сколько времени занимает доступ к памяти или жесткому диску), быстродействия вычислительной техники и т. д. Хоть эти вопросы и важны, они выходят за рамки теории сложности вычислений, и игнорирование мультиплекативных констант позволяет абстрагироваться от них и выделить интересующий нас второй сомножитель в чистом виде.

По этой причине в теории сложности вычислений широкое распространение получило обозначение « O -большое». Типичный результат выглядит следующим образом: «данный алгоритм работает за время $O(n^2 \log n)$ », и его следует понимать как «существует такая константа $c > 0$, что время работы алгоритма в наихудшем случае не превышает $cn^2 \log n$, начиная с некоторого n ».

Практическая ценность асимптотических результатов такого рода зависит от того, насколько мала неявно подразумеваемая константа c . Как мы уже отмечали выше, для подавляющего большинства известных алгоритмов она находится в разумных пределах, поэтому, как правило, имеет место следующий тезис: ал-

горитмы, более эффективные с точки зрения их асимптотического поведения, оказываются также более эффективными и при тех сравнительно небольших размерах входных данных, для которых они реально используются на практике.

Теория сложности вычислений по определению считает, что алгоритм, работающий за время $O(n^2 \log n)$ лучше алгоритма с временем работы $O(n^3)$, и в подавляющем большинстве случаев это отражает реально существующую на практике ситуацию. И такой способ сравнения эффективности различных алгоритмов оказывается достаточно универсальным — сложность в наихудшем случае с точностью до мультипликативной константы для подавляющего большинства реально возникающих алгоритмов оказывается достаточно простой функцией¹⁵, и практически всегда функции сложности разных алгоритмов можно сравнить между собой по их асимптотическому поведению.

Таким образом, можно определять классы алгоритмов, замкнутых относительно выбора вычислительной модели.

Определение 1.2.1. Алгоритм называется **полиномиальным**, если его сложность в наихудшем случае $t(n)$ ограничена сверху некоторым полиномом (многочленом) от n .

Это определение оказывается вполне независимым относительно выбора вычислительной модели: любые «разумные» модели (модели, в которых не может происходить экспоненциального нарастания памяти, как в алгоритме 13 «Переполнение памяти умножением») оказываются равносильными в смысле этого определения.

Например, нетрудно убедиться, что алгоритм 7 «Дейкстры» и алгоритм 9 «MST Прима» полиномиальны. Для задачи «умножение двух чисел» также имеется полиномиальный алгоритм, хотя алгоритм 14 «Умно-

¹⁵Следует оговориться, что иногда рассматривают функции сложности, зависящие от двух-трех параметров, например, от числа вершин и ребер входного графа, или вертикального и горизонтального размера входной матрицы. Тем не менее, даже в таких случаях часто удается провести сравнение асимптотической сложности различных алгоритмов.

жение» и неполиномиален. В самом деле, его однородная сложность равна $R_1 + 1$, что не ограничено никаким полиномом от битовой длины $\lceil \log_2(|R_1| + 1) \rceil$.

Полиномиальный алгоритм для умножения легко строится, например, с помощью моделирования обыкновенного умножения столбиком (проделайте это в качестве упражнения).

Полиномиальные алгоритмы противопоставляются экспоненциальным, т. е. таким, для которых $t(n) \geq 2^{n^k}$ для некоторой фиксированной константы $k \geq 1$ и почти всех n . Например, экспоненциальными являются алгоритмы [6](#) «TSP-перебор», [13](#) «Переполнение памяти умножением» (относительно однородной меры сложности) и [14](#) «Умножение».

В теории сложности задач дискретной оптимизации естественным образом возникают по крайней мере две модификации понятия полиномиального алгоритма, одно из которых является его усилением, а другое — ослаблением.

Предположим, что полиномиальный алгоритм к тому же еще и однороден. Тогда он обладает следующими двумя свойствами (второе из которых, как мы видели выше, влечет первое):

1. длина битовой записи всех чисел, возникающих в процессе работы алгоритма, ограничена некоторым полиномом от длины битовой записи входных данных (это свойство на самом деле присуще любому полиномиальному алгоритму);
2. число арифметических операций (однородная сложность) ограничено полиномом от *размерности* задачи.

Основная причина, по которой мы предпочли не рассматривать умножение и деление в качестве основных операций, состоит в том, что это нарушило бы импликацию $2 \Rightarrow 1$ (см. алгоритм [13](#) «Переполнение памяти умножением»). Но если мы разрешим умножение и деление и при этом потребуем, чтобы свойство [1](#) обеспечивалось внутренней структурой алгоритма, то мы получим класс алгоритмов, называемых в литературе *сильно полиномиальными алгоритмами*.

Таким образом, сильно полиномиальные алгоритмы обобщают однородные полиномиальные алгоритмы. Хорошим примером сильно полиномиального алгоритма, который, по-видимому, нельзя привести к однородному виду (т. е. избавиться от делений и умножений), служит известный алгоритм Гаусса решения систем линейных уравнений.

Итак, отличительной чертой сильно полиномиальных алгоритмов является то, что время их работы ограничено полиномом от *размерности* задачи. На другом полюсе находятся *псевдополиномиальные алгоритмы*, в которых требуется, чтобы время работы алгоритма было полиномиально лишь от суммы *абсолютных значений* (а не от битовой длины записи) входящих в задачу числовых параметров. Типичные примеры таких алгоритмов приведены в разделе 2.2.1.

1.2.5 Полиномиальность и эффективность

Может ли полиномиальный алгоритм быть неэффективным? Разумеется, может, если в полиномиальной оценке времени его работы $t(n) \leq C \cdot n^k$ либо мультипликативная константа C , либо показатель k чрезмерно велики.

Опыт показывает, что такое случается крайне редко, и подавляющее большинство полиномиальных алгоритмов для естественных задач удовлетворяет оценке $t(n) \leq 10 \cdot n^3$. В тех же немногих случаях, когда полиномиальный алгоритм оказывается малопригодным с практической точки зрения, это обстоятельство всегда стимулирует бурные исследования по построению на его основе действительно эффективного алгоритма, что, как правило, приводит к интересным самим по себе побочным следствиям. Хрестоматийным примером такого рода может служить метод эллипсоидов для линейного программирования [Kha79], стимулировавший бурное развитие целого направления ([Kar84]), получившего название «метод внутренней точки» и занявшего в настоящее время лидирующие позиции в разработке наиболее эффективных алгоритмов для задач линейного программирования большой размерности.

Кстати, одна из самых известных современных открытых проблем в теории алгоритмов (и вообще математики) заключается в ответе на вопрос: «существует ли сильно полиномиальный алгоритм для задачи линейного программирования?» ([[Sma00](#)]).

Упражнение 1.2.1. Посмотрите на каждый из приведенных ранее алгоритмов и определите, является ли он полиномиальным, сильно полиномиальным или псевдополиномиальным?

Может ли неполиномиальный алгоритм быть эффективным? Ответ утвердительный, по меньшей мере, по трем причинам. Во-первых, может случиться так, что примеры, на которых время работы алгоритма велико, настолько редки, что вероятность обнаружить хотя бы один из них на практике пренебрежимо мала. В математических терминах это означает, что алгоритм полиномиален в среднем относительно любого «разумного» вероятностного распределения и, по-видимому, под эту категорию попадает симплекс-метод (см. [[Cxp91](#)]).

Во-вторых, многие псевдополиномиальные алгоритмы являются эффективными, когда возникающие на практике числовые параметры не слишком велики.

Наконец, *субэкспоненциальные* алгоритмы, время работы которых, например, имеет вид $O(n^{c \log \log \log n})$, при всех разумных длинах входа n могут быть весьма практически эффективными.

Но в заключение еще раз подчеркнем, что примеров задач, на которых нарушается основополагающее равенство

«полиномиальность»=«эффективность»

крайне мало по сравнению с числом примеров, на которых оно блестяще подтверждается.

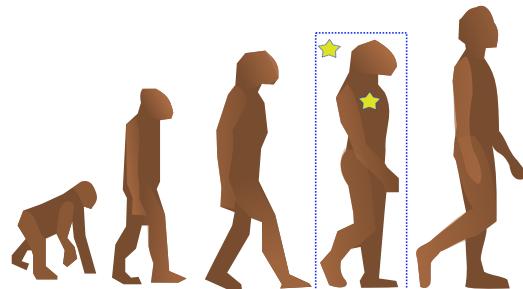
Отметим, что единственную серьезную конкуренцию машинам RAM как основы для построения теории сложности вычислений на сегодняшний день составляют так называемые машины Тьюринга, которые мы будем рассматривать в разделе [6.1](#). Их принципиальное отличие состоит в отсутствии непрямой

индексации: после проведения каких-либо действий с некоторой ячейкой процессор («головка») может перейти лишь в одну из «соседних» ячеек. Машины Тьюринга удобны при рассмотрении комбинаторных задач с небольшим количеством числовых параметров, а также (в силу своей структурной простоты) для теоретических исследований.

Глава 2

Аппроксимация с гарантированной точностью

2.1 Алгоритмы с оценками точности



*Приближенные алгоритмы с оценками точности.
Жадные алгоритмы для задач типа покрытия
и упаковки. Приближенные алгоритмы на графах.*

Одним из общих подходов к решению \mathcal{NP} -трудных задач, который активно развивается в настоящее

время, является разработка приближенных алгоритмов с гарантированными оценками качества получаемого решения.

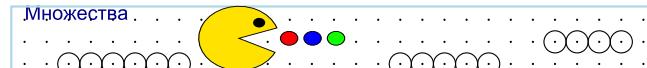
Точность приближенного алгоритма характеризует *мультипликативная ошибка*, которая показывает, в какое максимально возможное число раз может отличаться полученное решение от оптимального (по значению заданной целевой функции).

Определение 2.1.1. Алгоритм называется *C-приближенным*, если при любых исходных данных он находит допустимое решение со значением целевой функции, отличающимся от оптимума не более чем в C раз.

Заметим, что иногда также говорят об $\frac{1}{C}$ -приближенных алгоритмах, причем смысл отклонения (больше или меньше единицы) обычно ясен из контекста и направления оптимизации (максимизации или минимизации). Мультипликативная ошибка может быть константой или зависеть от параметров входной задачи. Наиболее удачные приближенные алгоритмы позволяют задавать точность своей работы. В качестве примера перечислим приближенные алгоритмы, которые мы рассмотрим далее в этой книге:

- 2-приближенный алгоритм для задачи о рюкзаке из раздела [2.1.3](#).
- $3/2$ -приближенный алгоритм для метрической задачи коммивояжера из раздела [2.1.4](#).
- 0,878-приближенный алгоритм для задачи о максимальном разрезе в графе из раздела [4.4.2](#).
- 2-приближенный алгоритм для минимального вершинного покрытия из раздела [2.1.2](#).
- $(1 + \ln m)$ -приближенный алгоритм для задачи о покрытии из раздела [2.1.1](#).
- $(1 + \varepsilon)$ -приближенный алгоритм для задачи о рюкзаке из раздела [2.2.2](#).

2.1.1 Жадные алгоритмы для «Покрытия множеств»



Жадные алгоритмы для задачи о покрытии подмножеств, локальный алгоритм для вершинного покрытия.

Одной из простейших эвристик, часто применяемых при решении различных задач, является так называемая **жадная эвристика**. Для некоторых задач, например, для задачи об оствомном дереве минимального веса в графе, эта эвристика фактически позволяет найти оптимальное решение (см. алгоритм 9 «MST Прима»). В более трудных случаях она позволяет получать гарантированные оценки точности получаемого с ее помощью решения.

Рассмотрим сейчас жадную эвристику для классической \mathcal{NP} -трудной задачи — задачи о покрытии.

Задача 10. «Покрытие множества»¹.

- *Множество X , $|X| = m$.*
- *Семейство подмножеств $\{S_1, \dots, S_n\}$, $S_j \subseteq X$.*
- *$X = \bigcup_{j=1}^n S_j$ (покрытие гарантировано).*

¹В англоязычной литературе — *Set Cover*.

Найти минимальное по мощности множество подмножеств, покрывающее X :

$$\begin{aligned} |J| &\rightarrow \min, \\ J &\subseteq \{1, 2, \dots, n\}, \\ X &= \cup_{j \in J} S_j. \end{aligned}$$

Число $|J|$ называется размером минимального покрытия.

Известно, что задача о покрытии \mathcal{NP} -полнна. По этой причине трудно надеяться на существование полиномиального алгоритма точного ее решения.

Рассмотрим алгоритм 15, который действует в соответствии с простейшей жадной эвристикой: «На каждом шаге выбирать максимально непокрытое подмножество», т. е. подмножество, содержащее максимальное число элементов, не покрытых на предыдущих шагах. Далее мы оценим гарантированную им точность.

Теорема 1. Алгоритм 15 гарантирует точность $1 + \ln m$ для задачи 10 «Set Cover».

Доказательство. Пусть,

X_k — число непокрытых элементов после k -го шага.

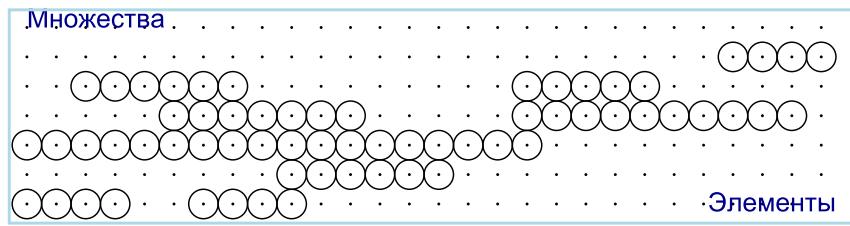
$M = |J|$ — размер минимального покрытия.

Так как на любом шаге k , X_k непокрытых элементов мы можем «накрыть» минимальным покрытием размера M , при этом обязательно будет подмножество, которое накроет $\frac{X_k}{M}$ непокрытых элементов (иначе будет противоречие — минимальное покрытие не сможет накрыть X_k непокрытых элементов, и, следовательно, и полное множество тоже). Значит, жадный алгоритм на k -м шаге накроет не менее $\frac{X_k}{M}$ непокрытых элементов, и получится

Алгоритм 15 Жадный алгоритм для задачи о покрытии

«Каждый раз выбирать максимально непокрытое подмножество».

На рисунке столбцы соответствуют элементам множества, строки — подмножествам. т. е. каждый графический объект с координатами x, y представляет принадлежность некоторого элемента x подмножеству y . Показаны фазы работы жадного алгоритма — темным цветом выделены выбранные алгоритмом подмножества.



$$X_{k+1} \leq X_k - \frac{X_k}{M} = X_k(1 - 1/M). \quad (2.1)$$

$$X_k \leq m(1 - 1/M)^k \leq m \exp\left(-\frac{k}{M}\right).$$

Найдем наибольшее k_0 , при котором $m \exp\left(-\frac{k_0}{M}\right) \geq 1$.

Тогда при $k_1 = k_0 + 1$ выполнено $X_{k_1} < 1$, что означает, что все элементы покрыты. При этом $k_1/M \leq 1 + \ln m$, значит, размер покрытия, построенного жадным алгоритмом, превосходит минимальное не более чем в $1 + \ln m$ раз. \square

Упражнение 2.1.1. Докажите неравенство (2.1).

Упражнение 2.1.2. Постройте пример, где оценка $O(1 + \ln m)$ мультипликативной ошибки жадного алгоритма достигается по порядку. Указание: достаточно рассмотреть случай, когда размер минимального покрытия $M = 2$.

Упражнение 2.1.3. Постройте пример, где оценка $1 + \ln m$ достигается асимптотически.

Встречаются и другие оценки точности жадного алгоритма. Доказано, например (см. [Joh73]), что мультипликативная ошибка жадного алгоритма не превосходит

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{m}.$$

Асимптотически это выражение как раз равно $\ln m$. Более точная оценка получена в [Sla96], где доказано, что мультипликативная ошибка жадного алгоритма не превосходит $\ln m - \ln \ln m + O(1)$.

Возникает законный вопрос: а насколько хороши полученные оценки? Достижимость оценки означает лишь, что мы достаточно точно оценили ошибку жадного алгоритма. Однако вполне возможно существование других полиномиальных алгоритмов с лучшими оценками. Вопрос этот на самом деле является непростым, и только относительно недавно на него удалось дать удовлетворительный ответ [Fei98].

Оказалось, что при разумных теоретико-сложностных предположениях, а именно $\mathcal{NP} \not\subset \mathcal{DTIME}(n^{O(\log \log n)})^2$, никакой полиномиальный алгоритм для задачи о покрытии не может иметь мультипликативную ошибку меньше $(1 - \delta) \ln m$ для любого фиксированного $\delta > 0$.

Все это является следствием так называемой \mathcal{PCP} -теоремы и ее связями с неаппроксимируемостью (см. теорему 37).

²См. определения 6.2.4 на стр. 259 и 6.1.7 на стр. 250.

Но, с другой стороны, существует ряд классических разновидностей задачи о покрытии с лучшими оценками аппроксимации. Например, «максимизационный» вариант задачи 10 «Set Cover».

Задача 11. « K -покрытие»

Задано:

- Множество X , $|X| = m$.
- Семейство подмножеств $\{S_1, \dots, S_n\}$, $S_j \subseteq X$.

Выбрать такие k подмножеств, чтобы мощность их объединения была максимальна:

$$\begin{aligned} |\cup_{j \in J} S_j| &\rightarrow \max, \\ J &\subseteq \{1, 2, \dots, n\}, \\ |J| &= k. \end{aligned}$$

Оказалось, жадный алгоритм дает хорошие результаты и для этой задачи.

Теорема 2. Жадный алгоритм 15 является $(1 - e^{-1})$ -приближенным алгоритмом для задачи 11 « K -покрытие».

Доказательство. Пусть Y_i обозначает число покрытых элементов после i -го шага жадного алгоритма, а M — оптимум в задаче о k -покрытии. Справедливо следующее неравенство (см. упражнение 2.1.1):

$$Y_{i+1} \geq Y_i + \frac{M - Y_i}{k} = \frac{M}{k} + Y_i \left(1 - \frac{1}{k}\right).$$

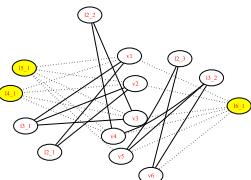
Решая это рекуррентное неравенство, получим:

$$\begin{aligned}
 Y_k &\geq \frac{M}{k} \left[1 + \left(1 - \frac{1}{k}\right) + \left(1 - \frac{1}{k}\right)^2 + \dots + \left(1 - \frac{1}{k}\right)^{k-1} \right] = \\
 &= \frac{M}{k} \left[\frac{1 - \left(1 - \frac{1}{k}\right)^k}{\frac{1}{k}} \right] = M \left[1 - \left(1 - \frac{1}{k}\right)^k \right] \geq \\
 &\geq M \left[1 - \exp \left\{ -\frac{k}{k} \right\} \right] = M(1 - e^{-1}).
 \end{aligned}$$

□

Упражнение 2.1.4. Покажите, что оценка точности $1 - e^{-1}$ для задачи о k -покрытии асимптотически достижима.

2.1.2 Приближенные алгоритмы для «Вершинного покрытия»



Жадный и локальный алгоритмы для вершинного покрытия.

Рассмотрим один из вариантов задачи о покрытии на графах.

Определение 2.1.2. «Вершинное покрытие»³.

Для неориентированного графа $G = (V, E)$ подмножество вершин $V' \subseteq V$ называется **вершинным покрытием**, если каждое ребро из E содержит хотя бы одну вершину из V' .

Задача 12. «Минимальное вершинное покрытие»⁴.

Дан неориентированный граф $G = (V, E)$ (множество вершин V , множество ребер E). Найти вершинное покрытие минимальной мощности.

Первая интуитивная идея для решения задачи 12 «Min Vertex Covering» — использовать жадный алгоритм, выбирающий на каждом шаге вершину, имеющую на этом шаге наибольшее число непокрытых ребер (см. алгоритм 16).

Несмотря на «интуитивную разумность», оказалось, что этот алгоритм не обеспечивает даже константной точности.

Например, «плохим» для этого алгоритма будет граф, образованный следующим образом (см. рис. 2.1):

- Введем n вершин «первого уровня» v_1, \dots, v_n .
- Разобьем вершины первого уровня на $\lfloor \frac{n}{2} \rfloor$ непересекающихся пар, добавим $\lfloor \frac{n}{2} \rfloor$ вершин второго уровня $l_1^2, \dots, l_{\lfloor n/2 \rfloor}^2$, и соединим каждую добавленную вершину со «своей» парой вершин первого уровня.
- Добавим $\lfloor \frac{n}{3} \rfloor$ вершин третьего уровня $l_1^3, \dots, l_{\lfloor n/3 \rfloor}^3$, причем соединим каждую из них со всеми вершинами из одной из непересекающихся троек вершин первого уровня v_1, \dots, v_n .

...

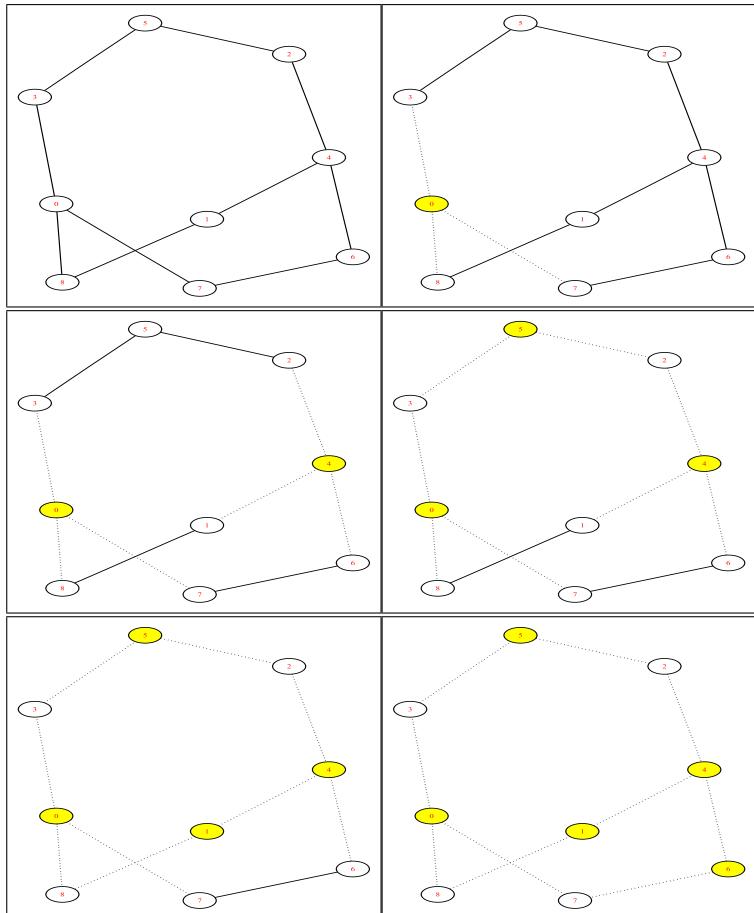
³В англоязычной литературе — *Vertex Cover*.

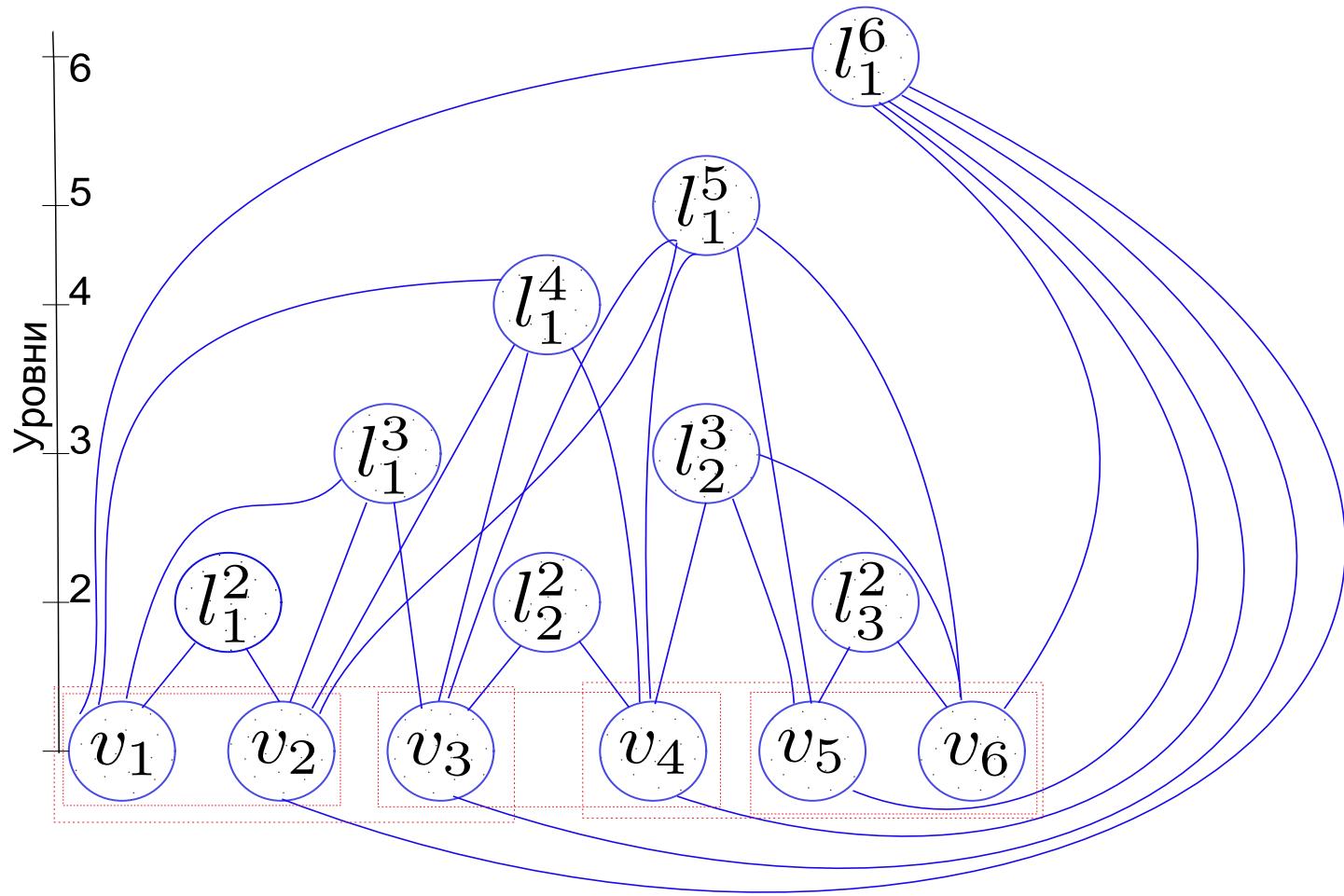
⁴В англоязычной литературе — *Min Vertex Covering*.

Алгоритм 16 Жадный алгоритм для вершинного покрытия

```
def greedy_vertex_covering(G):
    C = set()
    while G.number_of_edges() > 0: #цикл  $O(|E|)$ 
        v_max = max_degree = -1
        for n in G.nodes(): # цикл  $\approx O(|V|)$ 
            if G.degree(n) > max_degree:
                v_max, max_degree = n, G.degree(n)
        C.add(v_max)
        for v in G.neighbors(v_max):
            #удаляем покрытые ребра
            G.remove_edge(v_max, v)
    return C
```

- Закрашенные вершины составляют вершинное покрытие .
- Пунктиром показаны покрытые (удаленные) ребра.





- Добавим последнюю ($\lfloor \frac{n}{n-1} \rfloor$) вершину $n - 1$ -го уровня, соединенную с $n - 1$ вершиной первого уровня.

Для всех вершин уровня $k > 1$ степень вершины будет ровно k , степень вершин первого уровня будет не больше $n - 2$ (степень $n - 2$ будет у вершин первого уровня, которые оказались связанными со всеми вершинами верхних уровней).

И хотя очевидным вершинным покрытием для такого графа являются вершины v_1, \dots, v_n , алгоритм 16 начнет «есть» вершины начиная с верхних уровней, при этом степени вершин v_1, \dots, v_n будут уменьшаться, так что в вершинное покрытие войдут только все вершины l_k^j , и не войдет ни одна из v_1, \dots, v_n . Работа алгоритма на таком графе (для $n = 6$) показана на рис. 2.2.

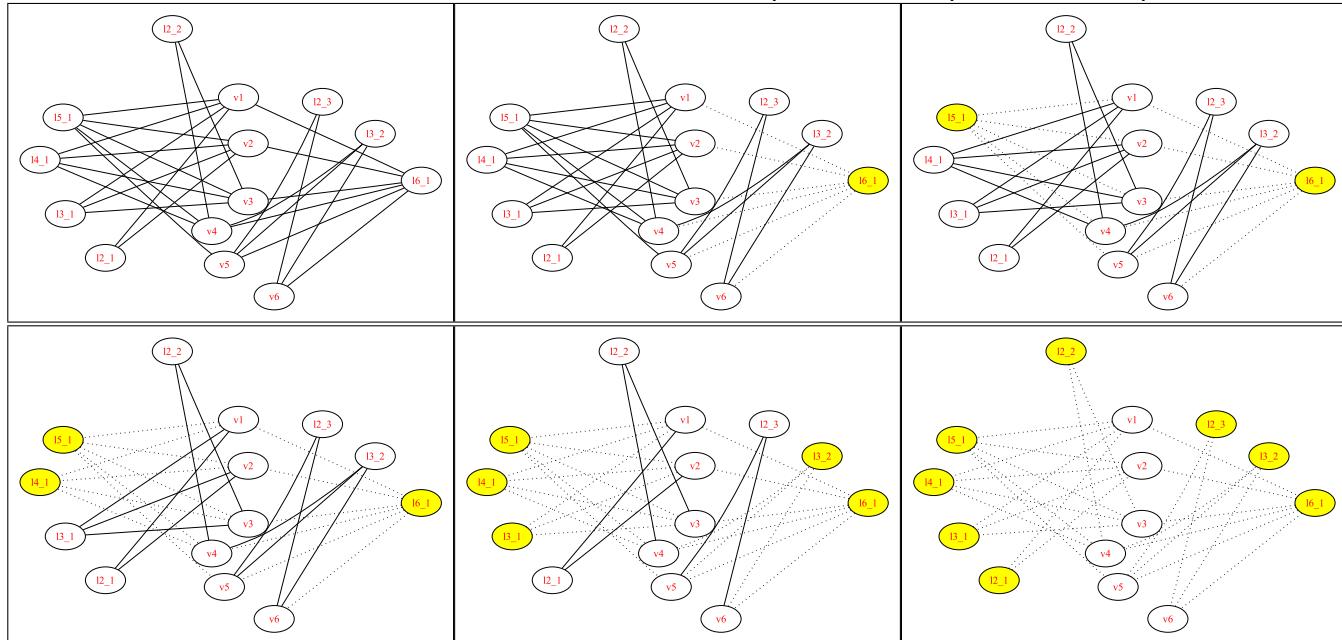
Посчитаем f_g число вершин, выбранных алгоритмом 16, и сравним его с f^* — оптимальным размером вершинного покрытия для этого графа:

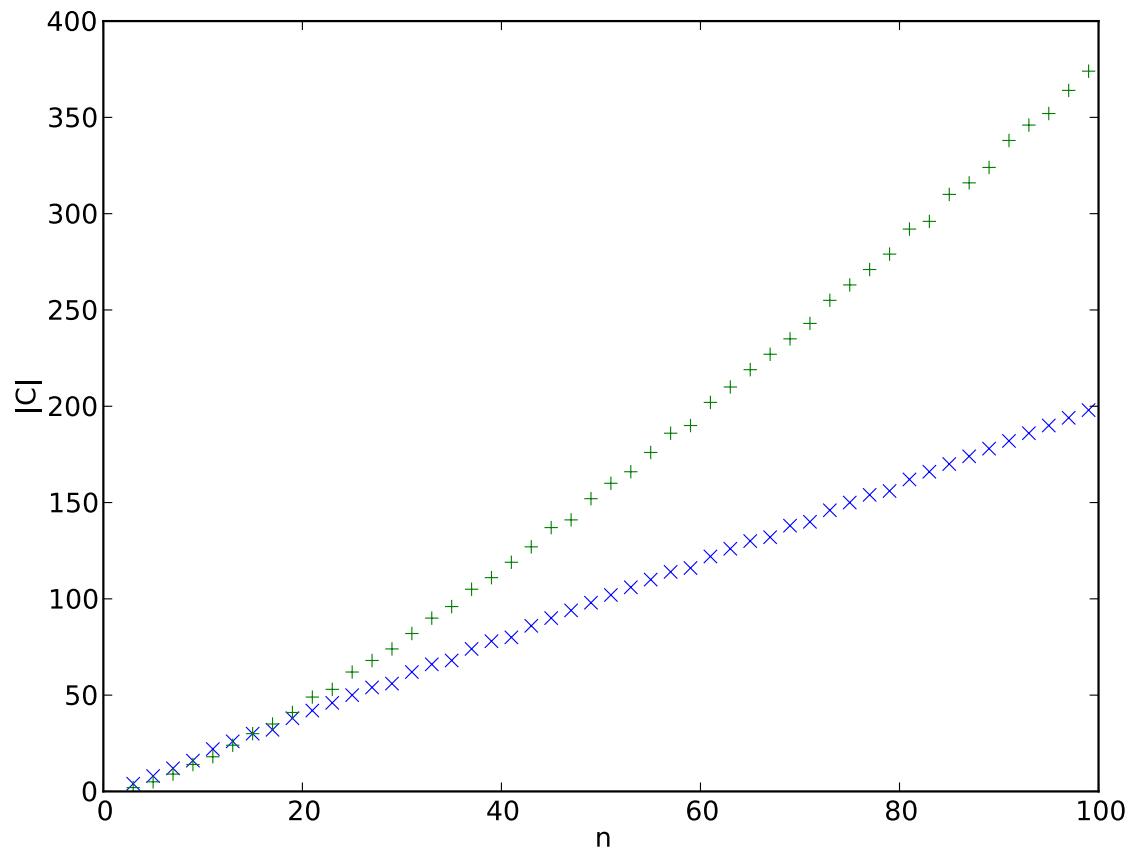
$$\begin{aligned} f_g &= \sum_{j=2}^{n-1} \left\lfloor \frac{n}{j} \right\rfloor > \sum_{j=2}^{n-1} \left(\frac{n}{j} - 1 \right) = n \sum_{j=2}^{n-1} \frac{1}{j} - (n-2) > \\ &> n \int_2^n \frac{1}{x} dx - n = n(\ln n - \ln 2 - 1) \geq f^*(\ln n - \ln 2 - 1). \end{aligned}$$

И хотя жадный алгоритм не смог обеспечить никакой констатной точности приближения для задачи 12 «Min Vertex Covering», оказалось, что есть еще более простой, не «жадный», а скорее «ленивый», локальный алгоритм 17, который является 2-приближенным. Алгоритм 17 просто добавляет для первого попавшегося непокрытого ребра обе его вершины, исключает покрытые этими вершинами ребра, после чего цикл повторяется.

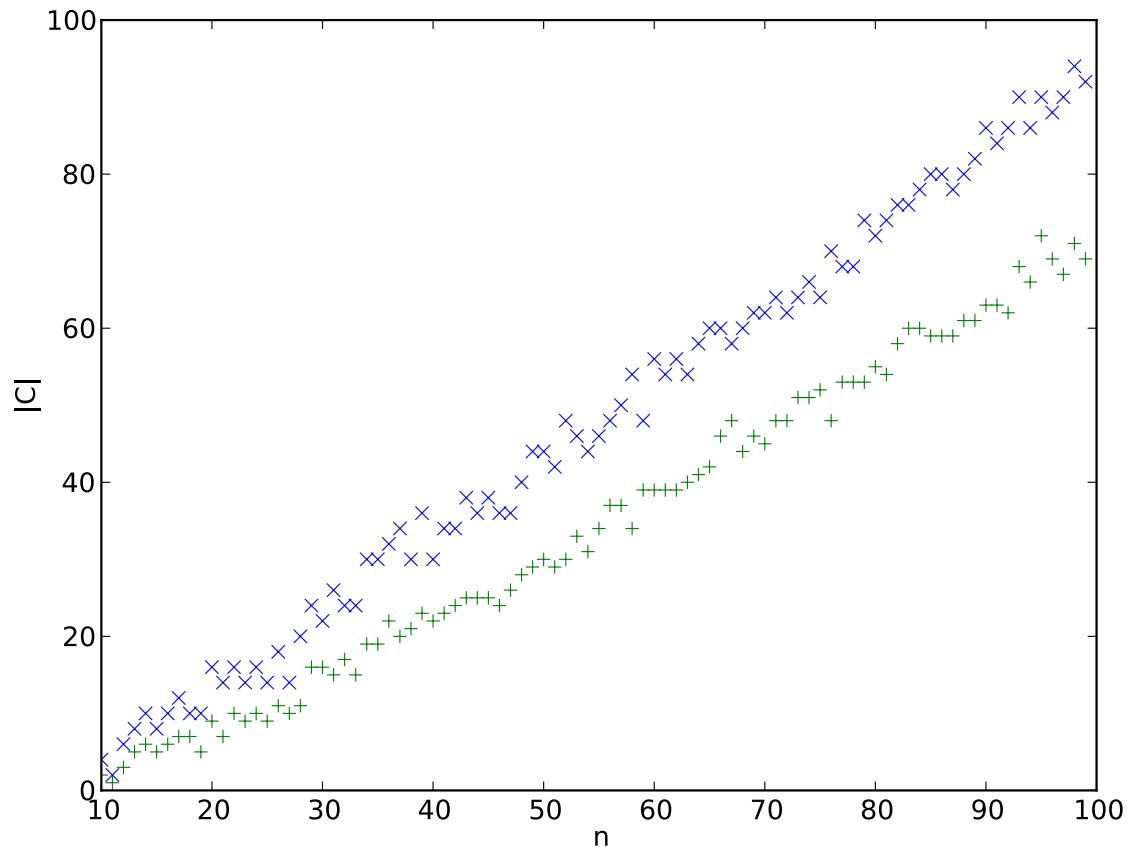
Несмотря на «вопиющую неоптимальность» — добавление в вершинное покрытие для выбранного ребра **обеих** вершин, этот алгоритм может гарантировать константную точность, в частности, на рис. 2.3 видно, как он выигрывает у жадного алгоритма на «плохих» для жадного алгоритма графах.

Рис. 2.2: Неоптимальность жадного алгоритма для вершинного покрытия





+ — размер вершинного покрытия для жадного алгоритма 16.

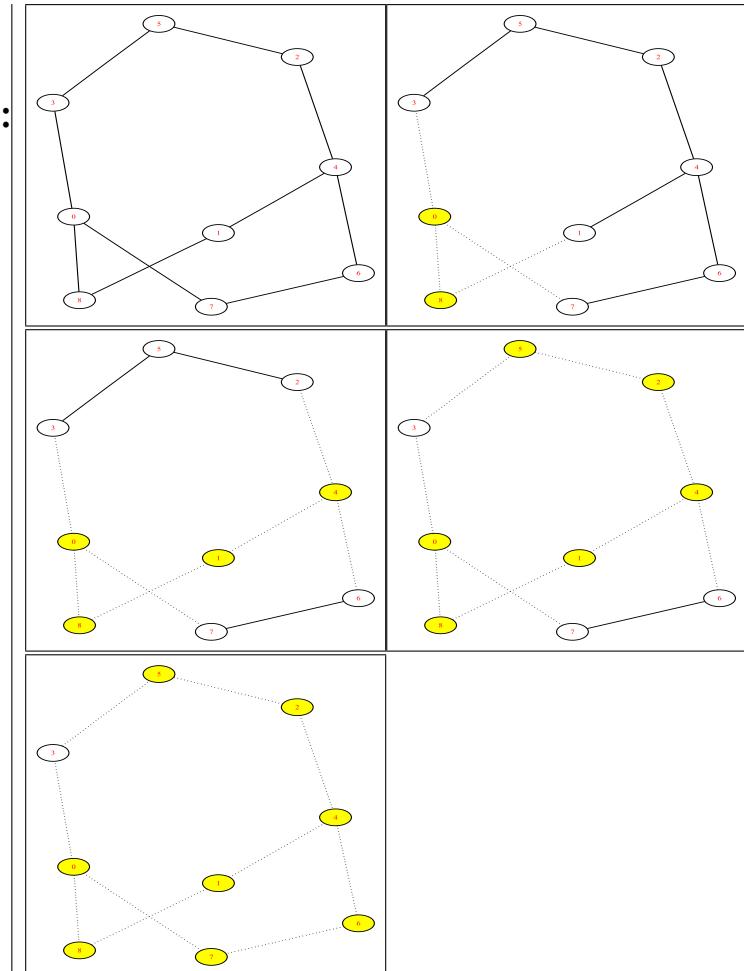


+ — размер вершинного покрытия для жадного алгоритма 16.

Алгоритм 17 «Ленивый» алгоритм для задачи 12 «Min Vertex Covering»

```
def lazy_vertex_covering(G):
    C = set()
    while G.number_of_edges() > 0:
        #первое попавшееся ребро
        (u, v) = G.edges()[0]
        C.add(u)
        C.add(v)
        for w in G.neighbors(u):
            G.remove_edge(u, w)
        for w in G.neighbors(v):
            G.remove_edge(v, w)
    return C
```

- Закрашенные вершины составляют вершинное покрытие .
- Пунктиром показаны покрытые (удаленные) ребра.



Итак, докажем точность приближения.

Определение 2.1.3. *Паросочетание*⁵ — подмножество ребер графа, такое, что никакие два ребра из этого подмножества не инцидентны какой-либо одной вершине.

Теорема 3. Алгоритм 17 гарантирует точность 2 для задачи 12 «Min Vertex Covering».

Доказательство. Рассмотрим множество E^* «попавшихся алгоритму» ребер, вершины которых добавлялись в покрытие C . Это множество E^* является паросочетанием, т. е. все ребра из него попарно несмежны, а вершины этих ребер образуют вершинное покрытие. Любое вершинное покрытие, включая минимальное, должно содержать по крайней мере одну вершину для каждого ребра из E^* , поэтому $OPT \geq |E^*|$, где OPT — число вершин в минимальном покрытии. Учитывая, что $|E^*| = |C|/2$, получаем:

$$|C| \leq 2 \cdot OPT.$$

□

С другой стороны, на случайных графах алгоритм 17 несколько проигрывает «жадному» алгоритму 16 (см. рис. 2.4). Учитывая, что оба алгоритма — быстрые, на практических задачах можно запускать оба алгоритма, отбирая лучший ответ.

Упражнение 2.1.5. Найдите приближенный алгоритм с точностью $\frac{1}{2}$ для нахождения максимального подмножества дуг, не образующих цикл, в ориентированном подграфе.

Упражнение 2.1.6. Найдите приближенный алгоритм с точностью $\frac{1}{2}$ для нахождения максимального (по включению) паросочетания максимального объема.

⁵Matching в англоязычной литературе.

Упражнение 2.1.7. Найдите приближенный алгоритм с точностью $\frac{1}{2}$ для нахождения максимального (по включению) паросочетания минимального объема.

Упражнение 2.1.8. Студент предложил для задачи 2.1.2 «Vertex Covering» приближенный алгоритм с точностью $\frac{1}{2}$: найти в графе дерево (методом поиска в ширину), выкинуть из него листья, а оставшееся объявить вершинным покрытием. Прав ли он? Докажите или опровергните.

Упражнение 2.1.9. Студент предложил для задачи 2.1.2 «Vertex Covering» приближенный алгоритм с точностью $\frac{1}{2}$: найти в графе дерево (методом поиска в глубину), выкинуть из него листья, а оставшееся объявить вершинным покрытием. Прав ли он? Докажите или опровергните.

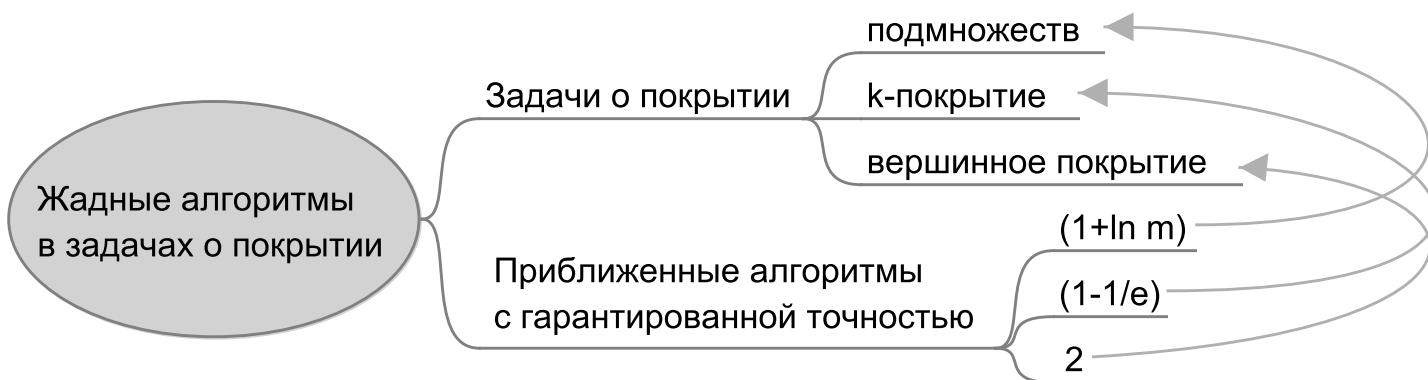


Рис. 2.5: Карта-памятка разделов 2.1.2 и 2.1.1

2.1.3 Жадный алгоритм для «Рюкзака»



Жадный алгоритм для задачи о рюкзаке с гарантированной точностью 2.

Как мы уже отмечали, жадные эвристики являются простейшими и самыми популярными методами решения различных вычислительно трудных задач. Посмотрим, что эта эвристика может дать для задачи о рюкзаке.

Рассмотрим следующую задачу.

Задача 13. «0–1 Рюкзак (Knapsack)»

Даны:

$c_1, \dots, c_n, c_j \in N$ — «стоимости» предметов;

$a_1, \dots, a_n, a_j \in N$ — «размеры» или «веса»;

$B \in N$ — «размер рюкзака».

Найти максимальное значение f^* целевой функции

$$f \equiv \sum_{i=1}^n c_i x_i \rightarrow \max$$

с ограничением на размер «рюкзака»:

$$\sum_{i=1}^n a_i x_i \leq B, \quad x_i \in \{0, 1\}.$$

Содержательно задача означает выбор предметов с наибольшей суммарной стоимостью, умещающихся в рюкзак заданного размера. Эта задача часто возникает при выборе оптимального управления в различных экономико-финансовых областях (распределение бюджета отдела по проектам и т. п.).

Рассмотрим, какого результата можно добиться, используя, как в разделе 2.1.1, «жадный подход».

Первая идея, которая обычно возникает при знакомстве с этой задачей, это выбирать предметы по убыванию их относительной стоимости, помещая в рюкзак все, что помещается (см. алгоритм 18).

К сожалению, о качестве эвристики алгоритма 18 ничего хорошего утверждать нельзя.

Упражнение 2.1.10. Докажите, что для любого числа k можно представить входной набор данных, для которых алгоритм 18 выберет набор, который в k раз хуже оптимального.

Понятно, что проблема состоит в том, что «польстившись» на первый небольшой, но «относительно дорогой» предмет, алгоритм рискует пропустить большой и ценный предмет из оптимального набора.

Оказывается, гарантированное качество работы этой эвристики можно улучшить, если после окончания ее работы сравнить стоимость полученного допустимого решения с максимальным коэффициентом c_{\max} и в соответствии с максимумом выбрать либо «жадное решение», либо один предмет с максимальной стоимостью (мы считаем, что размеры всех предметов не превосходят размер рюкзака — в противном случае их просто можно исключить из рассмотрения).

Получится так называемый *модифицированный жадный алгоритм* (см. алгоритм 19 «Рюкзак-Жадный»), для которого уже можно гарантировать качество найденного решения.

Теорема 4. Для значения решения f_G , полученного модифицированным жадным алгоритмом 19 «Рюкзак-Жадный», и оптимального значения f^* для задачи 13 «Knapsack» выполняется

$$\frac{f^*}{2} \leq f_G.$$

Алгоритм 18 Жадный алгоритм для задачи 13 «Knapsack»

```

def usefulness((c, a)):
    return a*1.0/c

def KnapsackGreedy(T, B):
    T.sort(key=usefulness)
    Cg = Ag = 0
    for c, a in T:
        # если лезет в рюкзак
        if Ag + a <= B:
            # берем предмет (c,a)
            Ag = Ag + a
            Cg = Cg + c
    return Cg

```

Вес рюкзака B= 10 кг

Входной массив T <= [(3, 6), (4, 3), (5, 2), (6, 5), (7, 5), (8, 1)]

Отсортированный T => [(8, 1), (5, 2), (7, 5), (4, 3), (6, 5), (3, 6)]

Берем предмет: <= (\$8 , 1 кг)

Берем предмет: <= (\$5 , 2 кг)

Берем предмет: <= (\$7 , 5 кг)

Набран рюкзак стоимостью \$20

Алгоритм 19 «Модифицированный жадный» для «Рюкзака»

```
def knapsack_greedy(T, B):
    T.sort(key=usefullness)
    Сmax = Cg = Ag = 0
    for c, a in T:
        if a <= B:
            Сmax = max(c, Сmax)
            # если лезет в рюкзак
        if Ag + a <= B:
            # берем предмет (c,a)
            Ag = Ag + a
            Cg = Cg + c
            #выбираем, что больше
    return max(Cg, Сmax)
```

Вес рюкзака B= 10 кг
 Входной массив T <= [(3, 6), (4, 3), (5, 2), (6, 5), (7, 5), (8, 1)]
 Отсортированный T => [(8, 1), (5, 2), (7, 5), (4, 3), (6, 5), (3, 6)]
 Берем предмет: <= (\$8 , 1 кг)
 Берем предмет: <= (\$5 , 2 кг)
 Берем предмет: <= (\$7 , 5 кг)
 Cg=\$20 или Сmax=\$8 ?
 Набран рюкзак стоимостью \$20

Вес рюкзака B= 100 кг
 Входной массив T <= [(10, 1), (170, 100), (50, 40), (40, 20)]
 Отсортированный T => [(10, 1), (40, 20), (170, 100), (50, 40)]
 Берем предмет: <= (\$10 , 1 кг)
 Берем предмет: <= (\$40 , 20 кг)
 Берем предмет: <= (\$50 , 40 кг)
 Cg=\$100 или Сmax=\$170 ?
 Набран рюкзак стоимостью \$170

Доказательство. Обозначим набор предметов, выбранных жадным алгоритмом 19 «Рюкзак-Жадный», через S_g , а стоимость этого набора через C_g . Рассмотрим набор \tilde{S}_g , полученный жадным алгоритмом, которому разрешено (после того, как рюкзак будет заполнен) взять еще один предмет, и обозначим его стоимость $\tilde{C}_g = \sum_{i \in \tilde{S}_g} c_i$ (см. рис. 2.6).

Заметим, что по определению $\tilde{C}_g \leq C_g + c_{max}$. С другой стороны, $\tilde{C}_g \geq f^*$. Чтобы убедиться в этом, заметим, что любой предмет (обозначим его индекс через p) из оптимального набора или содержится в \tilde{S}_g , или его относительная стоимость не превышает относительной стоимости любого предмета из \tilde{S}_g :

$$\frac{c_p}{a_p} \leq \frac{c_i}{a_i} \quad \forall i \in \tilde{S}_g.$$

Принимая во внимание, что стоимость равна площади, из рисунка 2.6 мы видим, что высота любого предмета вне набора \tilde{S}_g не больше высоты предметов в наборе \tilde{S}_g . Добавление более «низких» прямоугольников из оптимального набора вместо каких-либо прямоугольников из \tilde{S}_g может только уменьшить суммарную площадь, т. е. стоимость. Поэтому $\tilde{C}_g \geq f^*$. Отсюда следует:

$$\begin{aligned} C_g + c_{max} &\geq \tilde{C}_g \geq f^*, \\ 2f_G &\equiv 2 \max(C_g, c_{max}) \geq C_g + c_{max} \geq f^*. \end{aligned}$$

Откуда:

$$f_G \geq \frac{f^*}{2}.$$

□

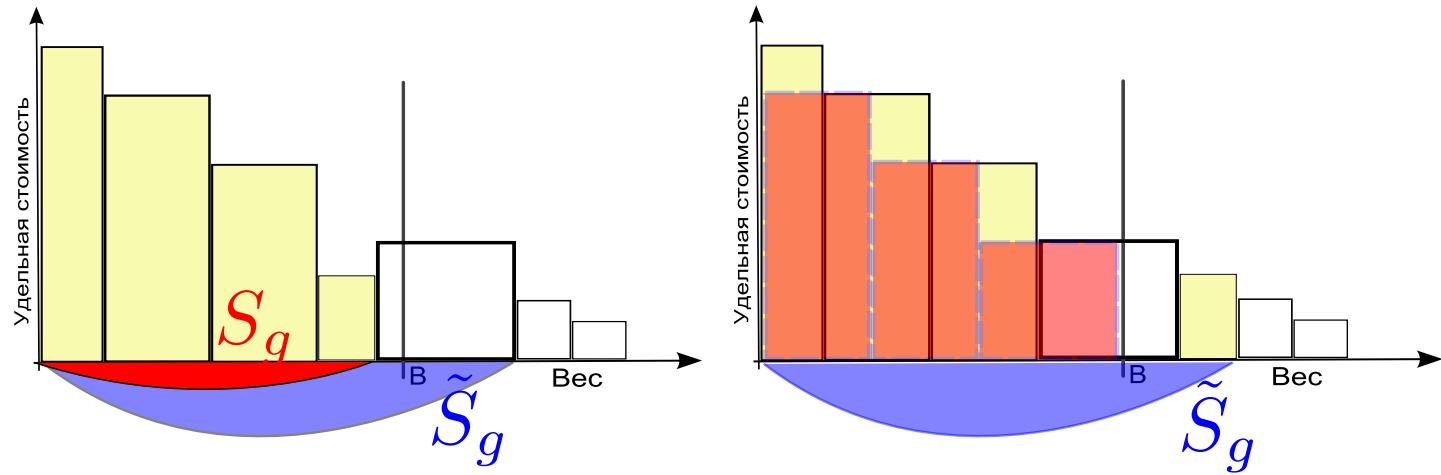


Рис. 2.6: Наборы S_g и \tilde{S}_g в «жадном» алгоритме для «рюкзака»

2.1.4 Алгоритм Кристофида

В этом разделе мы вернемся к уже известной нам задаче 4 «TSP» и рассмотрим для нее два приближенных алгоритма.

Нам понадобятся некоторые определения.

Определение 2.1.4. Эйлеровым путем в графе называется произвольный путь, проходящий через каждое ребро графа в точности один раз.

Определение 2.1.5. Замкнутый эйлеров путь называется эйлеровым обходом или эйлеровым циклом.

Определение 2.1.6. Эйлеров *граф* — граф, в котором существует эйлеров обход.

Приведем критерий эйлеровости графа.

Теорема 5. Эйлеров обход в графе существует тогда и только тогда, когда граф связный и все его вершины четной степени.

Доказательство. Доказательство достаточности условия теоремы будет следствием анализа алгоритма нахождения эйлерова пути (алгоритм 20).

Необходимость условия очевидна, так как если некоторая вершина v появляется в эйлеровом обходе k раз, то это означает, что степень этой вершины в графе составляет $2k$. \square

Нахождение эйлерова цикла можно выполнить эффективно с помощью алгоритма 20, основная идея которого содержится в построении произвольных замкнутых циклов (если вы окажетесь в эйлеровом графе и будете идти произвольно по его ребрам, сжигая их после своего прохода, то рано или поздно вы вернетесь в точку старта) и объединении таких циклов в единый эйлеров цикл. Пусть дан граф $G = (V, E)$, где V — множество вершин графа, E — множество ребер графа.

Лемма 6. Сложность алгоритма 20 «Цикл Эйлера» есть $O(|E|)$.

Доказательство. Сложность внутреннего цикла $O(|E|)$, т.к. на каждой итерации удаляется по крайней мере одно ребро. \square

Определение 2.1.7. Задача коммивояжера (задача 4 «TSP») называется **метрической**, если для матрицы расстояний выполнено неравенство треугольника:

$$\forall i, j, k \quad d_{ik} \leq d_{ij} + d_{jk}.$$

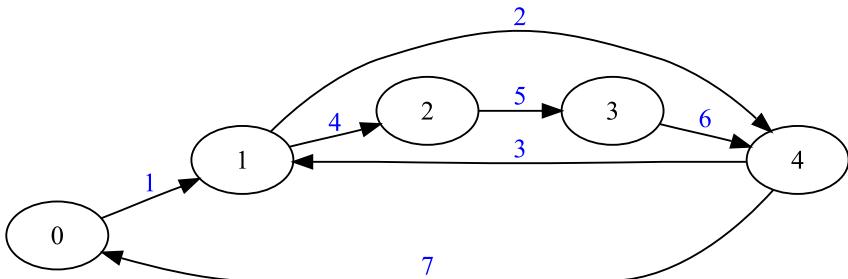
Алгоритм 20 Алгоритм нахождения Эйлерова цикла

```

def EulerCircuit(G):
    if not EulerCircuitExists(G):
        return None

    # Первая попавшаяся — в ЭЦ!
    EP = [ G.nodes()[0] ]
    while G.number_of_edges() > 0:
        for i, v in enumerate(EP):
            # куда бы добавить цикл?
            if G.degree(EP[i]) > 0:
                break
        while G.degree(v) > 0:
            # пока не в «тупике»
            w = G.neighbors(v)[0]
            G.remove_edge(v, w)
            i += 1
            EP.insert(i, w)
            v = w # и повторяем все с w
    return EP

```



Заметим, что здесь рассматривается полный граф.

Необходимо отметить, что метрическая задача коммивояжера \mathcal{NP} -полна. Это легко доказать, заметив, что если веса ребер полного графа принимают только два значения 1 и 2, то задача является метрической. В свою очередь, построение минимального обхода здесь эквивалентно ответу на вопрос: существует ли в графе с вершинами исходного графа и ребрами, имеющими вес 1, гамильтонов цикл?

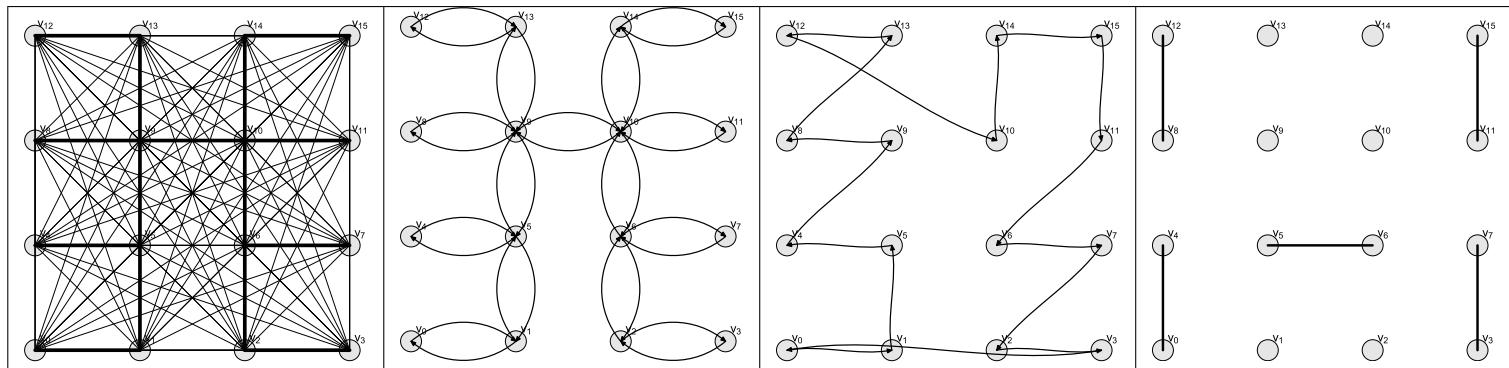
Для этой вариации задачи можно предложить следующий приближенный алгоритм 21 с мультиплексивной точностью 2.

Пример. В рассмотренном в алгоритме 21 «Метрическая TSP» примере эйлеров цикл составляет сле-

[0] + 0 : < 1 2 3 4 0 > -> [0, 1, 2, 3, 4, 0]
[0, 1, 2, 3, 4, 0] + 1 : < 4 1 > -> [0, 1, 4, 1, 2, 3, 4, 0]
[0, 1, 4, 1, 2, 3, 4, 0]

Алгоритм 21 Простой алгоритм для решения метрической задачи коммивояжера

1. Найти минимальное оствое дерево T с матрицей весов $[d_{ij}]$ (ребра из MST выделены жирным).
2. Построить эйлеров граф G и эйлеров обход в G , продублировав все ребра дерева T .
3. Из эйлерова маршрута (обхода) гамильтонов цикл строится путем последовательного вычеркивания вершин, встретившихся ранее.



дующий маршрут:

$$\begin{aligned}
 v_0 &\rightarrow v_1 \rightarrow v_5 \rightarrow v_4 \rightarrow v_5 \rightarrow v_9 \rightarrow v_8 \rightarrow v_9 \rightarrow v_{13} \rightarrow \\
 &\quad \rightarrow v_{12} \rightarrow v_{13} \rightarrow v_9 \rightarrow v_{10} \rightarrow v_{14} \rightarrow v_{15} \rightarrow v_{14} \rightarrow v_{10} \rightarrow \\
 &\quad \rightarrow v_{11} \rightarrow v_{10} \rightarrow v_6 \rightarrow v_7 \rightarrow v_6 \rightarrow v_2 \rightarrow v_3 \rightarrow \\
 &\quad \rightarrow v_2 \rightarrow v_6 \rightarrow v_{10} \rightarrow v_9 \rightarrow v_5 \rightarrow v_1 \rightarrow v_0.
 \end{aligned}$$

Соответствующий ему вложенный тур (гамильтонов цикл) таков:

$$[v_0, v_1, v_5, v_4, v_9, v_8, v_{13}, v_{12}, v_{10}, v_{14}, v_{15}, v_{11}, v_6, v_7, v_2, v_3, v_0].$$

Теорема 6. Алгоритм 21 «Метрическая TSP» находит путь, который длиннее оптимального не более чем в два раза.

Доказательство. Длина кратчайшего гамильтонова пути не меньше размера минимального остовного дерева. Длина эйлерова маршрута в G равна удвоенной длине минимального остовного дерева (по построению). По неравенству треугольника полученный гамильтонов цикл имеет длину, не превосходящую длину эйлерова обхода, т. е. удвоенную длину минимального остовного дерева.

Это и означает, что длина найденного гамильтонова цикла превосходит длину кратчайшего гамильтонова цикла не более чем в два раза. \square

Однако лучшим по точности полиномиальным приближенным алгоритмом для метрической задачи коммивояжера остается алгоритм Кристофидеса (алгоритм 22), предложенный им в 1976 г.

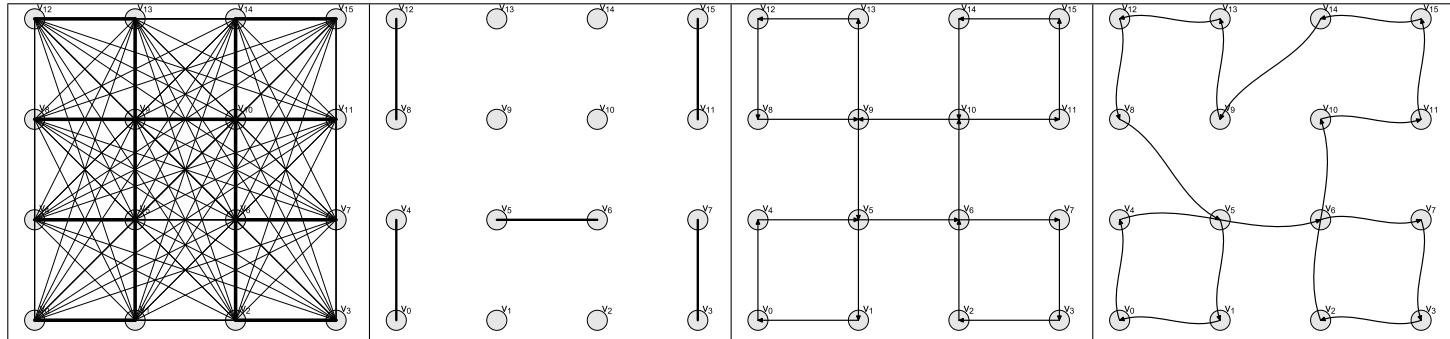
Определение 2.1.8. Совершенное паросочетание⁶ — паросочетание, покрывающее все вершины графа.

Алгоритм 22 Алгоритм Кристофицеса для метрической TSP

Вход: $m \times n$ матрица $[d_{ij}]$ для графа G — постановка задачи 4 «TSP» согласно определению 2.1.7 «Метрическая TSP».

Выход: гамильтонов цикл P_H для графа G .

1. Найти минимальное остовное дерево T (для весов $[d_{ij}]$).
2. Выделить множество $N(T)$ всех вершин нечетной степени в T и найти кратчайшее совершенное паросочетание M в полном графе G с множеством вершин $N(T)$.
3. Построить эйлеров граф G_E с множеством вершин $\{v_1, \dots, v_n\}$ и множеством ребер $T \cup M$.
4. Найти эйлеров обход P_E в G_E .
5. Построить гамильтонов цикл P_H из P_E последовательным вычеркиванием посещенных вершин.



Упражнение 2.1.11. Почему множество всех вершин нечетной степени в минимальном оставном дереве T четно?

Отметим, что алгоритм 22 «Кристофицес-TSP» является полиномиальным алгоритмом. Шаг 1 может быть выполнен за время $O(n^2)$, шаг 2 — за время $O(n^3)$. Шаги 3–5 выполняются за линейное время.

Теорема 7. Алгоритм 22 «Кристофицес-TSP» находит гамильтонов цикл, длина которого превосходит длину оптимального цикла не более чем в $3/2$ раза.

Доказательство. Отметим, что построенный граф G_E действительно эйлеров (согласно теореме 5), т.к. каждая вершина, которая имеет четную степень, имеет ту же степень, и после добавления ребер паросочетания T , а степень каждой вершины нечетной степени увеличивается на единицу (из-за добавления ребра паросочетания). Кроме того, граф связен, т.к. содержит оставное дерево.

Длина результирующего гамильтонова цикла P_H удовлетворяет неравенству

$$c(P_H) \leq c(P_E) = c(G_E) = c(T) + c(M).$$

С другой стороны, для кратчайшего гамильтонова цикла P_H^* выполнено неравенство

$$c(T) \leq c(P_H^*) \leq c(P_H).$$

Пусть i_1, i_2, \dots, i_{2m} — множество вершин нечетной степени в T в том порядке, в каком они появляются в P_H^* . Рассмотрим два паросочетания на этих вершинах:

$$\begin{aligned} M_1 &= \{i_1, i_2\}, \{i_3, i_4\}, \dots, \{i_{2m-1}, i_{2m}\}, \\ M_2 &= \{i_2, i_3\}, \{i_4, i_5\}, \dots, \{i_{2m}, i_1\}. \end{aligned}$$

⁶Perfect matching в англоязычной литературе.

Из неравенства треугольника (докажите это в качестве упражнения)

$$c(P_H^*) \geq c(M_1) + c(M_2).$$

Но M — кратчайшее паросочетание, значит, $c(M_1) \geq c(M)$, $c(M_2) \geq c(M)$. Имеем

$$c(P_H^*) \geq 2c(M) \Rightarrow c(M) \leq \frac{1}{2}c(P_H^*).$$

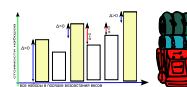
Окончательно имеем

$$c(P_H) \leq c(T) + c(M) \leq c(P_H^*) + \frac{1}{2}c(P_H^*) = \frac{3}{2}c(P_H^*).$$

□

2.2 Аппроксимация с заданной точностью

2.2.1 «Рюкзак»: динамическое программирование



Варианты алгоритмов динамического программирования для задачи о рюкзаке. Алгоритм Немхаузера – Ульмана.

В силу важности метода динамического программирования для построения эффективных приближенных алгоритмов мы остановимся на нем несколько подробнее и проиллюстрируем его на примере следующей \mathcal{NP} -полной задачи.

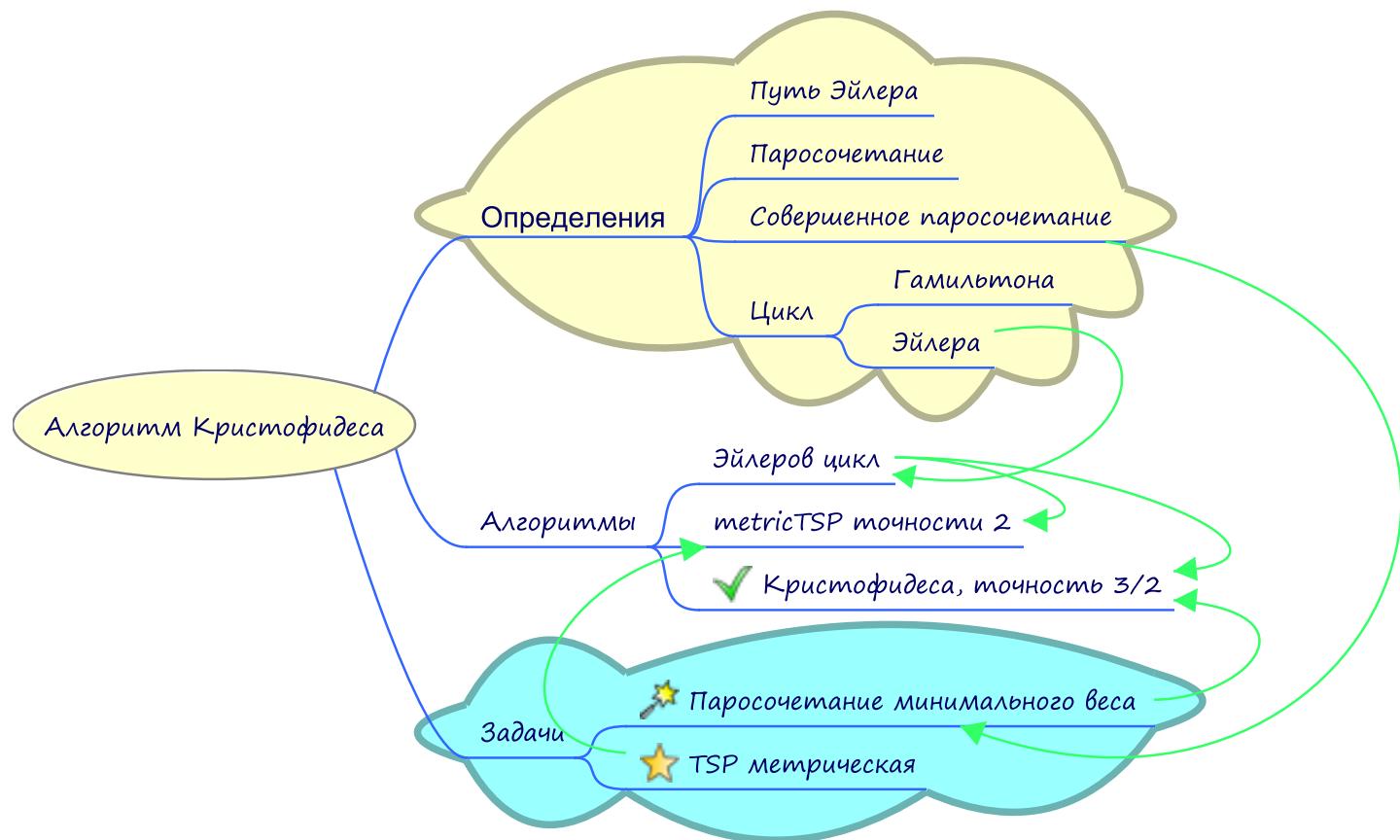


Рис. 2.7: Карта-памятка раздела 2.1.4

Задача 14. «Сумма размеров» («Рюкзак-выполнимость»)

Даны:

$a_1, \dots, a_n, a_j \in N$ — «размеры» или «веса»;

$B \in N$ — «размер рюкзака».

Существует ли решение уравнения:

$$\sum_{i=1}^n a_i x_i = B, \quad x_i \in \{0, 1\}.$$

Обозначим через T множество частичных сумм $\sum_{i=1}^n a_i x_i$ для всех 0/1 векторов (x_1, \dots, x_n) и через T_k — множество всех частичных сумм $\sum_{i=1}^k a_i x_i$, которые не превосходят B (T_0 положим равной нулю).

Для того чтобы решить задачу, достаточно проверить включение $B \in T$. Если оно выполнено, то ответ «Да», в противном случае — «Нет». Множество всех частичных сумм T может быть построено путем просмотра всех булевых (0/1) векторов (т. е. «полным перебором» 2^n векторов). Динамическое программирование дает псевдополиномиальный алгоритм, основанный на рекуррентном соотношении

$$T_{k+1} = T_k \cup \{T_k + a_{k+1}\},$$

где $\{T_k + a_{k+1}\}$ обозначает множество всех чисел вида

$$s + a_{k+1}, \quad s \in T_k,$$

которые не превосходят B .

Отметим, что этот алгоритм на самом деле находит все различные значения частичных сумм и, таким образом, дает точное решение задачи. Число шагов алгоритма есть величина $O(nB)$, поскольку

Алгоритм 23 Сумма размеров

```

def KnapsackSat(A, B):
    T = [0]
    for a in A:
        news = []
        for x in T:
            new = x + a
            if (new <= B and
                new not in T):
                news.append(new)
        T = T + news

    if B in T:
        print "Solution exists"
    else:
        print "No solution"

```

A = [1, 3, 4, 7, 6] B = 10 + 1 --> [0, 1] + 3 --> [0, 1, 3, 4] + 4 --> [0, 1, 3, 4, 5, 7, 8] + 7 --> [0, 1, 3, 4, 5, 7, 8, 10] + 6 --> [0, 1, 3, 4, 5, 7, 8, 10, 6, 9] Solution exists	A = [2, 4, 6, 8] B = 9 + 2 --> [0, 2] + 4 --> [0, 2, 4, 6] + 6 --> [0, 2, 4, 6, 8] + 8 --> [0, 2, 4, 6, 8] No solution
---	--

1. число циклов равно n ;
2. размер каждого множества T_i не превосходит B .

При небольших значениях B снижение сложности с 2^n до Bn дает очевидный эффект, и это характерно для многих псевдополиномиальных алгоритмов.

Так как задача 14 «Сумма размеров» \mathcal{NP} -полна, рассчитывать на построение для нее алгоритма существенно лучшего, чем псевдополиномиальный алгоритм 23 «Сумма размеров», не приходится.

Упражнение 2.2.1. Рассмотрим некоторую модификацию задачи 23 «Сумма размеров», разрешим даже отрицательные размеры. Формально:

- Даны целые числа a_i , $\forall i \in [1 \dots n]$ — $n^2 \leq a_i \leq n^2$ и число B .
- Найти ответ на «Существует ли решение в 0–1 переменных (x_1, \dots, x_n) уравнения $\sum_{i=1}^n a_i x_i = B?$ ».

Существует ли полиномиальный алгоритм для этой задачи?

Рассмотрим теперь внешне похожую \mathcal{NP} -полную задачу 13 «Knapsack» и проиллюстрируем, как метод динамического программирования работает для нее.

Как и прежде, достаточно найти множество T_k всех пар (c, A) , где c — произвольная сумма $\sum_{i=1}^k c_i x_i$ и A — соответствующая сумма $\sum_{i=1}^k a_i x_i \leq B$.

Описанный выше способ позволяет рекурсивно построить множество T_n за Bf^*n шагов (напомним, что через f^* мы обозначаем стоимость оптимального набора). Это аналогично заполнению за n шагов 0/1 таблицы размером $B \times f^*$, представляющей пространство всевозможных допустимых (не обязательно оптимальных) решений задачи 13 «Knapsack».

Однако уменьшить перебор можно учитывая следующий ключевой факт: *Если на k -м шаге у нас имеется несколько частичных решений с одинаковой массой, но различной стоимостью, то можно для каждой массы оставить решения лишь с максимальной стоимостью, а если есть несколько частичных решений с одинаковой стоимостью, но различной массой, то можно смело выкинуть решения с большей массой.* При любом из упомянутых действий мы не потеряем оптимальное решение!

Соответственно, это означает отсутствие необходимости держать в памяти все частичные решения, достаточно на каждой итерации помнить не больше, чем B наиболее «дорогих» частичных решений, либо не больше, чем f^* наиболее «легких» решений.

Алгоритм 24 «Рюкзак-ДинПрог» помнит о наиболее «дорогих» частичных допустимых решениях и тем самым дает точное решение задачи за время $O(nB)$.

Действительно,

1. число циклов равно n ;
2. размер множества отобранных частичных решений не превосходит B .

Упражнение 2.2.2. Постройте алгоритм динамического программирования для задачи 13 «Knapsack», основанный на отборе наиболее «легких» частичных решений. Какова будет его времененная сложность?

Таким образом, для небольших значений параметров c_1, \dots, c_n или B можно построить эффективный псевдополиномиальный алгоритм для точного решения задачи о рюкзаке. Еще раз отметим, что он не является полиномиальным. При росте значений параметров c_1, \dots, c_n, B эффективность этого алгоритма будет уменьшаться. Кроме этого, стоит помнить и о пространственной сложности алгоритма — нам требуется $\Omega(B)$ или $\Omega(f^*)$ памяти для хранения частичных решений.

Упражнение 2.2.3. Придумайте входные наборы для алгоритма 24 «Рюкзак-ДинПрог», на которых он будет работать экспоненциальное время.

Алгоритм 24 «Рюкзак»: динамическое программирование

```

def knapsack_dynprog(items, B):
    # Вес → самый дорогой набор
    T = {0: ItemSet()}
    for item in items: # По всем предметам
        news = []
        for sol in T.values(): # по всем частичным
            testme = sol + item # новый кандидат
            if testme.weight <= B and (
                testme.weight not in T
                or testme.cost > T[testme.weight].cost):
                news.append(testme) # подходит!
        for sol in news: # регистрируем новых
            T[sol.weight] = sol
    result = ItemSet()
    for sol in T.values():
        if sol.cost > result.cost: # ищем самое
            # дорогое решение
            result = sol
    return result, len(T)

```

Предметы ($\frac{\text{стоимость}}{\text{вес}}$): $[\frac{6}{3}, \frac{3}{4}, \frac{2}{5}, \frac{5}{6}, \frac{5}{7}, \frac{1}{8}]$, $B = 9$

Sols	item	news
0: $\frac{0}{0}$	$\frac{6}{3}$	$[\frac{6}{3}]$
0: $\frac{0}{0}$, 3: $\frac{6}{3}$	$\frac{3}{4}$	$[\frac{3}{4}, \frac{9}{7}]$
0: $\frac{0}{0}$, 3: $\frac{6}{3}$, 4: $\frac{3}{4}$, 7: $\frac{9}{7}$	$\frac{2}{5}$	$[\frac{2}{5}, \frac{8}{8}, \frac{5}{9}]$
0: $\frac{0}{0}$, 3: $\frac{6}{3}$, 4: $\frac{3}{4}$, 5: $\frac{2}{5}$, 7: $\frac{9}{7}$, 8: $\frac{8}{8}$, 9: $\frac{5}{9}$	$\frac{5}{6}$	$[\frac{5}{6}, \frac{11}{9}]$
0: $\frac{0}{0}$, 3: $\frac{6}{3}$, 4: $\frac{3}{4}$, 5: $\frac{2}{5}$, 6: $\frac{5}{6}$, 7: $\frac{9}{7}$, 8: $\frac{8}{8}$, 9: $\frac{11}{9}$	$\frac{5}{7}$	[]
0: $\frac{0}{0}$, 3: $\frac{6}{3}$, 4: $\frac{3}{4}$, 5: $\frac{2}{5}$, 6: $\frac{5}{6}$, 7: $\frac{9}{7}$, 8: $\frac{8}{8}$, 9: $\frac{11}{9}$	$\frac{1}{8}$	[]

Оптимальное решение: $\frac{11}{9}$

Упражнение 2.2.4. Придумайте входные наборы для алгоритма из упражнения 2.2.2, на которых он будет работать экспоненциальное время.

Что же делать, если параметры c_1, \dots, c_n или B задачи велики настолько, что сложность алгоритма 24 «Рюкзак-ДинПрог» (или алгоритма из упражнения 2.2.2) не укладывается в требования технического задания по времени и/или по памяти?

Можно организовать «разумный» перебор, не привязываясь к ограничению входных параметров типа B или $\max_i c_i$, перечисляя так называемые *доминирующие* частичные решения.

Определение 2.2.1. Пусть S_1 и S_2 допустимые подмножества предметов для задачи 13 «Knapsack». S_1 **доминирует** над S_2 , если:

- стоимость S_1 больше стоимости S_2 ,
- вес S_1 не больше веса S_2 .

или если

- стоимость S_1 равна S_2 ,
- вес S_1 меньше S_2 .

Набор доминирующих подмножеств есть набор *Парето-оптимальных* решений, т. е. таких решений, в которых нельзя улучшить один параметр (стоимость) без ухудшения другого параметра (увеличения веса).

Алгоритм Немхаузера – Ульмана (алгоритм 25 «Рюкзак Немхаузера–Ульмана»), аналогично алгоритму 24 «Рюкзак-ДинПрог», порождает подмножества поочередным добавлением предметов. Только вместо таблицы (размера не более B) «наиболее дорогих решений» поддерживается список доминирующих

Алгоритм 25 «Рюкзак» Немхаузера – Ульмана

```
def KnapsackNemhauserUllman(items, B):
    pareto = deque([ItemSet()])
    # Парето-оптимальные по весу
    for item in items:
        news = deque()
        for solution in pareto:
            if solution.weight + item.weight <= B:
                news.append(solution + item)
        oldpareto = copy.deepcopy(pareto) # hide
        oldnews = copy.deepcopy(news) # hide
        pareto = merge_item_sets(pareto, news)
    return pareto[-1], len(pareto)
```

Предметы ($\frac{\text{стоимость}}{\text{вес}}$): $[\frac{6}{3}, \frac{3}{4}, \frac{2}{5}, \frac{3}{3}, \frac{6}{8}]$, $B = 10$

$pareto_{old}$	news	$pareto_{new}$
$[\frac{0}{0}]$	$[\frac{6}{3}]$	$[\frac{0}{0}, \frac{6}{3}]$
$[\frac{0}{0}, \frac{6}{3}]$	$[\frac{3}{4}, \frac{9}{7}]$	$[\frac{0}{0}, \frac{6}{3}, \frac{9}{7}]$
$[\frac{0}{0}, \frac{6}{3}, \frac{9}{7}]$	$[\frac{2}{5}, \frac{8}{8}]$	$[\frac{0}{0}, \frac{6}{3}, \frac{9}{7}]$
$[\frac{0}{0}, \frac{6}{3}, \frac{9}{7}]$	$[\frac{3}{3}, \frac{9}{6}, \frac{12}{10}]$	$[\frac{0}{0}, \frac{6}{3}, \frac{9}{6}, \frac{12}{10}]$
$[\frac{0}{0}, \frac{6}{3}, \frac{9}{6}, \frac{12}{10}]$	$[\frac{6}{8}]$	$[\frac{0}{0}, \frac{6}{3}, \frac{9}{6}, \frac{12}{10}]$

Оптимальное решение: $\frac{12}{10}$

решений, упорядоченных по возрастанию веса. При добавлении каждого предмета порождается новый список, в котором подмножества также упорядочены по весу и являются доминирующими по отношению друг к другу. Остается слить эти два списка, удалив доминируемые подмножества.

Заметим, что и алгоритм 25 «Рюкзак Немхаузера–Ульмана» остается в наихудшем случае экспоненциальным.

Упражнение 2.2.5. Придумайте входные наборы для алгоритма 25 «Рюкзак Немхаузера–Ульмана», на которых он будет работать экспоненциальное время.

Однако практика использования показала, что на реальных данных алгоритм 25 «Рюкзак Немхаузера–Ульмана» работает достаточно хорошо. Объяснение этому будет дано в разделе 3.5.

Если же необходимо застраховаться от возможных «плохих» входных данных, то можно использовать идею алгоритма 24 «Рюкзак-ДинПрог» для эффективного приближенного решения задачи 13 «Knapsack» с любой наперед заданной точностью (см. раздел. 2.2.2).

Упражнение 2.2.6. Некоторая торговая сеть решила ускорить обслуживание на кассах, по возможности исключив возню кассиров с копейками. К сожалению, просто «прощать» покупателям «копеечную» часть суммы покупки нельзя — для контролирующих органов необходимо, чтобы зарегистрированные на кассе суммы полностью совпадали с полученными деньгами.

К кассе покупатель подходит с корзиной товаров, каждая строка корзины — тип товара, его цена и количество. На каждый товар, можно давать скидку, но такую, чтобы его цена не стала ниже его цены в рублях (т. е. если товар стоит 12 рублей и 34 копейки, нельзя сбросить цену ниже 12 рублей).

Задача — найти алгоритм, который уменьшит цены всех товаров (при вышеуказанном ограничении), чтобы сумма всей корзины стала целым числом рублей с одной стороны, а с другой, чтобы потери для торговой сети были минимальными.

Очевидно, что существует тривиальное допустимое решение — округлить цены всех товаров, но требуется найти именно решение с минимальными потерями.

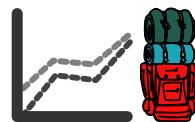
Алгоритм должен работать быстро, за линейное от количества товаров время — это должен быть мгновенный расчет на кассе.

Упражнение 2.2.7. DVS⁷ технология позволяет снижать напряжение на процессоре, и добиваться экономии электроэнергии за счет увеличения времени выполнения задачи.

Пусть процессор поддерживает два уровня напряжения — $U_L < U_H$. Есть набор из n задач, каждая из которых имеет энергоемкость и время выполнения для обоих режимов — т. е. $\forall i$, энергоемкости c_i^H и c_i^L и длительности t_i^H и t_i^L .

Нужно выполнить все задачи на одном процессоре за время не более T , при этом добиться минимального энергопотребления.

2.2.2 Полностью полиномиальная приближенная схема для «Рюкзака»



ε -оптимальные алгоритмы для задачи о рюкзаке с выбираваемой точностью и временем выполнения, полиномиальным по n и $\frac{1}{\varepsilon}$ (\mathcal{FPTAS} — Fully Polynomial Time Approximation Scheme).

Одним из общих подходов к решению переборных задач является разработка приближенных алгоритмов с гарантированными оценками качества получаемого решения⁸(см. определение 2.1.1 « C -приближенный алгоритм»).

⁷Dynamic Voltage Scaling

⁸Напомним, что алгоритм, не гарантирующий точность решения, однако применяемый на практике из-за хороших практических результатов, принято называть эвристикой (см. определение 1.1.1).

Особую роль среди приближенных алгоритмов играют те, которые способны находить решения с любой, заданной как параметр, точностью.

Определение 2.2.2. Алгоритм с мультипликативной ошибкой не более $(1 + \varepsilon)$, где $\varepsilon > 0$, называется ε -оптимальным.

Тот же термин ε -оптимальное используется для обозначения допустимого решения со значением целевой функции, отличающимся от оптимума не более чем в $(1 + \varepsilon)$ раз (таким образом, задача, стоящая перед ε -оптимальным алгоритмом, состоит в отыскании какого-либо ε -оптимального решения).

Определение 2.2.3. Полностью полиномиальной аппроксимационной схемой (FPTAS) называется приближенный алгоритм, в котором уровень точности ε выступает в качестве нового параметра, и алгоритм находит ε -оптимальное решение за время, ограниченное полиномом от длины входа и величины ε^{-1} .

Только одно обстоятельство является препятствием для построения полностью полиномиальной аппроксимационной схемы для задачи 13 «Knapsack» методом динамического программирования (см. раздел 2.2.1). Это наличие «больших» коэффициентов в целевой функции. Действительно, как мы видели в разделе 2.2.1, динамическое программирование дает точный псевдополиномиальный алгоритм для задачи о рюкзаке со сложностью $O(nB)$ или $O(nf^*)$. Если f^* не ограничена сверху никаким полиномом (то есть имеются большие коэффициенты стоимостей), то этот псевдополиномиальный алгоритм 26 не является полиномиальным.

Но к счастью, существует общий метод (который условно можно назвать масштабированием), позволяющий перейти к задаче с небольшими коэффициентами в целевой функции, оптимум которой не сильно отличается от оптимума исходной задачи. Зададимся вопросом: что произойдет, если мы округлим стоимости c_1, \dots, c_n , взяв целые части от деления их на некоторый параметр $scale \in \mathbb{Q}$ и затем домножив снова на $scale$ ($\tilde{c}_i = \lfloor c_i / scale \rfloor \cdot scale$)?

Алгоритм 26 «Рюкзак» с отбором «легких» решений

```
def knapsack_dynprog_lightest(items, B):
    T = {0: ItemSet()} # Цена → самый легкий набор

    for item in items: # По всем предметам
        newT = []
        for sol in T.values(): # по всем частичным
            test = sol + item # тестовый набор
            if test.weight <= B and (
                test.cost not in T
                or test.weight < T[test.cost].weight):
                newT.append(test) # подходит!

        for sol in newT: # регистрируем
            T[sol.cost] = sol # новые решения

    return T[max(T.keys())] # самое дорогое
```

Предметы ($\frac{\text{стоимость}}{\text{вес}}$): $[\frac{6}{3}, \frac{3}{4}, \frac{2}{5}, \frac{5}{6}, \frac{5}{7}, \frac{1}{8}]$, $B = 9$

T	item	newT
0: $\frac{0}{0}$	$\frac{6}{3}$	$[\frac{6}{3}]$
0: $\frac{0}{0}$, 6: $\frac{6}{3}$	$\frac{3}{4}$	$[\frac{3}{4}, \frac{9}{7}]$
0: $\frac{0}{0}$, 9: $\frac{9}{7}$, 3: $\frac{3}{4}$, 6: $\frac{6}{3}$	$\frac{2}{5}$	$[\frac{2}{5}, \frac{5}{9}, \frac{8}{8}]$
0: $\frac{0}{0}$, 2: $\frac{2}{5}$, 3: $\frac{3}{4}$, 5: $\frac{5}{9}$, 6: $\frac{6}{3}$, 8: $\frac{8}{8}$, 9: $\frac{9}{7}$	$\frac{5}{6}$	$[\frac{5}{6}, \frac{11}{9}]$
0: $\frac{0}{0}$, 2: $\frac{2}{5}$, 3: $\frac{3}{4}$, 5: $\frac{5}{6}$, 6: $\frac{6}{3}$, 8: $\frac{8}{8}$, 9: $\frac{9}{7}$, 11: $\frac{11}{9}$	$\frac{5}{7}$	[]
0: $\frac{0}{0}$, 2: $\frac{2}{5}$, 3: $\frac{3}{4}$, 5: $\frac{5}{6}$, 6: $\frac{6}{3}$, 8: $\frac{8}{8}$, 9: $\frac{9}{7}$, 11: $\frac{11}{9}$	$\frac{1}{8}$	$[\frac{1}{8}]$

Оптимальное решение: $\frac{11}{9}$

- Задачу можно решать, «отмасштабировав» все стоимости \tilde{c}_i на величину $scale$ (т. е. поделив, причем без потерь, т.к. все \tilde{c}_i делятся на $scale$ нацело), и это не изменит оптимального набора.
- Время работы алгоритма динамического программирования с отбором наиболее «легких» подмножеств (алгоритм 26) будет ограничено $O(\frac{nf^*}{scale})$.
- Веса a_i мы не меняли, значит, любое допустимое решение «округленной» задачи является также допустимым решением исходной задачи.

- Из-за потерь «округления» оптимум получившейся задачи может стать меньше исходной, т.к. предметы стали стоить несколько «дешевле».

Осталось понять, как связан выигрыш во времени работы алгоритма с потерей точности решения, «стоит ли игра свеч».

Итак, формально, пусть для «округленной» задачи:

\tilde{c}_i — стоимости, $\tilde{c}_i = \lfloor c_i / scale \rfloor \cdot scale$;

\tilde{x}_i — показывает включение предмета в оптимальный набор «округленной» задачи, $\tilde{x}_i \in \{0, 1\}$;

x_i^* — показывает включение предмета в оптимальный набор исходной задачи, $x_i^* \in \{0, 1\}$;

\tilde{f} — оптимум «округленной» задачи, $\tilde{f} = \sum_{i=1}^n \tilde{c}_i \tilde{x}_i$.

Посмотрим, насколько может быть хуже «оптимум» округленной задачи \tilde{f} по сравнению с оптимумом исходной задачи f^* .

Максимальная абсолютная погрешность из-за «округления» только одного j -го предмета, входящего в оптимальный набор для исходной задачи, строго меньше $scale$.

Имеем

$$\tilde{f} = \sum_{i=1}^n \tilde{c}_i \tilde{x}_i \geq \sum_{i=1}^n \tilde{c}_i x_i^* \geq \sum_{i=1}^n (c_i - scale) x_i^* \geq f^* - n \cdot scale.$$

Заметим, для стоимости «рюкзака» при подстановке \tilde{x}_i в исходную задачу выполняется: $\sum_{i=1}^n c_i \tilde{x}_i \geq \sum_{i=1}^n \tilde{c}_i \tilde{x}_i = \tilde{f}$, поэтому дальше в этом разделе мы будем использовать \tilde{f} как нижнюю оценку аппроксимации исходной задачи.

т. е. получаем неравенство для абсолютной погрешности:

$$f^* - \tilde{f} \leq n \cdot scale.$$

Если потребовать, чтобы абсолютная погрешность не превосходила $\frac{\varepsilon}{1+\varepsilon} f^*$, то аппроксимация будет ε -оптимальным решением:

$$\tilde{f} \geq f^* - \frac{\varepsilon}{1 + \varepsilon} f^* = \frac{f^*}{(1 + \varepsilon)}.$$

Чтобы максимально ограничить время работы аппроксимирующего алгоритма $O(\frac{nf^*}{scale})$, мы должны максимизировать параметр $scale$. При этом для сохранения ε -оптимальности надо соблюдать ограничение $scale \leq \frac{\varepsilon f^*}{n(1+\varepsilon)}$. Однако проблема состоит в том, что в момент масштабирования величина оптимума f^* неизвестна, и непонятно, как выбрать оптимальный коэффициент $scale$.

Но можно усилить ограничение на $scale$, рассматривая вместо f^* нижнюю оценку оптимума f_{lb} :

$$scale = \max \left\{ 1, \frac{\varepsilon f_{lb}}{n(1 + \varepsilon)} \right\}. \quad (2.2)$$

Тогда все вышеизложенные соображения о точности «отмасштабированного» решения сохранят силу. Таким образом, стоит задача выбора нижней оценки f_{lb} , которую, с одной стороны, можно найти быстро, а с другой — желательно, чтобы она была как можно ближе к f^* , т.к. это даст возможность увеличить коэффициент $scale$, и тем самым сильнее уменьшить коэффициенты $\tilde{c}_1, \dots, \tilde{c}_n$ и время выполнения алгоритма.

Общая схема представлена в алгоритме 27, где функции «KnapsackFPTAS» на вход, кроме обычных параметров рюкзака и точности аппроксимации ε , передают функцию, используемую для получения нижней оценки стоимости решения.

Осталось найти такие функции. Например, можно рассмотреть тривиальную аппроксимацию «*MaxItemCost*»:

$$f_{lb} \equiv c_{\max} = \max_i c_i.$$

Получим функцию «*KnapsackFPTAS_{MaxItemCost}*» в алгоритме 27.

Сложность «*KnapsackFPTAS_{MaxItemCost}*» будет:

$$\begin{aligned} O\left(\frac{n\tilde{f}}{\text{scale}}\right) &= O\left(\frac{n \cdot nc_{\max}}{\text{scale}}\right) = O\left(\frac{n \cdot nc_{\max}}{\frac{c_{\max}\varepsilon}{n(1+\varepsilon)}}\right) = \\ &= O\left(\frac{n^3(1+\varepsilon)}{\varepsilon}\right) = O\left(\frac{n^3}{\varepsilon}\right). \end{aligned}$$

Можно ли улучшить эту оценку? Ответ на этот вопрос положителен. Для этого рассмотрим менее наивную аппроксимацию величины f^* . Вспомним алгоритм 19 «Рюкзак-Жадный» из раздела 2.1.3.

Для значения решения f_G , полученного модифицированным жадным алгоритмом для задачи о рюкзаке, и оптимального значения f^* выполняется $\frac{f^*}{2} \leq f_G \leq f^*$.

Таким образом, мы получаем более точную нижнюю оценку f_{lb} для f^* , которая, как правило, больше тривиальной оценки c_{\max} . Посмотрим, поможет ли это нам улучшить верхнюю оценку времени выполнения алгоритма.

Теорема 8. Алгоритм «*KnapsackFPTAS_{KnapsackGreedy}*» имеет сложность $O\left(\frac{n^2}{\varepsilon}\right)$.

Доказательство. Используя неравенство $\tilde{f} \leq f^* \leq 2f_G$ и (2.2), получаем оценку сложности алгоритма

$$O\left(\frac{n\tilde{f}}{\text{scale}}\right) = O\left(\frac{n \cdot \tilde{f}}{\frac{\varepsilon \cdot f_G}{n(1+\varepsilon)}}\right) = O\left(\frac{2n^2(1+\varepsilon)}{\varepsilon}\right) = O\left(\frac{n^2}{\varepsilon}\right).$$

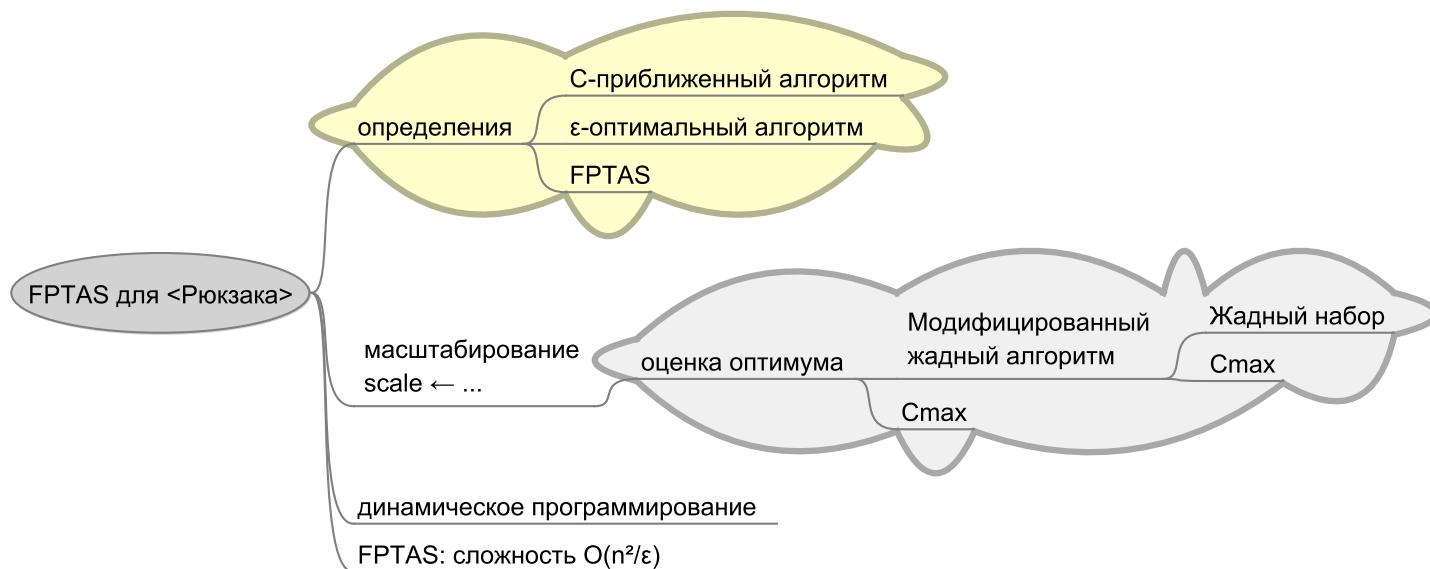


Рис. 2.8: Карта-памятка раздела 2.2.2

Алгоритм 27 PTAS для рюкзака

```
def knapsack_fptas(items, B, epsilon, lower_bound):
    # Вычисляем нижнюю оценку стоимости
    F_lb = lower_bound(items, B)
    # параметр округления $scale$
    scale = epsilon * F_lb / len(items) / (1 + epsilon)
    # Набор с округленными стоимостями
    Ds = [Item(item.cost/scale, item.weight) for item in items]
    knapset, indices = knapsack_dynprog_lightest(Ds, B)
    ApproxCost = sum(items[i].cost for i in indices)
```

Глава 3

Вероятностный анализ детерминированных алгоритмов

3.1 Сложность и полиномиальность в среднем

Среди подходов к решению \mathcal{NP} -трудных задач можно выделить два основных. Первый заключается в построении приближенных алгоритмов с гарантированными оценками точности получаемого решения (этот подход уже был рассмотрен нами в главе 2), а второй — в отказе от анализа сложности алгоритмов по наихудшему случаю и переходе к анализу сложности в среднем. Настоящая глава посвящена изучению второго подхода (анализу сложности в среднем).

Известно, что большинство задач дискретной оптимизации является \mathcal{NP} -трудным, и для них существование полиномиальных алгоритмов маловероятно. Несмотря на это, для многих исходных данных

(входов) такие задачи бывают легко разрешимы на практике. Дело в том, что вся трудность задачи может заключаться в небольшом подмножестве входов. Проблема нахождения таких входов важна для экспериментальной оценки эффективности алгоритмов и в то же время играет заметную роль в математической криптографии. Концепция сложности в среднем для таких задач представляется более адекватной, чем концепция сложности в худшем случае.

Имеется еще одно соображение в пользу перехода от анализа по худшему случаю к анализу сложности в среднем. Оно состоит в том, что некоторые практически эффективные алгоритмы не являются эффективными при анализе по худшему случаю. Пожалуй, наиболее известным примером такого алгоритма является симплекс-метод решения задач линейного программирования, который, не являясь полиномиальным алгоритмом, поразительно хорошо зарекомендовал себя при решении практических задач.

Таким образом, естественные попытки ослабить требования в определении эффективности привели к понятию сложности в среднем (*average case complexity*), под которым, грубо говоря, понимается математическое ожидание времени работы алгоритма при заданном вероятностном распределении на исходных данных. С середины 80-х годов начала развиваться теория сложности в среднем, в некотором смысле аналогичная теории полиномиальной сводимости для \mathcal{NP} -трудных задач. Кроме того, для ряда \mathcal{NP} -полных задач удалось построить полиномиальные в среднем алгоритмы. Однако достигнутые здесь успехи оказались гораздо скромнее результатов в теории сложности при анализе по худшему случаю.

В последнее время получен ряд интересных результатов для задач типа рюкзака с константным числом ограничений, введено новое понятие «сглаженной сложности» (*smoothed complexity*) и доказано, что некоторые варианты симплекс-метода имеют полиномиальную сглаженную сложность. Построены приближенные полиномиальные в среднем алгоритмы нахождения максимальной клики в графе с гарантируемыми оценками точности, существенно лучшими аналогичных оценок для худшего случая.

Перейдем к конкретизации сказанного выше. Пусть для данного алгоритма A время его работы на входе I обозначается через $T_A(I)$. Пусть также для каждого натурального n задано распределение вероятно-

стей на входах I длины n , обозначаемое $P_n(I)$. Поскольку T является случайной величиной, естественным кажется определить среднее время работы алгоритма A как математическое ожидание времени работы алгоритма $E_n T_A$ на входах длины n .

Определение 3.1.1. «Полиномиальный в среднем (точно)»

Алгоритм A называется **полиномиальным в среднем**, если среднее время его работы ограничено полиномом от длины входа, т. е.

$$\exists C > 0 : E_n T_A = O(n^C).$$

Однако в теоретических работах, касающихся теории сводимости и сложности в среднем, полиномиальность в среднем принято определять по-другому. Причина состоит в том, что приведенное выше определение слишком чувствительно: изменение T на T^2 (типичная оценка при симуляции одной вычислительной модели на другой) может привести к тому, что алгоритм, полиномиальный в среднем, скажем, на модели RAM, не является полиномиальным в среднем на машине Тьюринга.

Упражнение 3.1.1. Приведите пример функции T_A (времени работы некоторого алгоритма A) и распределения исходных данных $P_n(I)$, для которых T_A является полиномиальной в среднем ($E_n T_A = O(n^c)$), а T_A^2 — нет.

Эту ситуацию нельзя признать удовлетворительной, поскольку ничего подобного не происходит при анализе сложности по худшему случаю: определение полиномиального алгоритма не зависит от выбранной модели вычислений.

Поэтому в теоретических работах, касающихся теории сводимости и сложности в среднем, используют обычно другое определение.

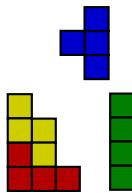
Определение 3.1.2. «Полиномиальный в среднем»

Алгоритм называется **полиномиальным в среднем**, если для времени работы алгоритма T выполняется:

$$\exists \varepsilon > 0 : \mathbf{E}_n T^\varepsilon = O(n).$$

Тем не менее, в исследовании сложности в среднем конкретных алгоритмов обычно используется определение 3.1.1 «Полиномиальный в среднем (точно)» с указанием конкретной вычислительной модели, для которой проводится анализ (обычно это машины с произвольным доступом к памяти — RAM). Мы в следующих разделах также будем следовать этому определению.

3.2 Задача упаковки



Алгоритм динамического программирования для задачи упаковки подмножеств. Полиномиальность алгоритма «в среднем».

Задача 15. «Упаковка» (Packing)

Дано конечное множество L из t элементов и система его подмножеств S_1, \dots, S_n . Требуется найти максимальную по числу подмножеств подсистему попарно непересекающихся подмножеств.

При реализации и анализе алгоритмов гораздо удобней формализовать представление отношений между элементами и подмножествами в виде матрицы инцидентности.

Определение 3.2.1. Пусть

$L = \{l_1, \dots, l_m\}$ — m -элементное множество;

$\{S_1, \dots, S_n\}$ — семейство подмножеств L ;

Тогда

- Элемент l_i и подмножество S_j **инцидентны**, если $l_i \in S_j$.
- **Матрицей инцидентности** называется $\{0,1\}$ -матрица $A = (a_{ij})$ размером $m \times n$, для которой:
 $a_{ij} = 1 \Leftrightarrow$ элемент l_i и подмножество S_j инцидентны.

Таким образом, задача 15 «Packing» будет эквивалентна следующей 0–1 целочисленной линейной программе:

$$\begin{aligned} \sum_{j=1}^n x_j &\rightarrow \max, \\ x_j &\in \{0, 1\} \quad \forall j : 1 \leq j \leq n, \\ \sum_{j=1}^n a_{ij} x_j &\leq 1 \quad \forall i : 1 \leq i \leq m. \end{aligned} \tag{3.1}$$

В этой целочисленной линейной программе n столбцов и переменных x_1, \dots, x_n соответствуют подмножествам S_1, \dots, S_n и их включению в решение, а 0–1-матрица инцидентности $A = (a_{ij})$ размера

$m \times n$ описывает состав каждого подмножества (см. определение 3.2.1 «Инцидентность»). Таким образом, m строк-ограничений, соответствующих m предметам, очевидным образом выражают ограничение на попарную непересекаемость выбранных подмножеств ни по одному предмету, а целевая функция равна числу таких подмножеств.

Заметим, что в литературе «Упаковкой подмножеств» называется иногда и более общая постановка ЦЛП, где c_1, \dots, c_n и b_1, \dots, b_m — произвольные положительные целые числа:

$$\begin{aligned} & \sum_{j=1}^n c_j x_j \rightarrow \max, \\ & \forall j : 1 \leq j \leq n \quad x_j \in \{0, 1\}, \\ & \forall i : 1 \leq i \leq m \quad \sum_{j=1}^n a_{ij} x_j \leq b_i. \end{aligned}$$

Известно, что задача 15 «Packing» \mathcal{NP} -полнна [Joh90].

Тривиальный метод решения задачи 15 «Packing» — полный перебор всех $N = 2^n$ наборов подмножеств с проверкой на совместность и выбором набора, содержащего наибольшее число подмножеств.

Более перспективный подход — ограничить перебор по допустимым решениям задачи, применив метод динамического программирования.

Обозначим через X_j множество всех допустимых булевых векторов для системы с $n - j$ нулевыми последними компонентами и через \mathbf{e}_j — вектор размерности n с единичной j -й компонентой и остальными нулевыми компонентами. Тогда можно строить множество допустимых решений X_j на основе множества X_{j-1} , пытаясь добавить вектор e_j ко всем допустимым решениям из X_{j-1} . Более формально это описано в алгоритме 28.

Алгоритм 28 «Упаковка»: динамическое программирование

```
def PackingDynP(A):
    m, n = A.shape
    X=[( zeros(n, int), #решение
          zeros(m, int), #покрытие
          0   )]           #размер покрытия
    for j in xrange(n):
        Sj = get_column(j)
        for sol, cov, size in X[:]:
            newcov = cov+Sj
            if max(newcov) <= 1:
                newsol = copy(sol)
                newsol[j] = 1
                X.append( (
                    newsol,
                    newcov,
                    size + 1) )
    return get_optimal_set(X)
```

Работа алгоритма показана на рис. 3.2.

```

[[0 1 0 1]
 [1 0 1 0]
 [0 1 1 0]] --> 3x4-matrix A

column(0)+< [0 0 0 0], [0 0 0]>
=> [0 1 0]+[0 0 0]=[0 1 0] => [1 0 0 0]
column(1)+< [0 0 0 0], [0 0 0]>
=> [1 0 1]+[0 0 0]=[1 0 1] => [0 1 0 0]
column(1)+< [1 0 0 0], [0 1 0]>
=> [1 0 1]+[0 1 0]=[1 1 1] => [1 1 0 0]
column(2)+< [0 0 0 0], [0 0 0]>
=> [0 1 1]+[0 0 0]=[0 1 1] => [0 0 1 0]
column(3)+< [0 0 0 0], [0 0 0]>
=> [1 0 0]+[0 0 0]=[1 0 0] => [0 0 0 1]
column(3)+< [1 0 0 0], [0 1 0]>
=> [1 0 0]+[0 1 0]=[1 1 0] => [1 0 0 1]
column(3)+< [0 0 1 0], [0 1 1]>
=> [1 0 0]+[0 1 1]=[1 1 1] => [0 0 1 1]
Выбран набор [1 1 0 0] размером 2

```

Рис. 3.1: Работа алгоритма 28 «Упаковка-ДинПрог»

Нетрудно убедиться, что сложность алгоритма 28 «Упаковка-ДинПрог» составляет $O((m + n)n|X_n|)$. Действительно, внешний цикл — $O(n)$, внутренний — $O(|X_n|)$, проверка на допустимость и добавление нового решения — $O(m + n)$.

Таким образом, сложность алгоритма существенно зависит от размера множества допустимых решений, возникающих при выполнении алгоритма.

Упражнение 3.2.1. Какие входные данные для алгоритма 28 «Упаковка-ДинПрог» заставят его работать

экспоненциально долго? А какие — за $O(n^3)$?

Но можно показать, что алгоритм 28 «Упаковка-ДинПрог» может быть полиномиальным в среднем¹, т. е. работать эффективно на случайных входных данных.

Теорема 9. Пусть a_{ij} в (3.1) являются независимыми случайными величинами, принимающими значения $\{0, 1\}$, причем выполняется:

$$\begin{aligned} \mathbb{P}\{a_{ij} = 1\} &= p, \\ \mathbb{P}\{a_{ij} = 0\} &= 1 - p, \\ mp^2 &\geq \ln n. \end{aligned} \tag{3.2}$$

Тогда алгоритм 28 «Упаковка-ДинПрог» является полиномиальным в среднем.

Доказательство. Математическое ожидание времени работы алгоритма есть $O(nm \mathbf{E}|X_n|)$. Поэтому для доказательства теоремы 9 достаточно будет оценить сверху математическое ожидание размера множества X_n . Пусть $k > 0$,

\bar{x}^k — вектор с k единицами (на позициях $\{j_1, \dots, j_k\}$) и $n - k$ нулями;

p_{ki} — вероятность выполнения i -го неравенства для \bar{x}^k ;

P_k — вероятность того, что \bar{x}^k — допустимое решение.

Упражнение 3.2.2. Чему будут равны \bar{x}^k , p_{ki} и P_k при $k = 0$?

¹Причем в более строгом варианте, в смысле определения 3.1.1 «Полиномиальный в среднем (точно)».

Сначала оценим сверху p_{ki} :

$$\begin{aligned}
 p_{ki} &\equiv \mathbb{P} \left\{ \sum_{j=1}^n a_{ij} x_j^{(k)} \leq 1 \right\} = \mathbb{P} \left\{ \sum_{j \in \{j_1, \dots, j_k\}} a_{ij} \leq 1 \right\} = \\
 &= \mathbb{P} \left\{ \sum_{j \in \{j_1, \dots, j_k\}} a_{ij} = 0 \right\} + \mathbb{P} \left\{ \sum_{j \in \{j_1, \dots, j_k\}} a_{ij} = 1 \right\} = \\
 &= (1-p)^k + kp(1-p)^{k-1} = (1-p)^{k-1}(1+p(k-1)) \leq \\
 &\leq (1-p)^{k-1}(1+p)^{k-1} = (1-p^2)^{k-1} \leq e^{-p^2(k-1)}.
 \end{aligned}$$

Теперь оценим собственно вероятность попадания \bar{x}^k в допустимые решения:

$$\mathbb{P}_k = \prod_{i=1}^m p_{ki} \leq \prod_{i=1}^m e^{-p^2(k-1)} = e^{-mp^2(k-1)}.$$

Упражнение 3.2.3. Докажите, что $\mathbf{E} |X_n| = \sum_{k=0}^n \binom{n}{k} \mathbb{P}_k$.

Следовательно, мы можем оценить математическое ожидание мощности X_n следующим образом:

$$\begin{aligned}
 \mathbf{E} |X_n| &= \sum_{k=0}^n \binom{n}{k} P_k = 1 + \sum_{k=1}^n \binom{n}{k} P_k \leq 1 + n + \sum_{k=2}^n \binom{n}{k} P_k < \\
 &< 1 + n + \sum_{k=2}^n n^k e^{-mp^2(k-1)} \leq 1 + n + \sum_{k=2}^n e^{k \ln n - mp^2(k-1)} = \\
 &= 1 + n + n \sum_{k=2}^n e^{(k-1)(\ln n - mp^2)}.
 \end{aligned}$$

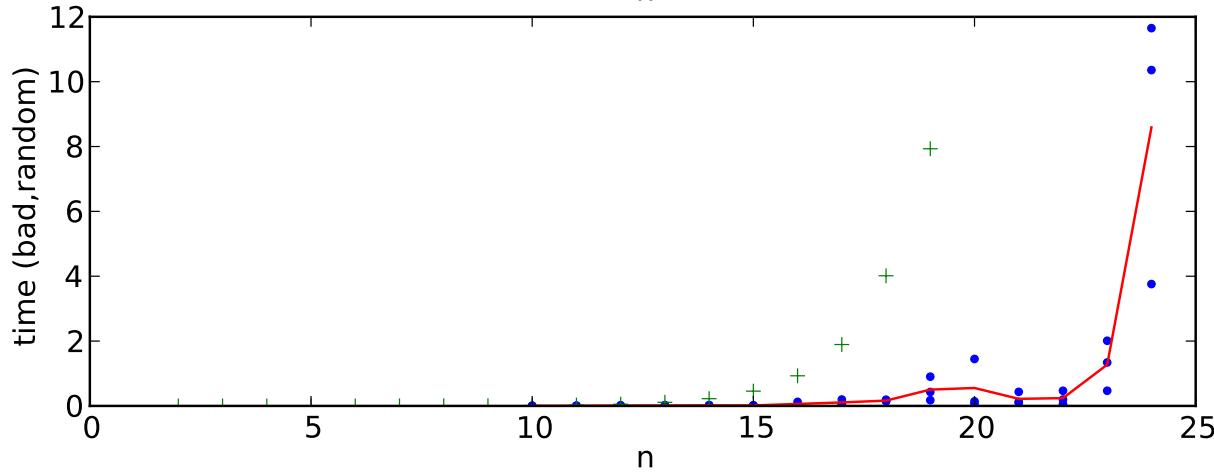
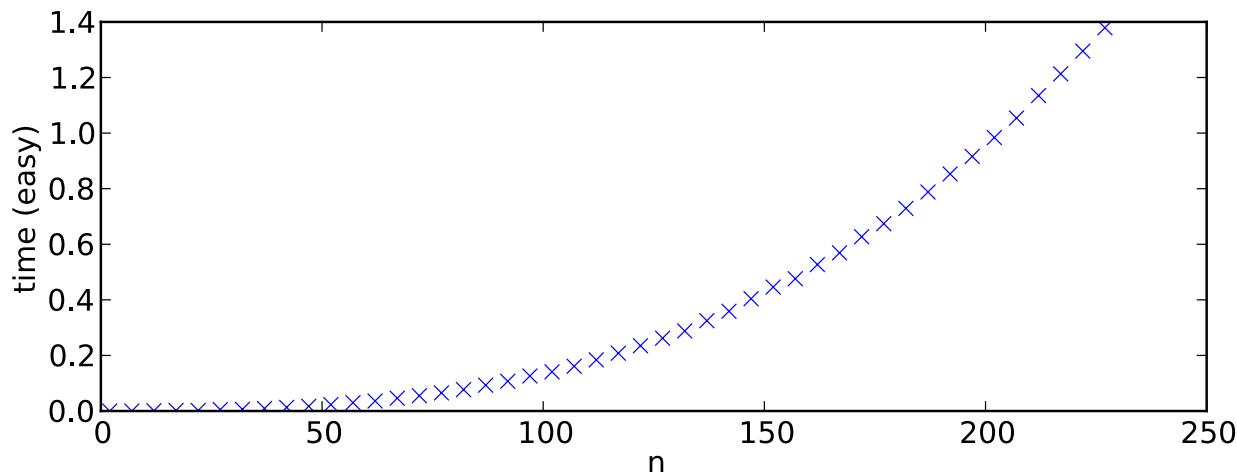
При условии $mp^2 \geq \ln n$ в последней сумме каждый член не превосходит 1. Это и означает, что $\mathbf{E}[X_n] = O(n^2)$. \square

Некоторые экспериментальные результаты поведения алгоритма 28 «Упаковка-ДинПрог» на случайных данных, можно увидеть на рисунке 3.2.

В заключение отметим, что из полученных результатов вытекает не только полиномиальный в среднем алгоритм для задачи 15 «Packing», но и полиномиальный в среднем алгоритм для более трудной задачи подсчета числа целых точек в многограннике, заданном ограничениями (3.1).

3.3 Выполнимость КНФ

Раздел основан на работе [Iwa89].



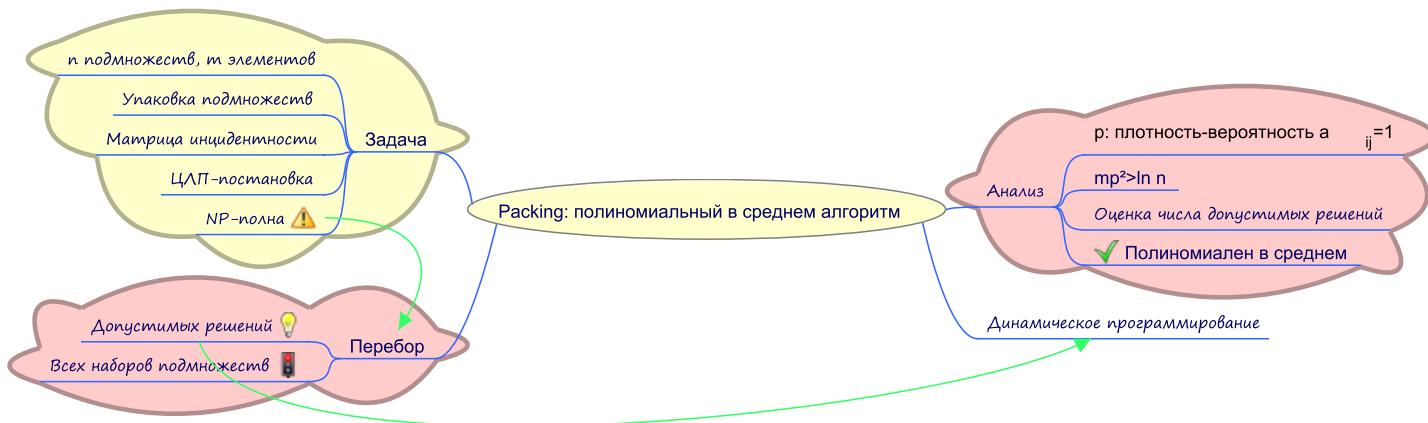


Рис. 3.3: Карта-памятка раздела 3.2.0

Задача 16. «Выполнимость/SAT»². Дано булевское выражение, являющееся **конъюнктивной нормальной формой (КНФ)**:

$$CNF = \bigwedge_{i=1}^m C_i, \quad (3.3)$$

где C_i — элементарные дизъюнкции вида

$$x_{j_1}^{\sigma_1} \vee \dots \vee x_{j_k}^{\sigma_k}, \quad (3.4)$$

²В англоязычной литературе — Satisfiability или просто SAT.

$1 \leq k \leq n$, $\sigma_j \in \{0, 1\}$, $x^1 = x$ и $x^0 = (\neg x)$.

Существует ли (булевский) набор переменных x_j , обращающий эту формулу в 1 (т. е. в «Истину»)?

Задаче 16 «SAT» выпала честь быть первой задачей, для которой была установлена \mathcal{NP} -полнота (см. раздел 6.2). И хотя \mathcal{NP} -полнота задачи, по сути, определяет «переборность» любого алгоритма для ее решения³, одни виды перебора могут быть перспективнее других.

Итак, как можно организовать решение этой задачи?

Самым простым решением будет перебор всех входных наборов $x = \{x_1, \dots, x_n\}$, пока $CNF(x)$ не станет равна единице. Однако в худшем случае, например, если CNF невыполнима, надо перебрать 2^n наборов x .

Для некоторых формул этот перебор можно сократить, если обеспечить подсчет размера множества невыполнимых наборов — $|X_0|$, где $X_0 = \{x : CNF(x) = 0\}$. CNF невыполнима тогда и только тогда, когда $|X_0| = 2^n$.

Остается вопрос: как подсчитать $|X_0|$ более эффективно, нежели перебором x ?

Введем (или напомним) несколько обозначений.

Определение 3.3.1. *Литерал — каждое вхождение переменной x_i (или ее отрицания) в скобку. Например, для $(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_3)$ литералами будут $x_1, \bar{x}_2, \bar{x}_1, x_3$.*

Для некоторого подмножества из k скобок $S = \{C_{j_1}, \dots, C_{j_k}\}$ мы обозначим через Lit_S все литералы S , т. е. все литералы $lit \in \{x_i, \bar{x}_i\}$, входящие в S .

Рассмотрим некоторую КНФ (обозначим ее $CNF(x)$). На каких наборах она равна нулю? Очевидно, если $CNF(x) = 0$, то одна или несколько скобок-дизъюнкций $C_j(x) = 0$.

³Разумеется, при гипотезе $\mathcal{P} \neq \mathcal{NP}$

Возьмем некоторое подмножество $S = \{C_{j_1}, \dots, C_{j_k}\}$ из k скобок и зададимся вопросом: когда все скобки в S будут равны нулю? Обозначим это утверждение предикатом $Z_S(x) = 1$ и раскроем определение:

$$Z_S(x) = \bigwedge_{i=1}^k \overline{C_{j_i}} = \bigwedge_{i=1}^k \overline{\bigvee_{lit \in C_{j_i}} lit} = \bigwedge_{lit \in \text{Lit}_S} \overline{lit}.$$

т. е. Z_S зависит только от множества литералов Lit_S и не зависит от распределения литералов по отдельным скобкам. А для множества S возможны два варианта. Множество S может быть:

зависимое — $\exists i$: литералы $x_i \in \text{Lit}_S$ и $\overline{x_i} \in \text{Lit}_S$. В этом случае $Z_S \equiv 0$, т. е. $|x : Z_S(x) = 1| = 0$ — нет выполняющих Z_S наборов.

независимое — $\nexists i$: $x_i \in \text{Lit}_S$ и $\overline{x_i} \in \text{Lit}_S$. Тогда есть выполняющие Z_S наборы x , причем каждая переменная из Lit_S может входить либо в положительной степени и иметь значение «0», либо в отрицательной степени и иметь значение «1». т. е. если $\overline{x_3} \in \text{Lit}_S$, то $x_3 = 1$, а если $x_7 \in \text{Lit}_S$, то $x_7 = 0$. Для остальных переменных, не входящих в Lit_S , значение может быть любым. Поэтому $|x : Z_S(x) = 1| = 2^{n - |\text{Lit}_S|}$.

Итак, если $\exists x : CNF(x) = 0$, то существует некоторое независимое S . Для подсчета размера множества обнуляющих наборов $|X_0|$ построим все независимые множества и просуммируем для каждого независимого множества S число «обнуляющих» наборов $2^{n - |\text{Lit}_S|}$.

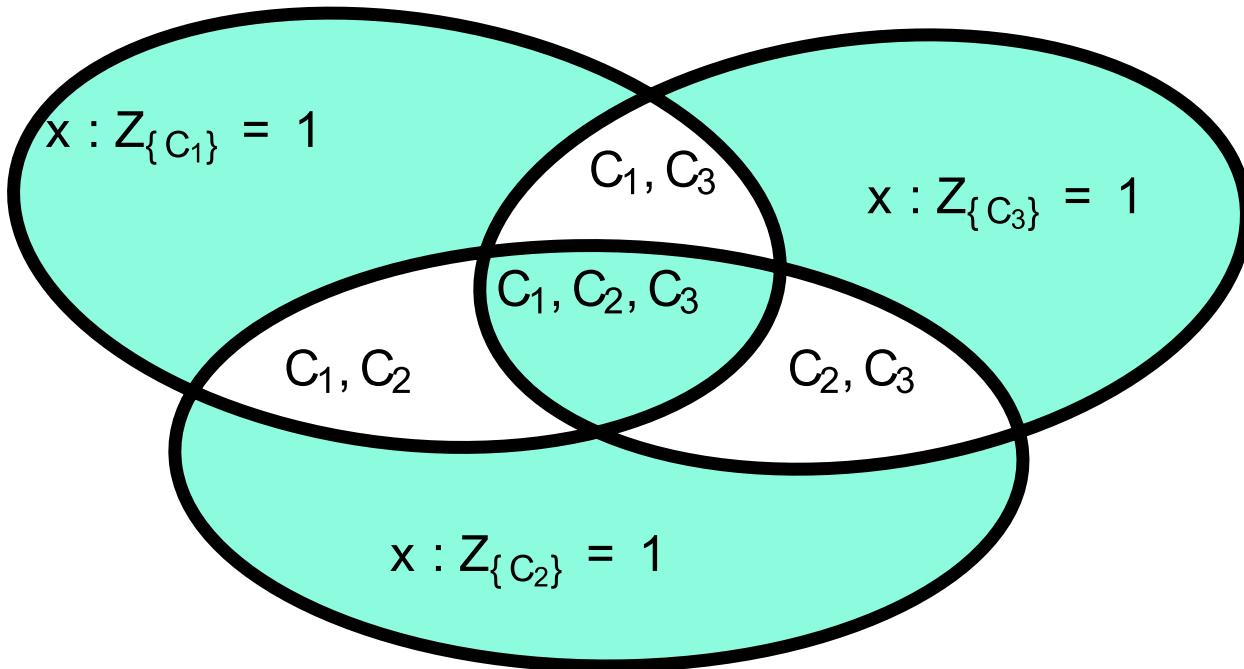


Рис. 3.4: «Обнуляющие» наборы для трехскобочной КНФ

Разумеется, напрямую суммировать нельзя, т.к. «обнуляющие» наборы различных множеств пересекаются. Например, рассмотрим трехскобочную (скобки C_1, C_2, C_3) КНФ, у которой все наборы скобок $\{C_1\}, \{C_2\}, \{C_3\}, \{C_1, C_2\}, \{C_1, C_3\}, \{C_2, C_3\}, \{C_1, C_2, C_3\}$ являются независимыми. Тогда множество «обнуляющих» наборов $\{x : Z_{\{C_1\}} = 1\}$ включает в себя множество $\{x : Z_{\{C_1, C_2\}} = 1\}$, которое включает

в себя, в свою очередь, множество $\{x : Z_{\{C_1, C_2, C_3\}} = 1\}$.

Как раз для подсчета мощности объединения пересекающихся множеств и применяется комбинаторная формула включений-исключений⁴, утверждающая, что для конечных множеств A_1, \dots, A_n мощность их объединения можно подсчитать по формуле

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{i=1}^n |A_i| - \sum_{i,j : i < j} |A_i \cap A_j| + \sum_{i,j,k : i < j < k} |A_i \cap A_j \cap A_k| - \dots (-1)^{n-1} |A_1 \cap \dots \cap A_n|.$$

В нашем случае A_i обозначает множество наборов, для которых i -я скобка C_i равна нулю, поэтому

$$|X_0| = |\bigcup_{i=1}^m A_i|.$$

Далее удобно разбить все независимые множества на « k -слои» N_k : независимые множества с k скобками, и организовать суммирование включений-исключений по этим слоям:

$$|X_0| = \sum_{k=1}^m (-1)^{k-1} \cdot \sum_{S \in N_k} 2^{n-|Lit_S|}.$$

Осталось понять, как построить все независимые множества скобок. Оказалось, их также удобно строить «послойно», получая N_{k+1} из N_k попыткой «добавить» каждую скобку C_j к каждому множеству из N_k следующим образом, и оставляя только независимые множества, т.е. те множества, в которых ни один литерал не встречается со своим отрицанием.

⁴Вариант: «формула включения-исключения», в англоязычной литературе — «inclusion-exclusion principle» или «sieve principle».

Стартуем же мы с пустого множества N_0 .

Заметим, что непосредственно для подсчета $|X_0|$ нет необходимости хранить все независимые множества, достаточно хранить текущее N_k (и вычисляемое N_{k+1}), и суммировать оценку $|X_0|$ по формуле включений-исключений одновременно с построением N_{k+1} .

Таким образом, мы получаем алгоритм 29.

Лемма 7. Сложность алгоритма 29 в наихудшем случае:

$$O(m^2n \cdot \max_k |N_k|).$$

Доказательство. Действительно:

- цикл по «слоям» — $O(m)$;
- цикл по скобкам — $O(m)$;
- цикл по независимым множествам из k скобок — $O(N_k)$.
- цикл по литералам в скобке (проверка независимости множества $S = C_j + t$) внесет множитель $O(n)$;

Общая трудоемкость — $O(m^2n \cdot \max_k |N_k|)$. □

Видно, что, как и в случае с другими алгоритмами динамического программирования, время работы алгоритма определяется размером множества частичных решений (в данном случае N_k). В частности, оно может быть экспоненциальным.

Алгоритм 29 Динамическое программирование для «SAT»

```

def sat_counting(cnf):
    size = 0 #счетчик выполняющих КНФ наборов
    n = cnf.num_vars()
    N = {cnf.empty_clauses_set(): SetOfLiterals([])}
    for k, dummy in enumerate(cnf): # цикл по слоям
        S = {} # новый слой независимых множеств
        for j, C in enumerate(cnf): # цикл по скобкам
            for t in N: # цикл по независимым множествам
                if j not in t: # если скобки там нет
                    s = t + j # добавляем её
                    if s not in S: # в слое S нет такого?
                        lit = copy(N[t]) # «наследуем» t
                        for x in C:
                            ok = -x not in lit
                            if not ok:
                                break # x несовместен
                            lit.add(x)
                        if ok and len(lit) > len(N[t]):
                            S[s] = lit # да, s - независимое
                            size += 2**(n - len(lit)) * (-1)**k
    N = S
    return size

```

$$CNF = (x_2 \vee x_3 \vee \bar{x}_1) \wedge (x_3 \vee \bar{x}_2) \wedge (x_1 \vee \bar{x}_3) \wedge (x_1 \vee x_2 \vee x_4)$$

k	$size$	N
0	0	$\{\}: \{\}$
1	12	$\{C_0\}: \{x_2, x_3, \bar{x}_1\}, \{C_3\}: \{x_1, x_2, x_4\}, \{C_1\}: \{x_3, \bar{x}_2\}, \{C_2\}: \{x_1, \bar{x}_3\}$
2	11	$\{C_2, C_3\}: \{x_1, x_2, x_4, \bar{x}_3\}$
3	11	
end	11	

$$11 < 2^4 \rightarrow .$$

$$CNF = (x_3 \vee \bar{x}_2) \wedge (x_1 \vee \bar{x}_3) \wedge (\bar{x}_3 \vee \bar{x}_1) \wedge (x_2 \vee x_3)$$

k	$size$	N
0	0	$\{\}: \{\}$
1	8	$\{C_0\}: \{x_3, \bar{x}_2\}, \{C_3\}: \{x_2, x_3\}, \{C_1\}: \{x_1, \bar{x}_3\}, \{C_2\}: \{\bar{x}_3, \bar{x}_1\}$
2	8	
3	8	
end	8	

$$8 = 2^3 \rightarrow .$$

Упражнение 3.3.1. Какие входные данные для алгоритма 29 заставят его работать экспоненциально долго?

С другой стороны, на многих входах этот алгоритм работает эффективно.

Упражнение 3.3.2. На каких входных данных этот алгоритм будет работать $O(m)$?

Для оценки практичности алгоритма попробуем ввести некоторое вероятностное распределение входных данных и оценить сложность алгоритма в среднем.

Теорема 10. Пусть для каждой скобки вероятность появления каждой из n переменных (или ее отрицания) равна p , причем

$$np^2 \geq \ln m.$$

Тогда алгоритм 29 является полиномиальным в среднем.

Доказательство. Для доказательства достаточно будет оценить сверху математическое ожидание $|N_k|$.

Пусть S_k — некоторое множество скобок, $|S_k| = k$.

Вероятность того, что внутри одной скобки x_i «не встретится» с $\neg x_i$ равна $1 - p^2$. Вероятность $\tilde{p}_i(S_k)$, что x_i избежит встречи с $\neg x_i$ в каждой из k скобок S_k равна

$$\tilde{p}_i(S_k) = (1 - p^2)^k.$$

Обозначим через $p_i(S_k)$ вероятность, что в S_k нет одновременно x_i и $\neg x_i$.

Эта вероятность $p_i(S_k)$, очевидно не больше вероятности $\tilde{p}_i(S_k)$, ибо в последнем случае разрешается x_i встретиться с $\neg x_i$, если они будут в разных скобках (поскольку мы оцениваем дополнения событий, то дополнение к более узкому событию и дает верхнюю оценку), поэтому

$$p_i(S_k) \leq (1 - p^2)^k.$$

Теперь оценим вероятность того, что S_k — независимое множество, обозначив ее через $P(S_k)$:

$$P(S_k) = \prod_{i=1}^n p_i(S_k) \leq (1 - p^2)^{kn}.$$

Видно, что эта вероятность зависит только от размера множества S_k , поэтому дальше будем обозначать ее через $P(k)$.

Теперь можно оценить математическое ожидание $\max_k |N_k|$:

$$\begin{aligned} \mathbf{E} \max_k |N_k| &\leq \sum_{k=1}^m \binom{m}{k} P(k) \leq \sum_{k=1}^m \binom{m}{k} (1 - p^2)^{kn} \leq \\ &\leq \sum_{k=1}^m m^k \exp\{-np^2 k\} \leq \sum_{k=1}^m \exp\{k(\ln m - np^2)\} \leq \sum_{k=1}^m 1 = m. \end{aligned}$$

Упражнение 3.3.3. Докажите, что

$$\mathbf{E} \max_k |N_k| \leq \sum_{k=1}^m \binom{m}{k} P(k).$$

□



Рис. 3.5: Карта-памятка раздела 3.3.0

3.4 Точность алгоритма для почти всех входов

Хотя человека, использующего жадный алгоритм, можно заставить сильно ошибиться при принятии сложных решений (например, при решении \mathcal{NP} -трудных задач), в типичном случае он обычно выигрывает. Этот почти философский тезис будет нами проиллюстрирован на примере жадного алгоритма для задачи о покрытии.

В данном разделе мы рассмотрим некоторое ослабление анализа сложности в среднем. Это ослабление происходит в двух направлениях: во-первых, мы рассматриваем не только точные алгоритмы, но и приближенные, а во-вторых, анализ сложности в среднем заменяется на анализ для «почти всех» исходных данных.

Ранее мы доказали, что для задачи о покрытии жадный алгоритм (см. раздел 2.1.1) в худшем случае гарантирует нахождение покрытия, размер которого превосходит размер минимального покрытия не бо-

лее чем в логарифмическое число раз (по m). Наша цель сейчас — показать, что для «типовых» данных (в отличие от худшего случая, см. упражнение 2.1.2) это отношение близко к единице, т. е. размер покрытия, найденного жадным алгоритмом, почти равен размеру минимального покрытия.

Как и в предыдущем разделе, мы будем использовать формулировку задачи о покрытии на языке $(0, 1)$ -матриц и целочисленных линейных программ:

$$\begin{aligned} \mathbf{c}\mathbf{x} &\rightarrow \min, \\ A\mathbf{x} &\geq \mathbf{b}, \\ \forall j \quad x_j &\in \{0, 1\}. \end{aligned} \tag{3.5}$$

В этой целочисленной линейной программе n столбцов-переменных x_1, \dots, x_n соответствуют подмножествам S_1, \dots, S_n и их включению в решение-покрытие. Матрица A также является матрицей инцидентности, а векторы стоимости и ограничений — также векторы размерности n и m соответственно, состоящие полностью из единиц.

Таким образом, m строк-ограничений, соответствующих m предметам, очевидным образом выражают ограничение на обязательное включение предмета хотя бы в одно из покрывающих подмножеств, а целевая функция равна числу таких подмножеств.

Как и в разделе 3.2, предположим, что $A = (a_{ij})$ — случайная $(0, 1)$ -матрица, такая, что для всех i, j выполнено

$$P\{a_{ij} = 1\} = p, \quad P\{a_{ij} = 0\} = 1 - p.$$

Пусть $R_G = \frac{Z_G}{M}$, где Z_G — величина покрытия, найденного жадным алгоритмом, а M — величина минимального покрытия. Наш основной результат заключается в следующей теореме.

Теорема 11. Пусть вероятность $0 < p < 1$ фиксирована, и пусть для задачи (3.5) со случайной матрицей A , определенной выше, выполнены следующие условия:

$$\forall \gamma > 0 \quad \frac{\ln n}{m^\gamma} \rightarrow 0, \text{ при } n \rightarrow \infty, \quad (3.6)$$

$$\frac{\ln m}{n} \rightarrow 0, \text{ при } n \rightarrow \infty. \quad (3.7)$$

Тогда для любого фиксированного $\varepsilon > 0$

$$\mathbb{P}\{R_G \leq 1 + \varepsilon\} \rightarrow 1 \text{ при } n \rightarrow \infty.$$

Заметим, что условия теоремы требуют, чтобы матрица A не была «экспоненциально вытянутой» по высоте или ширине. В частности, для «пропорциональных» матриц, у которых $m = cn$, где c — некоторая константа, и «несильно перекошенных матриц», у которых $m = O(n^c)$, условия (3.6) и (3.7) теоремы 11 выполнены автоматически.

Доказательство. Сначала докажем нижнюю оценку размера минимального покрытия.

Пусть X — случайная величина, равная числу покрытий размера

$$l_0 = l_0(\delta) = -\lceil (1 - \delta) \ln m / \ln(1 - p) \rceil,$$

тогда

$$\mathbf{E} X = \binom{n}{l_0} P(l_0),$$

где $P(l_0)$ — вероятность того, что фиксированные l_0 столбцов являются покрытием A . Нетрудно увидеть (убедитесь, что это действительно нетрудно!), что

$$P(l_0) = (1 - (1 - p)^{l_0})^m \leq \exp\{-m(1 - p)^{l_0}\}.$$

Используя неравенство $\binom{n}{k} \leq n^k$, получаем

$$\begin{aligned} \ln \mathbf{E} X &\leq l_0 \ln n - m(1 - p)^{l_0} \leq \\ &\leq -\frac{\ln m}{\ln(1 - p)} \ln n - m \exp\left\{-(1 - \delta) \ln(1 - p) \frac{\ln m}{\ln(1 - p)}\right\} \leq \\ &\leq -\frac{\ln m}{\ln(1 - p)} \ln n - mm^{-1}m^\delta \leq -\frac{\ln m}{\ln(1 - p)} \ln n - m^\delta. \end{aligned}$$

Проверим, что для любого фиксированного $0 < \delta < 1$ при условиях (3.6) последнее выражение стремится к $-\infty$ при n , стремящемся к бесконечности. Действительно, положив $\gamma = \delta/2$ в условии 3.6, получим требуемое, т.к. $\ln m$ растет медленнее, чем $m^{\delta/2}$.

Таким образом, вероятность того, что нет покрытия размера l_0 в случайной $(0, 1)$ -матрице, стремится к 1, поскольку согласно первому неравенству Чебышева⁵:

$$P\{X \geq 1\} \leq \mathbf{E} X \rightarrow 0.$$

Последнее означает, что $P\{M \geq l_0\} \rightarrow 1$.

Теперь докажем верхнюю оценку размера покрытия, построенного жадным алгоритмом.

Используем следующую известную лемму [AS92] о вероятности больших уклонений для сумм независимых случайных величин.

⁵Иногда упоминается как *неравенство Маркова*.

Лемма 8. Пусть Y — сумма n независимых случайных величин, каждая из которых принимает значение 1 с вероятностью p и 0 с вероятностью $1 - p$. Тогда

$$\mathbb{P}\{|Y - np| > \delta np\} \leq 2 \exp\{-(\delta^2/3)np\}.$$

По этой лемме вероятность того, что некоторая фиксированная строка содержит менее чем $(1 - \delta)pn$ единиц (или более чем $(1 + \delta)pn$ единиц) оценивается сверху так:

$$P_{bad} \leq 2 \exp\{-(\delta^2/3)np\},$$

и математическое ожидание числа таких строк не превосходит mP_{bad} . Тогда при $n \rightarrow \infty$, по условию (3.7) получаем

$$mP_{bad} \leq 2 \exp\{\ln m - (\delta^2/3)np\} = 2 \exp\{\ln m - O(1)n\} \rightarrow 0.$$

Из $P\{X \geq 1\} \leq \mathbb{E} X$ — первого неравенства Чебышева следует, что вероятность события «каждая строка содержит не менее $(1 - \delta)pn$ единиц» стремится к 1.

Теперь мы можем почти дословно повторить получение верхней оценки размера покрытия, построенного жадным алгоритмом, в худшем случае.

Пусть N_t — число непокрытых строк после t -го шага жадного алгоритма. Подсчитывая нижнюю оценку числа единиц в подматрице, образованной непокрытыми строками (N_{t-1}), и делая вывод о существовании столбца в этой подматрице с числом единиц не менее среднего, имеем:

$$\begin{aligned} N_t &\leq N_{t-1} - \frac{N_{t-1}(1 - \delta)pn}{n} = N_{t-1}(1 - (1 - \delta)p) \leq \\ &\leq N_0(1 - (1 - \delta)p)^t = m(1 - (1 - \delta)p)^t. \end{aligned}$$

Первое неравенство получается оценкой снизу суммарного числа единиц в подматрице из N_{t-1} непокрытых строк (это $N_{t-1}(1-\delta)pn$) и выводом, что найдется столбец, содержащий не менее среднего числа единиц (равного $\frac{N_{t-1}(1-\delta)pn}{n}$)

Найдем максимальное t , при котором еще есть непокрытый элемент:

$$m(1 - (1 - \delta)p)^t \geq 1.$$

Получим

$$\ln m + t \ln(1 - (1 - \delta)p) \geq 0,$$

или

$$t \leq -\frac{\ln m}{\ln(1 - (1 - \delta)p)}.$$

Отсюда получается верхняя оценка мощности «жадного покрытия» ($Z_G = t + 1$) в типичном случае:

$$\mathbb{P}(Z_G \leq 1 - \frac{\ln m}{\ln(1 - p(1 - \delta))}) \rightarrow 1.$$

Комбинируя это неравенство с нижней оценкой, можно получить, что для любого фиксированного $\varepsilon > 0$ и достаточно больших n

$$\mathbb{P}(R_G \leq 1 + \varepsilon) \rightarrow 1.$$

Действительно,

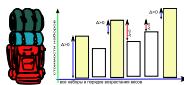
$$\ln(1 - p(1 - \delta)) \geq \ln(1 - p),$$

и для любого $\varepsilon > 0$ существует $\delta > 0$, такое, что

$$R_G \leq \frac{\ln(1 - p)}{(1 - \delta) \ln(1 - p(1 - \delta))} + o(1) \leq (1 - \delta)^{-1} + o(1) \leq 1 + \varepsilon.$$

□

3.5 «Рюкзак»: полиномиальность в среднем



Полиномиальность «в среднем» алгоритма динамического программирования Немхаузера–Ульмана для задачи о рюкзаке ([BV03]).

Итак, рассмотрим еще раз задачу 13 «Knapsack». Как мы видели в разделе 2.2.1, для задачи 13 «Knapsack» существует алгоритм 25 «Рюкзак Немхаузера–Ульмана», поддерживающий в процессе своей работы множество парето-оптимальных или, как иногда говорят, доминирующих решений (определение 2.2.1 «Парето-рюкзак»). Сложность этого алгоритма — $O(n \cdot |\text{ParetoSolutions}|)$, т. е. напрямую зависит от конечного размера множества доминирующих решений для данного входного рюкзака. И хотя можно показать, что существуют такие входные наборы, для которых размер множества доминирующих решений будет экспоненциальным по n (упражнение 2.2.5), для практических целей интересно проанализировать, насколько «часто» встречаются такие плохие входные наборы, возможно ли ожидать от этого алгоритма полиномиального времени работы «в среднем».

Оказалось, справедлива следующая теорема, доказательство которой и составляет все содержание этого раздела.

Теорема 12. Пусть

a_i — «веса», произвольные положительные числа;

c_i — «стоимости», независимые случайные величины, равномерно распределенные на $[0, 1]$;

$q = \max |\text{ParetoSolutions}|$ — число доминирующих подмножеств для всех n предметов.

Тогда

$$\mathbf{E}(q) = O(n^3).$$

Сначала введем некоторые определения.

Пусть $m = 2^n$, S_1, \dots, S_m — подмножества $[n]$ в порядке неубывания весов. Сразу заметим, что веса $\sum_{i \in S_k} a_i$ множеств S_k в нашем рассмотрении не возникают.

Для фиксированной $u \in [2 \dots m]$ и переменной $k \in [1 \dots u]$ определим:

$Plus_k$ — предметы в S_u , которых нет в S_k ;

$Minus_k$ — предметы в S_k , которых нет в S_u ;

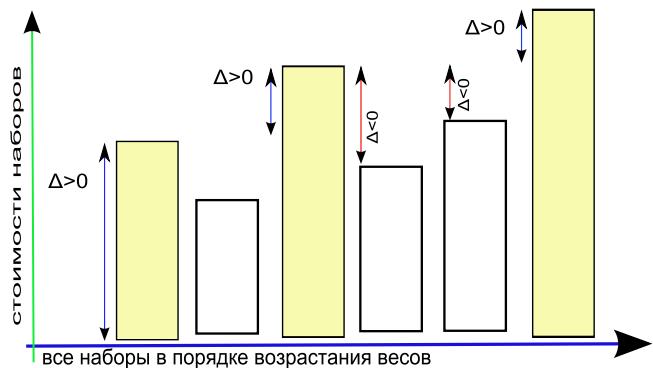
Δ_k^+ — сумма стоимостей предметов в $Plus_k$;

Δ_k^- — сумма стоимостей предметов в $Minus_k$;

Δ_u — минимальная разность $\Delta_k^+ - \Delta_k^-$, по всем k меньшим u ; условно говоря, это «максимальное преимущество в стоимости» над **всеми** предыдущими подмножествами S_1, \dots, S_{u-1} .

Или, более формально:

$$\begin{aligned} Plus_k &= S_u \setminus S_k, \\ Minus_k &= S_k \setminus S_u, \\ \Delta_k^+ &= \sum_{i \in Plus_k} c_i, \\ \Delta_k^- &= \sum_{i \in Minus_k} c_i, \end{aligned}$$



Следующее утверждение следует напрямую из этих определений и определения 2.2.1 «Парето-рюкзак»:

Лемма 9. $\forall u \geq 2, S_u$ — доминирующее множество тогда и только тогда, когда $\Delta_u > 0$.

Учитывая, что стоимость самого последнего множества S_m ограничена по крайней мере n , то интуитивно уже можно утверждать, что чем больше в среднем будет прирост стоимости $\Delta_u > 0$ для каждого доминирующего набора u , тем меньше в среднем будет размер всего множества парето-решений *ParetoSolutions*.

Далее мы последовательно:

1. оценим вероятность «неблагоприятных» с точки зрения минимизации размера *ParetoSolutions* событий, таких, как небольшой прирост $\Delta_u > 0$ для нового доминирующего подмножества;
2. определим математическое ожидание прироста стоимости $\Delta_u > 0$ для нового доминирующего подмножества u ;

3. получим мат. ожидание размера $ParetoSolutions$ и, следовательно, математическое ожидание времени работы алгоритма.

Итак, рассмотрим некоторое u , пусть $S_u = [1, \dots, t]$ (это только вопрос удобства — нам ничего не мешает перенумеровать предметы в исходном наборе).

Тогда:

$$\forall 1 \leq k < u \quad Plus_k \subseteq [1, \dots, t], \quad Minus_k \subseteq [t+1, \dots, n].$$

Для некоторого параметра $0 < \varepsilon < 1$ определим следующие события с достаточно «говорящими» названиями:

$$\begin{aligned} \text{ПаретоНабор} &= \{\Delta_u > 0\} \\ \text{дельтаМала} &= \{\Delta_u < \varepsilon^2\} \\ \text{ЭлементМал}_j &= \{c_j < \varepsilon\} \\ \text{ЭлементНеМал}_j &= \overline{\text{ЭлементМал}}_j = \{c_j \geq \varepsilon\} \\ \text{ВНабореНеМалы} &= \bigcap_{j=1}^t \text{ЭлементНеМал}_j \end{aligned}$$

Оценим вероятность того, что в доминирующем подмножестве есть элемент с «малой» стоимостью.

Лемма 10. $P(\overline{\text{ВНабореНеМалы}} | \text{ПаретоНабор}) \leq n\varepsilon$.

Доказательство. $\forall j \leq t$ и $0 < \varepsilon < 1$ имеем для некоторого x :

$$\mathsf{P}(c_j < \varepsilon \mid \forall k : \Delta_k^+ > \Delta_k^-) = \mathsf{P}(c_j < \varepsilon \mid c_j > x) \leq$$

$$\leq \begin{cases} \text{при } x \geq \varepsilon: & 0 \\ \text{при } x < \varepsilon: & \frac{\varepsilon - x}{1-x} = \frac{\varepsilon(1-x)}{1-x} - \frac{x(1-\varepsilon)}{1-x} \leq \varepsilon. \end{cases}$$

$$\mathsf{P}(\overline{\text{в наборе не малы}} \mid \text{ПаретоНабор}) = \mathsf{P}(\bigcup_{j=1}^t \text{ЭлементМал}_j \mid \text{ПаретоНабор}) \leq \sum_{j=1}^t \mathsf{P}(\text{ЭлементМал}_j \mid \text{ПаретоНабор}) \leq t \cdot \varepsilon \leq n\varepsilon.$$

□

Теперь оценим вероятность того, что прирост стоимости доминирующего подмножества будет «мал», при условии, что в наборе нет «малых» предметов.

Лемма 11. $\mathsf{P}(\text{ДельтаМала} \mid \text{ПаретоНабор} \wedge \text{в наборе не малы}) \leq n\varepsilon.$

Доказательство. Оценим $\mathsf{P}(\text{ПаретоНабор})$ при априорности событий (при условии): « $\forall j c_j \in [0, 1]$ », « в наборе не малы » (и,

следовательно, « $\Delta_k^+ \geq \varepsilon$ »):

$$\begin{aligned}
 P(\text{ПаретоНабор} | \text{ВНабореНеМалы}) &= \\
 &= P(\forall k \Delta_k^- \leq \Delta_k^+, \forall j > t \quad c_j \in [0, 1]) = \\
 &= \frac{1}{(1 - \varepsilon)^{n-t}} P(\forall k \Delta_k^- \leq (1 - \varepsilon) \Delta_k^+, \forall j > t \quad c_j \in [0, 1 - \varepsilon]) \leq \\
 &\leq \frac{1}{(1 - \varepsilon)^{n-t}} P(\forall k \Delta_k^- \leq \Delta_k^+ - \varepsilon^2, \forall j > t \quad c_j \in [0, 1 - \varepsilon]) \leq \\
 &\leq \frac{1}{(1 - \varepsilon)^{n-t}} P(\overline{\text{ДельтаМала}} | \text{ВНабореНеМалы}).
 \end{aligned}$$

Здесь был использован известный факт из геометрии: объем политопа (выпуклой области, задаваемой системой неравенств в $(n - t)$ -мерном пространстве), «уменьшенного» в $1 - \varepsilon$ раз, меньше исходного в $(1 - \varepsilon)^{n-t}$. А вероятность как раз и выражалась через объем политопа!

С другой стороны, $\overline{\text{ДельтаМала}} \subset \text{ПаретоНабор}$, и

$$\begin{aligned}
 P(\overline{\text{ДельтаМала}} | \text{ПаретоНабор} \wedge \text{ВНабореНеМалы}) &= \\
 &= \frac{P(\overline{\text{ДельтаМала}} | \text{ВНабореНеМалы})}{P(\text{ПаретоНабор} | \text{ВНабореНеМалы})} \geq (1 - \varepsilon)^{n-t} \geq (1 - \varepsilon)^n \geq 1 - n\varepsilon.
 \end{aligned}$$

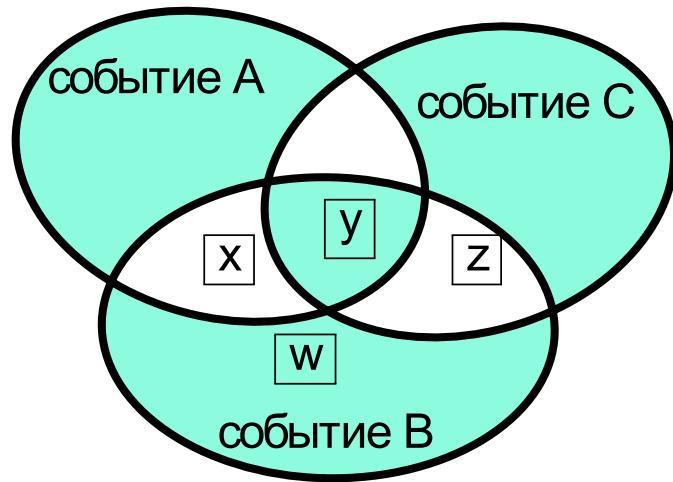
□

Далее нам понадобится небольшая техническая лемма из теории вероятностей.

Лемма 12.

$$P(A|B) = P(A|B \cap C) \cdot P(C|B) + P(A|B \cap \overline{C}) \cdot P(\overline{C}|B).$$

Доказательство. Рассмотрим пересечение событий A, B, C , где «разрезанные» части обозначены x, y, z, w :



Используя соотношение

$$\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)},$$

получим

$$\begin{aligned}
 & P(A|B \cap C) \cdot P(C|B) + P(A|B \cap \bar{C}) \cdot P(\bar{C}|B) = \\
 & = \frac{y}{y+z} \cdot \frac{y+z}{x+y+z+w} + \frac{x}{x+w} \cdot \frac{x+w}{x+y+z+w} = \\
 & = \frac{x+y}{x+y+z+w} = P(A|B).
 \end{aligned}$$

□

Теперь мы готовы оценить вероятность малого прироста стоимости доминирующего подмножества:

Лемма 13. $P(\text{дельтаMала} | \text{ПаретоНабор}) \leq 2n\varepsilon$.

Доказательство. Оценим $P(\text{дельтаMала} | \text{ПаретоНабор})$, используя леммы 10, 11, 12:

$$\begin{aligned}
 P(\text{дельтаMала} | \text{ПаретоНабор}) &= \\
 &= P(\text{дельтаMала} | \text{ПаретоНабор} \cap \text{ВНабореНеМалы}) \cdot P(\text{ВНабореНеМалы} | \text{ПаретоНабор}) \\
 &\quad + P(\text{дельтаMала} | \text{ПаретоНабор} \cap \overline{\text{ВНабореНеМалы}}) \cdot P(\overline{\text{ВНабореНеМалы}} | \text{ПаретоНабор}) \\
 &\leq P(\text{дельтаMала} | \text{ПаретоНабор} \cap \text{ВНабореНеМалы}) + P(\overline{\text{ВНабореНеМалы}} | \text{ПаретоНабор}) \\
 &\leq n\varepsilon + n\varepsilon = 2n\varepsilon.
 \end{aligned}$$

□

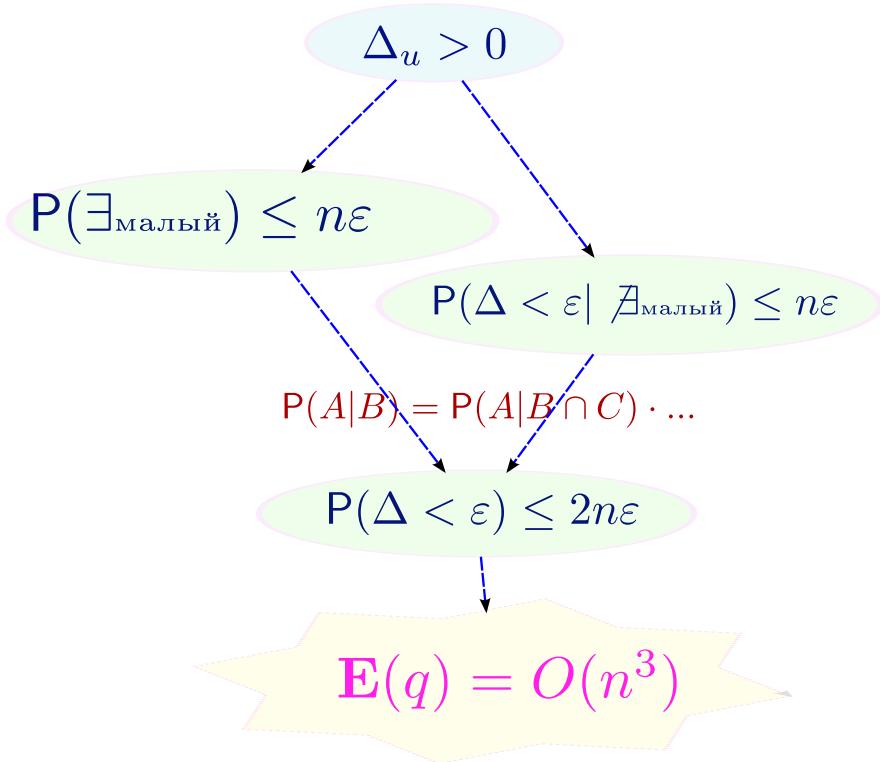


Рис. 3.6: Граф зависимостей утверждений в разделе 3.5

Теперь мы готовы перейти к доказательству самой теоремы. Возьмем $\varepsilon = \frac{1}{3n}$, тогда

$$\begin{aligned}\mathbf{E}(\Delta_u | \Delta_u > 0) &\geq \varepsilon^2 \cdot \mathbf{P}(\overline{\text{дельтаMала}} | \text{ПаретоНабор}) = \\ &= \varepsilon^2 \cdot (1 - \mathbf{P}(\text{дельтаMала} | \text{ПаретоНабор})) \geq \\ &\geq \varepsilon^2 \cdot (1 - 2n\varepsilon) = \frac{1}{9n^2} \cdot (1 - \frac{2}{3}) = \frac{1}{27n^2}.\end{aligned}$$

Обозначим математическое ожидание стоимости S_m через C_m . S_m — это множество максимального веса и стоимости, содержащее все предметы.

С одной стороны,

$$C_m = n \cdot \mathbf{E}(c_i) = \frac{n}{2}.$$

С другой стороны, учитывая, что стоимость пустого набора S_1 равна нулю, стоимость S_m можно подсчитать, как $\sum_{u \in \{2, \dots, m\}: \Delta_u > 0} \Delta_u$, и

$$C_m = \sum_{u=2}^m \mathbf{P}(\Delta_u > 0) \mathbf{E}(\Delta_u | \Delta_u > 0) \geq \frac{1}{27n^2} \sum_{u=2}^m \mathbf{P}(\Delta_u > 0).$$

Откуда получаем

$$\sum_{u=2}^m \mathbf{P}(\Delta_u > 0) = 27n^2 \cdot \frac{n}{2} = \frac{27}{2}n^3.$$

Теперь мы можем выразить математическое ожидание размера множества доминирующих решений:

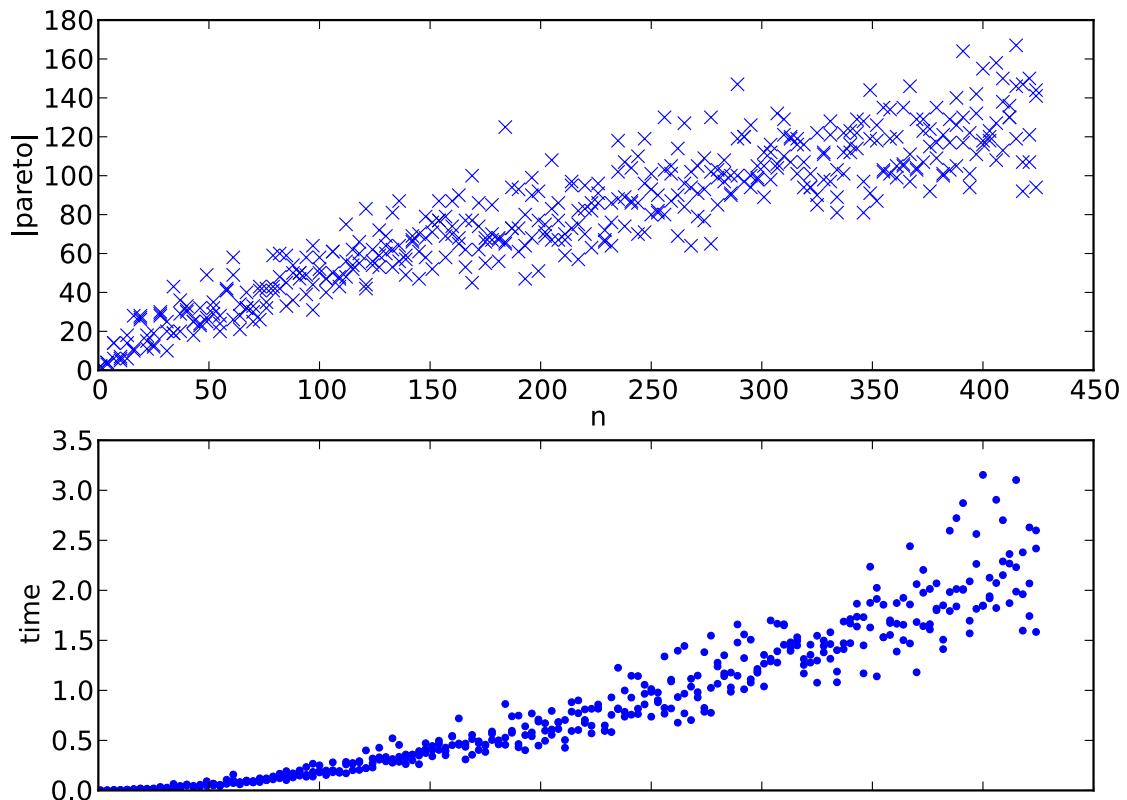
$$\mathbf{E}(q) = 1 + \sum_{u=2}^m \mathbf{P}(\Delta_u > 0) \leq 1 + \frac{27}{2}n^3 = O(n^3).$$

Вспоминая, что сложность алгоритма равна $O(nq)$, получаем полиномиальность алгоритма 25 «Рюкзак Немхаузера–Ульмана» в среднем, разумеется, на случайных входных данных с описанным в начале раздела распределением.

Напомним связь между утверждениями (леммами и теоремами) раздела 3.5 на рис. 3.6.

Следует отметить, что в работе [BV03] доказана полиномиальность в среднем не только для равномерного, но и для других типов вероятностных распределений исходных данных в задаче о рюкзаке.

Некоторые экспериментальные результаты поведения алгоритма 25 «Рюкзак Немхаузера–Ульмана» на случайных данных, можно увидеть на рисунке 3.7. Представлено время (показатель *time*) работы алгоритма 25 «Рюкзак Немхаузера–Ульмана» и максимальный размер множества частичных решений (показатель *ParetoS*) в зависимости от n — числа предметов для рюкзака.



Глава 4

Вероятностные алгоритмы и их анализ

4.1 Вероятностная проверка тождеств

Один из первых примеров вероятностных алгоритмов, более эффективных, чем детерминированные, был предложен Фрейвалдом для задачи проверки матричного равенства $AB = C$ (см. [Fre77]).

Обычный детерминированный алгоритм заключается в перемножении матриц A и B и сравнении результата с C . Сложность такого алгоритма есть $O(n^3)$ при использовании стандартного алгоритма умножения матриц размера $n \times n$ и $O(n^{2.376})$ при использовании лучшего из известных быстрых алгоритмов матричного умножения [CW90].

Вероятностный алгоритм Фрейвалда для этой задачи имеет сложность $O(n^2)$ и заключается в умножении левой и правой частей на случайный булев вектор $\mathbf{x} = (x_1, \dots, x_n)$ с последующим сравнением полученных векторов. Алгоритм выдает ответ, что $AB = C$, если $AB\mathbf{x} = C\mathbf{x}$, и является алгоритмом с од-

носторонней ошибкой, т. е. ошибается только для случаев неравенства — легко видеть, что если алгоритм сообщает, что тождество не выполнено, то всегда $AB \neq C$.

При этом ABx вычисляется как $A(Bx)$, что и обеспечивает оценку сложности алгоритма $O(n^2)$.

Следующая теорема обеспечивает корректность алгоритма.

Теорема 13. Пусть A, B , и C — $n \times n$ матрицы, элементы которых принадлежат некоторому полю \mathbf{F} , причем $AB \neq C$. Тогда для вектора x , выбранного случайно и равномерно из $\{0, 1\}^n$,

$$P\{ABx = Cx\} \leq 1/2.$$

Доказательство. Пусть $D = AB - C$. Мы знаем, что D — не полностью нулевая матрица и хотим оценить вероятность того, что $D\mathbf{x} = 0$. Без ограничения общности можно считать, что ненулевые элементы имеются в первой строке и они располагаются перед нулевыми. Пусть \mathbf{d} — вектор, равный первой строке матрицы D , и предположим, что первые k элементов в \mathbf{d} — ненулевые. Имеем

$$P\{D\mathbf{x} = 0\} \leq P\{\mathbf{d}^T \mathbf{x} = 0\}.$$

Но $\mathbf{d}^T \mathbf{x} = 0$ тогда и только тогда, когда

$$x_1 = \frac{-\sum_{i=2}^k d_i x_i}{d_1}.$$

Для каждого выбора x_2, \dots, x_k правая часть этого равенства фиксирована и равна некоторому элементу v поля \mathbf{F} . Вероятность того, что x_1 равно v , не превосходит $\frac{1}{2}$, поскольку x_1 равномерно распределено на двухэлементном множестве $\{0, 1\}$, а вероятность события $\mathbf{d}^T \mathbf{x} = 0$ либо равна $\frac{1}{2}$, если правая часть принадлежит множеству $\{0, 1\}$, либо равна нулю в противном случае. \square

Классическим примером задачи, где вероятностные алгоритмы традиционно успешно применяются, является задача проверки тождества для многочлена от многих переменных.

Задача 17. Проверить для заданного полинома $P(x_1, \dots, x_n)$ выполнение тождества $P(x_1, \dots, x_n) \equiv 0$.

Следующая лемма (см. [MR95]), по сути, описывает вероятностный алгоритм Монте–Карло с односторонней ошибкой.

Лемма 14. Пусть $Q(x_1, \dots, x_n)$ — многочлен от многих переменных степени d над полем F и пусть $S \subseteq F$ — произвольное подмножество. Если r_1, \dots, r_n выбраны случайно, независимо и равномерно из S , то

$$\Pr(Q(r_1, \dots, r_n) = 0 \mid Q(x_1, \dots, x_n) \not\equiv 0) \leq \frac{d}{|S|}.$$

Доказательство. По индукции по n .

Базисный случай $n = 1$ включает полиномы от одной переменной $Q(x_1)$ степени d . Поскольку каждый такой полином имеет не более d корней, вероятность того, что $Q(r_1) = 0$ не превосходит $\frac{d}{|S|}$.

Пусть теперь предположение индукции верно для всех полиномов, зависящих от не более $n - 1$ переменной.

Рассмотрим полином $Q(x_1, \dots, x_n)$ и разложим его по переменной x_1 :

$$Q(x_1, \dots, x_n) = \sum_{i=0}^k x_1^i Q_i(x_2, \dots, x_n),$$

где $k \leq d$ — наибольшая степень x_1 в Q .

Предполагая, что Q зависит от x_1 , имеем $k > 0$, и коэффициент при x_1^k , $Q_k(r_2, \dots, r_n)$ не равен тождественно нулю. Рассмотрим две возможности.

Первая — $Q_k(r_2, \dots, r_n) = 0$. Заметим, что степень Q_k не превосходит $d - k$, и по предположению индукции вероятность этого события не превосходит $\frac{(d-k)}{|\mathbf{S}|}$.

Вторая — $Q_k(r_2, \dots, r_n) \neq 0$. Рассмотрим следующий полином от одной переменной:

$$q(x_1) = Q(x_1, r_2, r_3, \dots, r_n) = \sum_{i=0}^k Q_i(r_2, \dots, r_n) x_1^i.$$

Полином $q(x_1)$ имеет степень k и не равен тождественно нулю, т.к. коэффициент при x_1^k есть $Q_k(r_2, \dots, r_n)$. Базовый случай индукции дает, что вероятность события

$$q(r_1) = Q(r_1, r_2, \dots, r_n) = 0$$

не превосходит $\frac{k}{|\mathbf{S}|}$.

Мы доказали два неравенства:

$$\mathbb{P}\{Q_k(r_2, \dots, r_n) = 0\} \leq \frac{d - k}{|\mathbf{S}|};$$

$$\mathbb{P}\{Q(r_1, r_2, \dots, r_n) = 0 \mid Q_k(r_2, \dots, r_n) \neq 0\} \leq \frac{k}{|\mathbf{S}|}.$$

Используя результат упражнения 4.1.1, мы получаем, что вероятность события $Q(r_1, r_2, \dots, r_n) = 0$ не превосходит суммы двух вероятностей $(d - k)/|\mathbf{S}|$ и $k/|\mathbf{S}|$, что дает в сумме желаемое $d/|\mathbf{S}|$.

Упражнение 4.1.1. Покажите, что для любых двух событий E_1, E_2 ,

$$\mathsf{P}\{E_1\} \leq \mathsf{P}\{E_1 \mid \overline{E}_2\} + \mathsf{P}\{E_2\}.$$

□

Упражнение 4.1.2. Имеется $n \times n$ матрица A , элементами которой являются линейные функции $f_{ij}(x) = a_{ij}x + b_{ij}$.

Придумайте алгоритм Монте-Карло с односторонней ошибкой для проверки этой матрицы на вырожденность ($\det A \equiv 0$).

Одной из многочисленных областей, где широко применяются вероятностные алгоритмы, являются параллельные и распределенные вычисления. Эффективным параллельным алгоритмом (или *NC*-алгоритмом) называется алгоритм, который на многопроцессорной RAM (так называемой PRAM) с числом процессоров, не превосходящим некоторого полинома от длины входа, завершает работу за время, ограниченное полиномом от логарифма длины входа.

Построение эффективного детерминированного параллельного алгоритма (*NC*-алгоритма) для нахождения максимального паросочетания в двудольном графе является одной из основных открытых проблем в теории параллельных алгоритмов. Удалось, однако, построить эффективный параллельный вероятностный алгоритм нахождения максимального паросочетания в двудольном графе (так называемый *RNC*-алгоритм) [MR95]. Примеры применения вероятностных алгоритмов для построения эффективных параллельных и распределенных алгоритмов рассматриваются в разделе 4.3.

4.2 Вероятностные методы в перечислительных задачах

В данном разделе мы рассмотрим перечислительные задачи и изучим, как вероятностные методы могут быть применены для решения таких задач.

Алгоритм для решения перечислительной задачи получает на вход конкретные входные данные рассматриваемой задачи (например, граф) и в качестве выхода выдает неотрицательное целое, являющееся числом решений задачи (например, число гамильтоновых циклов в графе или число совершенных паросочетаний).

Пусть Z — некоторая перечислительная задача, I — вход задачи. Пусть далее $\#(I)$ обозначает число различных решений для входа I задачи Z .

Определение 4.2.1. Пусть

Z — перечислительная задача.

I — вход для Z , $n = |I|$.

$\#(I)$ — число различных решений для входа I задачи Z .

$\varepsilon > 0$ — параметр точности.

Вероятностный алгоритм $A(I, \varepsilon)$ — **полиномиальная randomизированная аппроксимационная схема**, если время его работы ограничено полиномом от n , и

$$\mathbb{P}[(1 - \varepsilon)\#(I) \leq A(I, \varepsilon) \leq (1 + \varepsilon)\#(I)] \geq \frac{3}{4}.$$

Определение 4.2.2. Полностью полиномиальной рандомизированной аппроксимационной схемой (*FPRAS*) называется полиномиальная рандомизированная аппроксимационная схема, время работы которой ограничено полиномом от n и $1/\varepsilon$.

Определение 4.2.3. Полностью полиномиальной рандомизированной аппроксимационной схемой с параметрами (ε, δ) (или кратко (ε, δ) -*FPRAS*) для перечислительной задачи Z называется полностью полиномиальная рандомизированная аппроксимационная схема, которая на каждом входе I вычисляет ε -аппроксимацию для $\#(I)$ с вероятностью не менее $1 - \delta$ за время, полиномальное от n , $1/\varepsilon$ и $\log 1/\delta$.

Рассмотрим теперь одну перечислительную задачу.

Задача 18. $f(x_1, \dots, x_n) = C_1 \vee \dots \vee C_m$ — булева формула в дизъюнктивной нормальной форме (ДНФ), где каждая скобка C_i есть конъюнкция $L_1 \wedge \dots \wedge L_{k_i}$ k_i литералов (см. определение 3.3.1 «Литерал»).

Набор $v = (v_1, \dots, v_n)$ — **выполняющий для** f , если $f(v_1, \dots, v_n) = 1$.

Найти число выполняющих наборов для данной ДНФ.

Пусть

V — множество всех двоичных наборов длины n .

G — множество выполняющих наборов.

Рассмотрим алгоритм, основанный на стандартном методе Монте-Карло, и покажем, что он не является (ε, δ) -*FPRAS* для этой задачи.

1. Проведем N независимых испытаний:

- выбирается случайно $v \in V$ (в соответствии с равномерным распределением);
 - $y_i = f(v_i)$. Заметим, что $P\{y_i = 1\} = \frac{|G|}{|V|}$. Обозначим эту вероятность через p .
2. Рассмотрим сумму независимых случайных величин $Y = \sum_{i=1}^N y_i$. В качестве аппроксимации $|G|$ возьмем величину
- $$\tilde{G} = \frac{Y}{N}|V|.$$

Упражнение 4.2.1. Почему y_i независимые случайные величины?

Воспользуемся следующей леммой.

Лемма 15. Пусть x_1, \dots, x_n — независимые случайные величины, такие, что x_i принимают значения 0, 1, и $P\{x_i = 1\} = p$, $P\{x_i = 0\} = 1 - p$. Тогда для $X = \sum_{i=1}^n x_i$ и для любого $0 < \delta < 1$, выполнено неравенство

$$P\{|X - \mathbf{E} X| > \delta \mathbf{E} X\} \leq 2 \exp\{-(\delta^2/3) \mathbf{E} X\}.$$

Доказательство леммы изложено в [MR95].

Оценим вероятность того, что аппроксимация хорошая:

$$\begin{aligned} P\{(1 - \varepsilon)|G| \leq \tilde{G} \leq (1 + \varepsilon)|G|\} &= \\ &= P\{(1 - \varepsilon)\frac{N|G|}{|V|} \leq Y \leq (1 + \varepsilon)\frac{N|G|}{|V|}\} = \\ &= P\{(1 - \varepsilon)Np \leq Y \leq (1 + \varepsilon)Np\}. \end{aligned}$$

Применяя неравенства из леммы, получаем:

$$\mathsf{P}\{(1 - \varepsilon)|G| \leq \tilde{G} \leq (1 + \varepsilon)|G|\} > 1 - 2 \exp\{-(\varepsilon^2/3)Np\}.$$

Потребуем, чтобы эта вероятность хорошей аппроксимации была $\geq 1 - \delta$, тогда получим

$$2 \exp\left\{-\frac{\varepsilon^2 N p}{3}\right\} < \delta,$$

$$\frac{2}{\delta} < \exp\left\{\frac{\varepsilon^2 N p}{3}\right\},$$

$$N > \frac{1}{p} \frac{3}{\varepsilon^2} \ln \frac{2}{\delta}.$$

Почему полученная оценка не столь хороша? Потому что p может быть экспоненциально мало (например, если функция равна 1 лишь в одной точке), и тогда число требуемых шагов будет экспоненциально велико.

Покажем, как использовать этот же метод не напрямую, а после некоторой модификации задачи ([KLM89]).

Рассмотрим более общую (по сравнению с задачей о числе выполняющих наборов) задачу.

Задача 19. Пусть V — конечное множество, и имеется m его подмножеств $H_1, \dots, H_m \subseteq V$, удовлетворяющих следующим условиям:

1. $\forall i |H_i|$ вычислимо за полиномиальное время.
2. $\forall i$ возможно выбрать случайно и равномерно элемент из H_i .

3. $\forall v \in V$ за полиномиальное время проверяется $v \stackrel{?}{\in} H_i$.

Рассмотрим объединение $H = H_1 \cup \dots \cup H_m$. Надо оценить его мощность $|H|$.

Связь с исходной задачей 18:

« V »: базовое множество V состоит из всех двоичных наборов длины n .

« H_i »: $H_i = \{v \in V : C_i(v) = 1\}$, т. е. состоит из всех наборов, на которых скобка C_i равна 1 (выполнена).

Условие (1): Пусть r_i — число литералов в C_i , тогда:

$$\begin{cases} C_i \equiv 0 \Rightarrow |H_i| = 0, \\ C_i \not\equiv 0 \Rightarrow |H_i| = 2^{n-r_i}. \end{cases}$$

Условие (2): Случайная выборка элемента из H_i — зафиксировать переменные из C_i , а остальные переменные выбрать случайно.

Условие (3): вычислить $C_i(v)$, т. е. проверка, что некоторое $v \in V$ принадлежит H_i , эквивалентно проверке выполнимости скобки (конъюнкции) на данном наборе.

Идея, позволяющая предложить для рассматриваемой задачи \mathcal{FPRAS} , проста. Мы определим относительно небольшой универсум, подмножеством которого будет $H = H_1 \cup \dots \cup H_m$. Универсум U образуется из точек H , причем точка берется с кратностью, равной числу множеств H_i , которым она принадлежит. Тогда

$$|U| = \sum_{i=1}^m |H_i| \geq |H| \geq \frac{1}{m} \sum_{i=1}^m |H_i| = \frac{1}{m} |U|,$$

$$DNF = (x_3 \cdot \bar{x}_4 \cdot \bar{x}_1) \vee (x_1 \cdot x_4) \vee (x_4 \cdot \bar{x}_2) \vee (x_3 \cdot x_4 \cdot \bar{x}_2) \vee (x_3 \cdot \bar{x}_2)$$

x_1	x_2	x_3	x_4	C_1	C_2	C_3	C_4	C_5	ДНФ
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	0
0	0	1	0	■	.	.	.	□	1
1	0	1	0	■	1
0	1	1	0	■	1
1	1	1	0
0	0	0	1	.	.	■	.	.	1
1	0	0	1	.	■	□	.	.	1
0	1	0	1
1	1	0	1	.	■	.	.	.	1
0	0	1	1	.	.	■	□	□	1
1	0	1	1	.	■	□	□	□	1
0	1	1	1
1	1	1	1	.	■	.	.	.	1

- По вертикали — бинарные наборы v .
- $U = \{(v, i) \mid v \in H_i\}$, элементы соответствуют квадратикам на рисунке. $|U| = \sum_{i=1}^m |H_i| \geq |H|$.
- $\text{cov}(v) = \{(v, i) \mid (v, i) \in U\}$ (квадратики на строчке v), $\text{cov}(v) \leq m$.
- $U = \cup_{v \in H} \text{cov}(v)$, $|U| = \sum_{v \in H} |\text{cov}(v)|$.

и к U можно применять стандартный алгоритм Монте-Карло.

Конкретно для нашей задачи 18 «DNF#» определения и иллюстрация множеств U и H показаны на рис. 4.1. Наша цель теперь заключается в оценке мощности $G \subseteq U$, для чего мы применяем метод Монте-Карло к множеству U :

1. Выбираем случайно и равномерно $u = (v, i) \in U$:
 - (a) выберем i , $1 \leq i \leq m$ с вероятностью $\frac{|H_i|}{|U|} = \frac{|H_i|}{\sum_{i=1}^m |H_i|}$;
 - (b) выбирается случайно и равномерно $v \in H_i$.
2. $y_i = 1$ при $v \in G$, иначе $y_i = 0$.
3. $Y = \sum_{i=1}^N y_i$.
4. $\tilde{G} = \frac{Y}{N}|U|$.

Теорема 14. Метод Монте-Карло дает (ε, δ) -FPRAS для оценки $|G|$ при условии

$$N \geq \frac{3m}{\varepsilon^2} \ln \frac{2}{\delta}.$$

Время полиномиально по N .

Доказательство. Выбор случайного элемента (v, i) из U происходит равномерно из U — для доказательства достаточно перемножить соответствующие вероятности.

Проверка $(v, i) \stackrel{?}{\in} G$ — полиномиальна: $\forall j < i : C_j(v) = 0$.

Из полученных ранее оценок о числе испытаний стандартного метода Монте-Карло имеем

$$N > \frac{1}{p} \frac{3}{\varepsilon^2} \ln \frac{2}{\delta}.$$

С другой стороны

$$|U| = \sum_{v \in H} |\text{cov}(v)| \leq \sum_{v \in H} m \leq m|H| = m|G| \Rightarrow p = \frac{|G|}{|U|} \geq \frac{1}{m}.$$

Получаем, что достаточное число испытаний:

$$N > \frac{3m}{\varepsilon^2} \ln \frac{2}{\delta}.$$

□

Таким образом, описанный метод аппроксимации дает (ε, δ) - \mathcal{FPRAS} для оценки числа выполняющих наборов для любой ДНФ.

4.3 Вероятностные методы в параллельных вычислениях

4.3.1 Максимальное по включению независимое множество в графе

Модели параллельных вычислений

В литературе при описании параллельных алгоритмов чаще всего используются параллельные машины с произвольным доступом к памяти — *PRAM* (*Parallel Random Access Machine*).

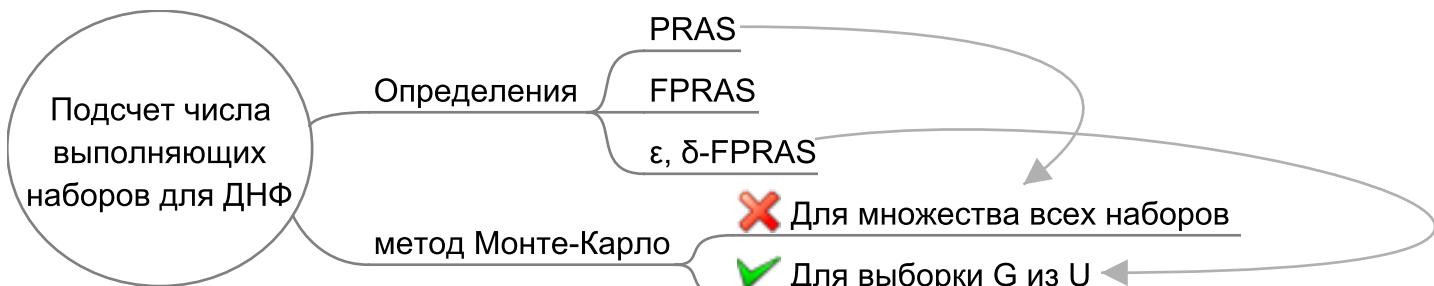


Рис. 4.2: Карта-памятка раздела 4.2.0

Благодаря абстрагированию от многих специфических особенностей параллельных архитектур, модель *PRAM* позволяет достаточно просто реализовывать параллельные алгоритмы, не отвлекаясь на технические детали, возникающие при работе с реальными параллельными архитектурами.

PRAM состоит из r идентичных *RAM*(подробнее о модели *RAM* написано в разделе 1.2.1), называемых процессорами, имеющих общую память (см. рис. 4.3).

Процессоры работают синхронно, и каждый из них может переписывать в свой сумматор содержимое любой ячейки памяти. Команды одноадресные, локальной памяти данных нет, за исключением одного сумматора в каждом процессоре. Каждый процессор работает по своей программе, хранимой в специальной локальной памяти. В *PRAM* имеется одна общая для всех процессоров входная лента, и каждый процессор имеет свою выходную ленту.

Вычисление производится синхронно всеми процессорами, начиная с первой команды. Вычисление заканчивается после того, как каждый процессор завершит свою работу (выполнит команду *HALT*). Результатом вычисления считается набор из r последовательностей чисел на выходных лентах.

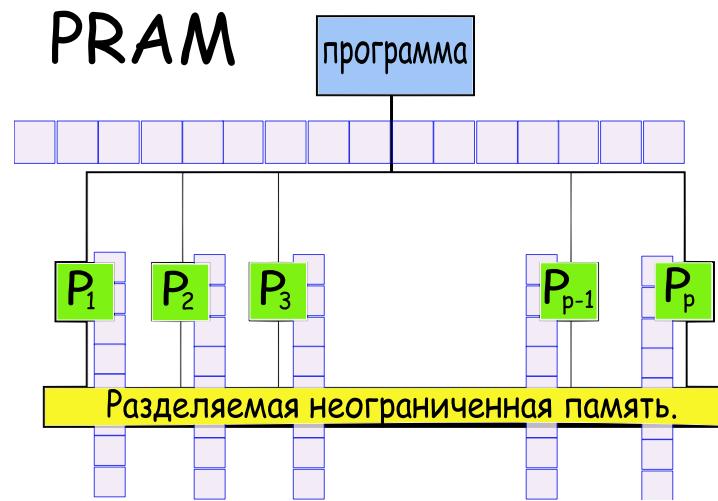


Рис. 4.3: *PRAM — Parallel Random Access Machine*

Процессоры могут писать в память и читать из памяти независимо друг от друга. Ограничением является лишь возможность писать или читать из одной ячейки для нескольких процессоров. Различают три типа *PRAM* в зависимости от того, что происходит, когда несколько процессоров одновременно обращаются к одной ячейке памяти:

EREW PRAM: *exclusive read/exclusive write* — самая слабая модель, процессоры могут читать одновременно только из разных ячеек и писать в разные ячейки памяти.

CREW PRAM: *concurrent read/exclusive write* — некоторым процессорам разрешается одновременное

чтение из одной ячейки памяти, но запрещается одновременная запись в одну ячейку памяти.

CRCW PRAM: *concurrent read/concurrent write* — наиболее сильная модель, разрешается как параллельное считывание из одной ячейки несколькими процессорами, так и параллельная записьическими процессорами в одну ячейку. При этом в зависимости от разрешения конфликта по записи в ячейку выделяют следующие типы *CRCW PRAM*:

WEAK CRCW PRAM — несколько процессоров могут записывать в одну ячейку только число 0.

COMMON CRCW PRAM — несколько процессоров могут записывать в одну ячейку только одноковое число, общее для всех процессоров.

ARBITRARY CRCW PRAM — несколько процессоров могут записывать в одну ячейку разные числа, а содержимое ячейки становится одним из этих чисел, но не известно каким.

PRIORITY CRCW PRAM — результатом записи в ячейку является значение, поступившее от процессора с большим номером.

STRONG CRCW PRAM — результатом записи в ячейку является максимальное значение, поступившее от процессоров.

Заметим, что при реализации алгоритмов для вышеперечисленных моделей *PRAM* на реальных компьютерах, приходится использовать различные программные или аппаратные механизмы обеспечения синхронизации, такие, как различные виды *семафоров* (*semaphors, mutexes*).

И хотя на первый взгляд самой простой для реализации может показаться модель *CRCW PRAM*, отметим, что это не так — ведь это только кажется, что если к памяти можно получить доступ без ограничений, то синхронизация и не требуется. А для реализации, наиболее простой оказывается модель *EREW PRAM*, которая предполагает, по сути, только последовательную обработку разделяемых данных.

Давайте познакомимся с *EREW PRAM*-алгоритмом для одной из задач на графах.

EREW PRAM-алгоритм для поиска максимального независимого множества в графе

Сначала дадим пару определений.

Определение 4.3.1. Пусть $G = (V, E)$ — неориентированный граф с n вершинами и m ребрами.

Подмножество вершин $I \subseteq V$ называется **независимым** в G , если никакое ребро из E не содержит обеих своих конечных вершин в I .

Определение 4.3.2. Независимое множество I называется **максимальным по включению**¹, если оно не содержится в качестве собственного подмножества в другом независимом подмножестве в G .

Задача нахождения максимального по мощности независимого множества \mathcal{NP} -трудна.

Однако для нахождения максимального по включению независимого множества существует тривиальный последовательный алгоритм 30 с временной сложностью $O(n)$.

Лемма 16. Жадный алгоритм 30 выдает максимальное по включению независимое множество и имеет сложность $O(n)$ при условии, что граф на вход подается в виде списка смежности.

Проблема заключается в том, чтобы построить эффективный параллельный алгоритм для этой задачи.

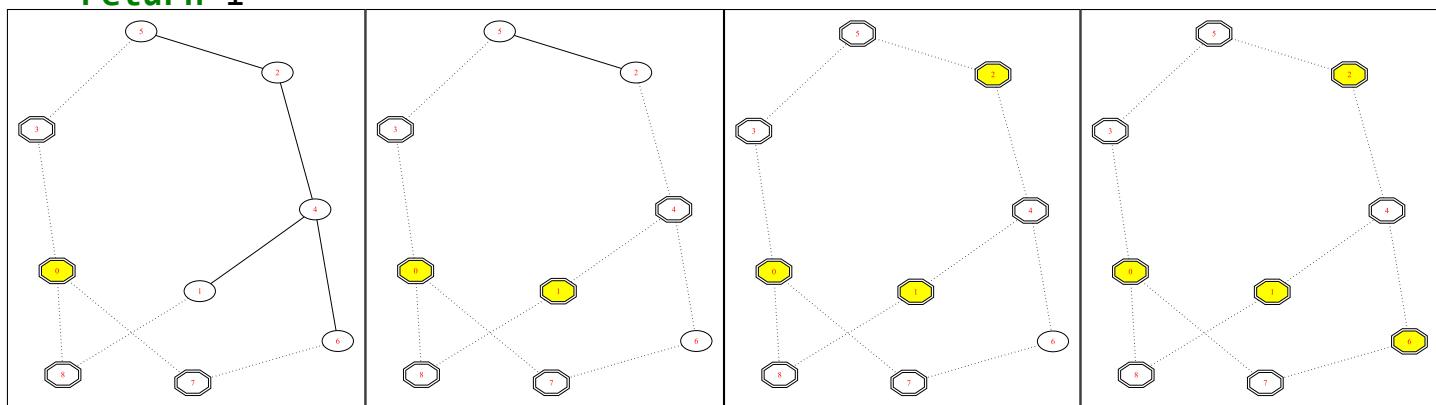
Определение 4.3.3. Пусть S — подмножество вершин, обозначим через $\Gamma(S)$ множество вершин соседних с S .

Неформально, идея параллельного алгоритма для поиска MIS, предложенного в [Lub85], состоит в том, что на каждой итерации находится независимое множество S , которое добавляется к уже построенному I , а $S \cup \Gamma(S)$ удаляется из графа.

¹В англоязычной литературе — *Maximal Independent Set*.

Алгоритм 30 Алгоритм MIS-Greedy

```
def greedyMIS(G):
    I = set([])
    while G.number_of_nodes() > 0: # цикл сложности  $O(n)$ 
        n = G.nodes()[0] # первая попавшаяся вершина
        I.add(n) # добавляем одну вершину из ребра
        for s in G.neighbors(n): # удаляем всех соседей
            G.remove_node(s)
        G.remove_node(n) # и исходную вершину
    return I
```



- Закрашенные вершины составляют множество I .
- Удаленные ребра показаны пунктиром, удаленные вершины — многоугольной формы.

Чтобы число итераций было небольшим, независимое множество S должно быть таким, чтобы $S \cup \Gamma(S)$ было большим.

Напрямую этого добиться трудно, но мы добиваемся того же эффекта, доказывая, что число ребер, инцидентных $S \cup \Gamma(S)$, составляет значительную долю оставшихся ребер.

Для реализации этой идеи мы выбираем большое случайное множество вершин $R \subseteq V$. Маловероятно, что R будет независимым, но, с другой стороны, имеется немного ребер, оба конца которых принадлежат R . Для получения независимого множества из R мы рассмотрим такие ребра и удалим концевые вершины с меньшей степенью.

Эта идея реализована в алгоритме 31 (вероятностный параллельный алгоритм Луби нахождения максимального по включению независимого множества в графе). Предполагается, что каждой вершине и каждому ребру приписан свой процессор. Таким образом, требуемое число процессоров есть $O(n+m)$. Через $d(v)$ обозначается степень вершины v в графе, т. е. число инцидентных ей ребер.

Упражнение 4.3.1. Покажите, что каждая итерация алгоритма может быть реализована за время $O(\log n)$ на модели EREW PRAM с $(n+m)$ процессорами.

Цель дальнейшего анализа показать, что случайная пометка вершин обеспечивает математическое ожидание числа итераций $O(\log n)$.

Определение 4.3.4. Вершина $v \in V$ называется **хорошой**, если она имеет не менее $\frac{d(v)}{3}$ соседних вершин степени не более $d(v)$. В противном случае вершина называется **плохой**.

Более формально:

- $\Gamma^*(v) = \{w \in \Gamma(v) : d(w) \leq d(v)\};$
- $v \in V$ — **хорошая**, если $|\Gamma^*(v)| \geq \frac{d(v)}{3}$, и **плохая** в противном случае.

Алгоритм 31 Алгоритм «Parallel MIS»

Вход: Граф $G = (V, E)$.

```

 $I \leftarrow \emptyset$ 
while  $V \neq \emptyset$  do
    for all  $v \in V$  (параллельно) do
        if  $d(v) = 0$  then
             $V \leftarrow V \setminus \{v\}$ ,  $I \leftarrow I \cup \{v\}$ 
        else
             $marked(v) \leftarrow 1$  с вероятностью  $\frac{1}{2d(v)}$ 
        end if
    end for
    for all  $(u, v) \in E$  (параллельно) do
        if  $marked(v) \wedge marked(u)$  then
             $marked(v) \leftarrow 0$  ( $d(v) \leq d(u)$ )
        end if
    end for
    for all  $v \in V$  (параллельно) do
        if  $marked(v)$  then
             $S \leftarrow S \cup \{v\}$ 
        end if
    end for
     $V \leftarrow V \setminus (S \cup \Gamma(S))$  {и все инцидентные ребра из  $E$ }
     $I \leftarrow I \cup S$ 
end while

```

Выход: Максимальное-независимое $I \subseteq V$.

Лемма 17. Пусть $v \in V$ — хорошая вершина степени $d(v) > 0$. Тогда вероятность того, что некоторая вершина $w \in \Gamma(v)$ становится отмеченной, не меньше $1 - \exp(-1/6)$:

$$\mathbb{P}\{\exists w \in \Gamma(v) : \text{marked}(w) = 1\} \geq 1 - \exp\left(-\frac{1}{6}\right).$$

Доказательство. Каждая вершина $w \in \Gamma(v)$ помечается независимо с вероятностью $\frac{1}{2d(w)}$. Поскольку v является хорошей, то размер $\Gamma^*(v)$ (множества соседей v со степенью не более $d(v)$) не меньше $\frac{d(v)}{3}$, а каждый из этих соседей также будет отмечен с вероятностью не менее $\frac{1}{2d(v)}$. Значит, вероятность P_{bad} , что ни одна из этих соседних v вершин не будет помечена, не превосходит

$$\begin{aligned} P_{bad} &= \mathbb{P}\{\nexists w \in \Gamma(v) : \text{marked}(w) = 1\} \leq \prod_{w \in \Gamma(v)} \left(1 - \frac{1}{2d(w)}\right) \leq \\ &\leq \prod_{w \in \Gamma^*(v)} \left(1 - \frac{1}{2d(w)}\right) \leq \left(1 - \frac{1}{2d(v)}\right)^{d(v)/3} \leq e^{-1/6}. \end{aligned}$$

□

Лемма 18. На каждой итерации помеченная вершина выбирается в S с вероятностью не менее $1/2$.

Доказательство. Единственная причина того, что помеченная вершина w становится непомеченной и, следовательно, не выбранной в S , является наличие соседней помеченной вершины степени не менее $d(w)$. Каждая такая соседняя вершина помечена с вероятностью, не превышающей $\frac{1}{2d(w)}$, и число таких соседних вершин не превышает, конечно, $d(w)$.

Таким образом, вероятность «плохого» события, что помеченная вершина не будет выбрана в S , не больше, чем

$$\begin{aligned} P_{bad} &= \mathbb{P}\{\exists v \in \Gamma(w) : d(v) \geq d(w) \wedge \text{marked}(v)\} \leq \\ &\leq |\{v \in \Gamma(w) \mid d(v) \geq d(w)\}| \times \frac{1}{2d(w)} \leq \\ &\leq |\{\text{marked}(v) \mid v \in \Gamma(w)\}| \times \frac{1}{2d(w)} = d(w) \times \frac{1}{2d(w)} = \frac{1}{2}. \end{aligned}$$

□

Комбинируя леммы 17 и 18, получаем лемму 19.

Лемма 19. Вероятность того, что хорошая вершина принадлежит множеству $S \cup \Gamma(S)$ не меньше, чем $(1 - e^{-1/6})/2$.

Определение 4.3.5. Ребро называется **хорошим**, если хотя бы одна из его концевых вершин является хорошей, иначе — **плохим** ($\in E_{bad}$).

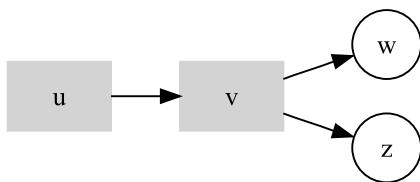
Финальный шаг заключается в оценке числа хороших ребер.

Лемма 20. В графе $G = (V, E)$ число хороших ребер не меньше, чем $|E|/2$.

Доказательство. Пусть $\binom{E}{2}$ обозначает множество всех неупорядоченных пар из множества E . Обозначим множество всех плохих ребер через E_{bad} .

Определим функцию $f : E_{bad} \rightarrow \binom{E}{2}$ так, чтобы для всех $e_1 \neq e_2 \in E_{bad}$ выполнено $f(e_1) \cap f(e_2) = \emptyset$.

Ориентируем каждое ребро графа E от вершины меньшей степени в сторону вершины большей степени (в случае равенства — произвольно). Пусть $(u \rightarrow v) \in E_{bad}$.



Ребру $(u \rightarrow v)$ сопоставим любую пару ребер, выходящих из v . То, что такая пара найдется, доказывается в упражнении 4.3.2.

Упражнение 4.3.2. Докажите, что для «плохой» вершины v число выходящих из нее ребер, по крайней мере, вдвое превосходит число входящих в нее ребер.

т. е. каждому плохому ребру можно сопоставить «личную» пару ребер («плохих» или «хороших») — неважно, главное, что каждое ребро $e \in E$ «принадлежит» не больше чем одному плохому ребру, а пар ребер не больше чем $\frac{|E|}{2}$). Отсюда (и из упражнения 4.3.2) следует, что $|E_{bad}| \leq \frac{|E|}{2}$.

Поскольку константная доля ребер инцидентна хорошим вершинам, а хорошие вершины удаляются с константной вероятностью, то математическое ожидание числа ребер, удаленных на итерации, составляет константную долю от текущего числа ребер. Отсюда легко следует, что математическое ожидание числа итераций алгоритма 31 есть $O(\log n)$ (докажите это в качестве упражнения).

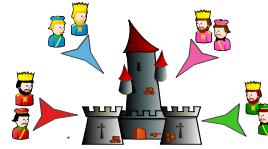
□

Теорема 15. Алгоритм 31 имеет реализацию на EREW PRAM с $O(n + m)$ процессорами, такую, что математическое ожидание времени работы есть $O(\log^2 n)$.

Доказательство. Вытекает из вышесказанного и результата упражнения 4.3.1.

□

4.3.2 Протокол византийского соглашения



Вероятностные методы в построении параллельных и распределенных алгоритмов. Задача о византийских генералах. Распределенный вероятностный протокол византийского соглашения.

В данном разделе мы рассмотрим классическую проблему теории распределенных вычислений о достижении византийского соглашения. Историческая аналогия, давшая название данному протоколу, заключается в выработке соглашения генералами Византии (наступать или не наступать) при условии, что некоторые генералы могут посыпать заведомо ложные сообщения (т. е. быть предателями).

Проблема византийского соглашения заключается в следующем (см. [Rab83]). Каждый из n процессоров имеет сначала значение 0 или 1 (в терминологии византийских генералов это соответствует мнению данного генерала «наступать — не наступать»). Пусть b_i — начальное значение i -го процессора. Имеется t ошибочно работающих процессоров, а остальные $n - t$ процессоров рассматриваются как идентичные и исправные. Более того, мы будем предполагать, что неисправные процессоры могут объединять свои усилия по предотвращению достижения соглашения. Предлагаемый протокол должен выстоять против таких атак. Между процессорами возможен обмен сообщениями по правилам, описанным ниже, при которых i -й процессор заканчивает протокол с решением $d_i \in \{0, 1\}$, которое должно удовлетворять следующим требованиям.

Требование 1. Все исправные процессоры должны завершить протокол с идентичным решением.

Требование 2. Если все исправные процессоры начинают протокол обмена с одного и того же значения v , тогда они все должны закончить с общим решением, эквивалентным v .

Множество неисправных процессоров задается до выполнения протокола, хотя, конечно, исправные

процессоры их не знают. Протокол выработки соглашения состоит из нескольких раундов. В течение одного раунда каждый процессор может послать по одному сообщению любому другому процессору. При этом до начала следующего раунда каждый процессор получает сообщения от всех остальных процессоров.

Процессор не обязан посылать одно и то же сообщение остальным процессорам. На самом деле, в описанном ниже протоколе каждое сообщение состоит из одного бита. Все исправные процессоры следуют в точности протоколу. Неисправные процессоры могут посыпать сообщения, полностью не соответствующие протоколу, и к тому же могут посыпать разные сообщения разным процессорам.

Можно даже считать, что неисправные процессоры перед началом каждого раунда договариваются между собой, какие сообщения послать каким процессорам с целью нанесения наибольшего урона (путаницы).

Соглашение считается достигнутым, если все исправные процессоры вычислили решения, удовлетворяющие двум свойствам, перечисленным выше.

Мы будем изучать число раундов, необходимых для достижения соглашения. Известно, что любой детерминированный протокол достижения соглашения требует не менее $t + 1$ раунда.

Сейчас мы представим простой рандомизированный алгоритм достижения соглашения с математическим ожиданием числа раундов, равным константе. Предполагается, что имеется глобальная «процедура подбрасывания монетки», доступная всем на каждом шаге, которая работает корректно и не подвержена ошибкам. Процедура выдает 1 и 0 независимо с вероятностью каждого исхода 1/2.

Для простоты будем полагать, что $t < n/8$. Пусть

$$L = 5n/8 + 1, \quad H = 6n/8 + 1, \quad G = 7n/8 + 1.$$

На самом деле для правильной работы протокола достаточно, чтобы $L \geq n/2 + t + 1$ и $H \geq L + t$.

В распределенном протоколе i -й ($1 \leq i \leq n$) процессор выполняет алгоритм 32, в котором *coin* обозначает результат подбрасывания монеты (случайный бит).

Подчеркнем, что неисправные процессоры не обязаны следовать протоколу и работают произвольно.

Алгоритм 32 Протокол византийского соглашения

Algorithm ByzGen:

Вход: Значение b_i

Выход: Решение d_i

- $vote := b_i$
- Для каждого раунда **выполнить**:
 1. Разослать всем процессорам $vote$.
 2. Получить $votes$ от всех остальных процессоров.
 3. Вычислить maj — самое часто встречающееся значение среди $votes$ (включая собственный).
 4. Вычислить $tally$ — число появлений maj среди полученных $votes$.
 5. **if** $coin = 1$ **then** $threshold := L$
else $threshold := H$.
 6. **if** $tally \geq threshold$ **then** $vote := maj$
else $vote := 0$.
 7. **if** $tally \geq G$ **then** присвоить d_i значение maj постоянно.

Отметим, что хотя переменная $vote$ каждого процессора и получает некоторое значение на шаге 6, оно

не является окончательным и может измениться в дальнейшем. Окончательные значения переменные d_i получают на шаге 7.

Лемма 21. *Если все правильные процессоры начинают раунд с одного и того же значения, то все они принимают решение, равное этому значению, за константное число раундов.*

Более интересен случай для анализа, когда не все правильные процессоры начинают работу с одинакового значения.

В отсутствие ошибочных процессоров для всех процессоров достаточно разослать всем свои значения, после чего все согласовано принимают решение в соответствии с «большинством голосов».

Описанный протокол реализует ту же идею, но в присутствии ошибочных процессоров.

Если два правильных процессора вычисляют разные значения для taj на шаге 3, *tally* не превосходит *threshold* вне зависимости от того, было ли выбрано значение для *threshold L* или *H* (поскольку разность между $n/2$ и обоими порогами больше t — числа неисправных процессоров).

Тогда все правильные процессоры присваивают *vote* значение 0 на шаге 6. В результате все правильные процессоры полагают свои решения равными нулю в следующем раунде. Таким образом, остается рассмотреть случай, когда все правильные процессоры вычисляют одно и то же значение для *taj* на шаге 3.

Скажем, что неисправные процессоры обманывают порог $x \in \{L, H\}$ в некотором раунде, если, посылая различные сообщения правильным процессорам, они вынуждают *tally* превзойти значение x хотя бы для одного правильного процессора и быть не более x хотя бы для одного правильного процессора. Поскольку разница между двумя порогами *L* и *H* больше t , неисправные процессоры могут обмануть не более одного порога за раунд. Так как порог выбран равновероятно из $\{L, H\}$, обман происходит с вероятностью, не превосходящей $1/2$. Значит, математическое ожидание числа раундов до получения порога без обмана равно 2. Если порог не обманут, то все правильные процессоры вычисляют одно и то же значение *v* для *vote* на шаге 6.

В следующем раунде каждый правильный процессор получает по крайней мере $G > H > L$ голосов за v и присваивает *maj* значение v на шаге 3. Затем на шаге 7 *tally* превосходит выбранный (любым образом) порог. Когда правильный процессор принимает решение d_i , остальные правильные процессоры должны иметь $tally \geq threshold$, поскольку $G \geq H + t$. Следовательно, они все будут голосовать за одно и то же значение d_i .

Теорема 16. Математическое ожидание числа раундов для алгоритма *ByzGen* для достижения византийского соглашения ограничено константой.

4.4 Вероятностное округление

4.4.1 Вероятностное округление для задачи «MAX-SAT»

Рассмотрим следующую задачу ([GW94]).

Задача 20. Максимальная выполнимость/MAX-SAT.

Даны t скобок конъюнктивной нормальной формы (КНФ) с n переменными. Найти значения переменных, максимизирующие число выполненных скобок.

В качестве примера рассмотрим следующую КНФ:

$$(x_1 \vee \bar{x}_2)(x_1 \vee x_2 \vee \bar{x}_3)(x_2 \vee \bar{x}_1)(x_3 \vee \bar{x}_2).$$

Переменные или их отрицания, входящие в скобку, называются литералами (так в первой скобке литералами являются x_1 и \bar{x}_2).

Задача MAX-SAT является \mathcal{NP} -трудной, но мы рассмотрим сейчас простое доказательство того, что для любых выполнимых m скобок существуют значения переменных, при которых выполнено не менее $m/2$ скобок.

Теорема 17. Для любых m выполнимых скобок (т. е. скобок, не содержащих одновременно переменную и ее отрицание) существуют значения переменных, при которых выполнено не менее $m/2$ скобок.

Доказательство. Предположим, что каждой переменной приписаны значения 0 или 1 независимо и равновероятно. Для $1 \leq i \leq m$ пусть $Z_i = 1$, если i -я скобка выполнена, и $Z_i = 0$ в противном случае.

Для каждой дизъюнкции (скобки) с k литералами (переменными или их отрицаниями) вероятность, что эта дизъюнкция не равна 1 при случайном приписывании значений переменным, равна 2^{-k} , поскольку это событие имеет место, когда значение каждого литерала в дизъюнкции равно 0, а значения разным переменным приписываются независимо. Значит, вероятность того, что скобка равна 1, есть $1 - 2^{-k} \geq 1/2$, и математическое ожидание $\mathbf{E} Z_i \geq 1/2$. Отсюда математическое ожидание числа выполненных скобок (равных 1) равно $\mathbf{E} \sum_{i=1}^m Z_i = \sum_{i=1}^m \mathbf{E} Z_i \geq m/2$. Это означает, что есть приписывание значений переменным, при котором $\sum_{i=1}^m Z_i \geq m/2$. \square

Эта теорема дает по существу приближенный вероятностный алгоритм.

Определение 4.4.1. Вероятностный приближенный алгоритм A гарантирует точность C , если для всех входов I

$$1 \geq \frac{\mathbf{E} m_A(I)}{m_0(I)} \geq C > 0,$$

где $m_0(I)$ — оптимум, $m_A(I)$ — значение, найденное алгоритмом, и решается задача максимизации.

Отличие приближенных алгоритмов от детерминированных состоит в рассмотрении математического ожидания времени работы.

Описанный в теореме 17 вероятностный алгоритм дает точность $1/2$ для MAX-SAT. Теперь опишем другой вероятностный алгоритм, гарантирующий для MAX-SAT точность $3/4$.

Для этого мы переформулируем MAX-SAT в задачу целочисленного линейного программирования (ЦЛП). Каждой скобке (элементарной дизъюнкции) C_j поставим в соответствие булеву переменную $z_j \in \{0, 1\}$, которая равна 1, если скобка C_j выполнена, каждой входной переменной x_i сопоставляем переменную y_i , которая равна 1, если $x_i = 1$, и равна 0 в противном случае.

Обозначим C_j^+ индексы переменных в скобке C_j , которые входят в нее без отрицания, а через C_j^- — множество индексов переменных, которые входят в скобку с отрицанием.

Тогда MAX-SAT допускает следующую формулировку в виде задачи ЦЛП:

$$\begin{aligned} & \sum_{j=1}^m z_j \rightarrow \max \\ & \sum_{i \in C_j^+} y_i + \sum_{i \in C_j^-} (1 - y_i) \geq z_j \quad \forall j. \\ & y_i, z_j \in \{0, 1\} \quad \forall i, j. \end{aligned} \tag{4.1}$$

Упражнение 4.4.1. Докажите эквивалентность ЦЛП (4.1) и задачи 20 «MAX-SAT».

Рассмотрим и решим линейную релаксацию целочисленной программы (4.1).

$$\begin{aligned}
 & \sum_{j=1}^m z_j \rightarrow \max \\
 & \sum_{i \in C_j^+} y_i + \sum_{i \in C_j^-} (1 - y_i) \geq z_j \quad \forall j. \\
 & y_i, z_j \in [0, 1] \quad \forall i, j.
 \end{aligned} \tag{4.2}$$

Пусть \hat{y}_i, \hat{z}_j — решение линейной релаксации (4.2). Ясно, что $\sum_{j=1}^m \hat{z}_j$ является верхней оценкой числа выполненных скобок для данной КНФ.

Рассмотрим теперь вероятностный алгоритм 33 (так называемое вероятностное округление), более интересный, чем тривиальное равновероятностное округление каждой переменной в 1 и 0. При вероятностном округлении в алгоритме 33 каждая переменная y_i независимо принимает значение 1 с вероятностью \hat{y}_i (и 0 с вероятностью $1 - \hat{y}_i$).

Для целого k положим $\beta_k = 1 - (1 - 1/k)^k$.

Лемма 22. Пусть в скобке C_j имеется k литералов. Вероятность того, что она выполнена при вероятностном округлении, не менее $\beta_k \hat{z}_j$.

Доказательство. Поскольку мы рассматриваем отдельно взятую скобку, без ограничения общности можно предположить, что все переменные входят в нее без отрицаний (докажите этот факт в качестве упражнения!). Пусть эта скобка имеет вид: $x_1 \vee \dots \vee x_k$. Из ограничений линейной релаксации (4.2) следует, что

$$\hat{y}_1 + \dots + \hat{y}_k \geq \hat{z}_j.$$

Алгоритм 33 Приближенный вероятностный алгоритм для задачи 20 (MAX-SAT) на основе линейной релаксации

Вход: Формулировка задачи 20 «MAX-SAT» в виде (4.1)

$\hat{y} \leftarrow$ решения линейной релаксации (4.2)

for all $i \in \{1..m\}$ **do**

$y_i \leftarrow 0$

if $\text{random}(0..1) \leq \hat{y}_i$ **then**

$y_i \leftarrow 1$ { $y_i \leftarrow 1$ с вероятностью \hat{y}_i }

end if

end for

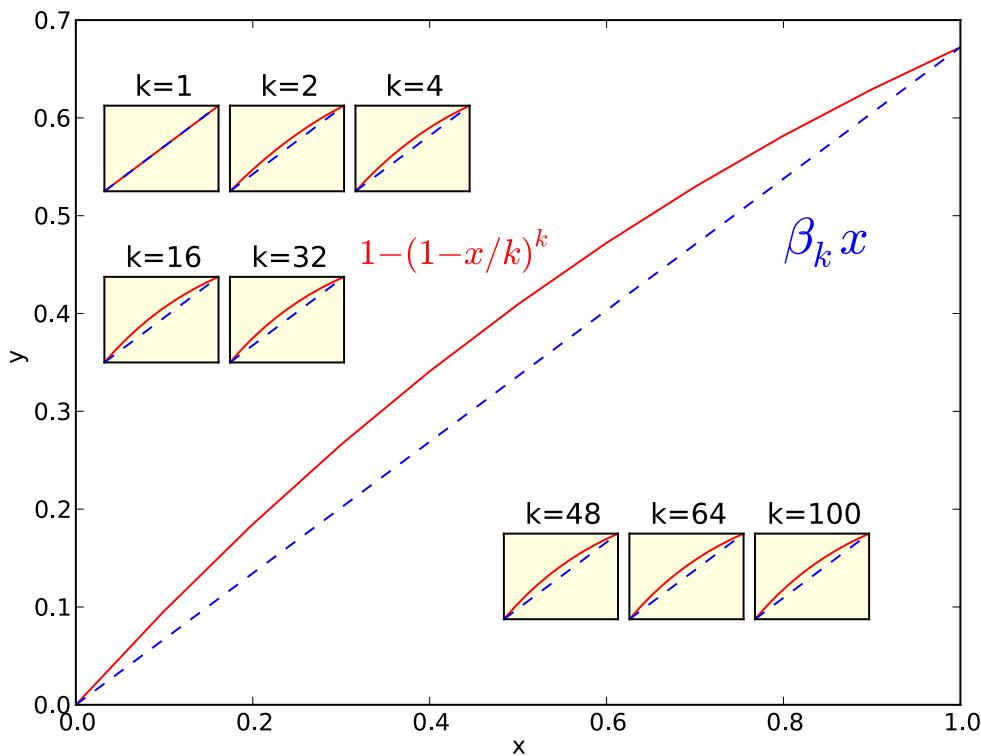
Выход: (y_1, \dots, y_m) — приближенное решение (4.1), со средней точностью $(1 - 1/e)$.

Скобка C_j остается невыполненной при вероятностном округлении, только если каждая из переменных y_i округляется в 0. Поскольку каждая переменная округляется независимо, это происходит с вероятностью $\prod_{i=1}^k (1 - \hat{y}_i)$. Остается только показать, что

$$1 - \prod_{i=1}^k (1 - \hat{y}_i) \geq \beta_k \hat{z}_j.$$

Выражение в левой части достигает минимума при $\hat{y}_i = \hat{z}_j/k$ для всех i . Остается показать, что $1 - (1 - z/k)^k \geq \beta_k z$ для всех положительных целых k и $0 \leq z \leq 1$.

Поскольку $f(x) = 1 - (1 - x/k)^k$ — вогнутая функция (см. рис. 4.4), для доказательства того, что она не меньше линейной функции на отрезке, достаточно проверить это нестрогое неравенство на концах этого отрезка, т. е. в точках $x = 0$ и $x = 1$ (проделайте это в качестве упражнения). \square

Рис. 4.4: График функции $1 - (1 - x/k)^k$ при различных k

Используя тот факт, что $\beta_k \geq 1 - 1/e$ для всех положительных целых k , получаем, что справедлива следующая

Теорема 18. Для произвольного входа задачи MAX-SAT (произвольной КНФ) среднее число скобок, выполненных при вероятностном округлении, не меньше $(1 - 1/e)$ от максимально возможного числа выполненных скобок.

А теперь мы опишем простую, но общую идею, которая позволит получить приближенный вероятностный алгоритм, имеющий точность $3/4$.

А идея такова: на данном входе запускаем два алгоритма и выбираем из решений лучшее. В качестве двух алгоритмов рассматриваем два алгоритма, описанных выше:

1. округление каждой переменной независимо в 0 или 1 с вероятностью $1/2$;
2. вероятностное округление решения линейной релаксации соответствующей целочисленной программы (алгоритм 33 «вероятностный MAX-SAT»).

Для входной КНФ ϕ обозначим через $n_1 = n_1(\phi)$ и $n_2 = n_2(\phi)$ число выполненных скобок для первого и второго алгоритма соответственно.

Теорема 19.

$$\mathbf{E} \max\{n_1, n_2\} \geq \frac{3}{4} \sum_j \hat{z}_j.$$

Доказательство. Поскольку для неотрицательных n_1 и n_2 выполняется $\max\{n_1, n_2\} \geq (n_1 + n_2)/2$, достаточно показать, что $\mathbf{E}(n_1 + n_2)/2 \geq \frac{3}{4} \sum_j \hat{z}_j$.

Пусть S^k обозначает множество скобок, содержащих ровно k литералов, тогда:

$$n_1 = \sum_k \sum_{C_j \in S^k} (1 - 2^{-k}) \geq \sum_k \sum_{C_j \in S^k} (1 - 2^{-k}) \hat{z}_j.$$

По лемме 22 имеем

$$\mathbf{E} n_2 \geq \sum_k \sum_{C_j \in S^k} \beta_k \hat{z}_j.$$

Следовательно,

$$\mathbf{E} \frac{n_1 + n_2}{2} \geq \sum_k \sum_{C_j \in S^k} \frac{(1 - 2^{-k}) + \beta_k}{2} \hat{z}_j.$$

Простое вычисление показывает, что $(1 - 2^{-k}) + \beta_k \geq 3/2$ для всех натуральных k и, значит,

$$\mathbf{E} \frac{n_1 + n_2}{2} \geq \frac{3}{4} \sum_k \sum_{C_j \in S^k} \hat{z}_j = \frac{3}{4} \sum_j \hat{z}_j.$$

□

4.4.2 Максимальный разрез в графе



Задачи полуопределенного и векторного программирования. Использование эффективных алгоритмов для решений этих задач в вероятностном алгоритме решения задачи о максимальном разрезе в графе. Раздел основан на статье [GW95].

Определение 4.4.2. Матрица $X \in R^{n \times n}$ является **положительно полуопределенной** если

$$\forall a \in R^n, \quad a^\top X a \geq 0.$$

Обозначение: $X \succcurlyeq 0$.

Для симметрической $X \in R^{n \times n}$ следующее эквивалентно:

- $X \succcurlyeq 0$;
- X имеет неотрицательные собственные значения;
- $X = V^\top V$ для некоторого $V \in R^{m \times n}$, где $m \leq n$.

Задача 21. «Полуопределенное программирование»².

$$\begin{aligned} \sum_{i,j} c_{ij} x_{ij} &\rightarrow \max(\min) \\ \forall k \sum_{i,j} a_{ijk} x_{ij} &= b_k, \\ X = (x_{ij}) &\succcurlyeq 0, \\ \forall i, j \quad x_{ij} &= x_{ji}. \end{aligned}$$

²В англоязычной литературе SDP, semidefinite programming.

Хотя задача 21 «SDP», несмотря на свою схожесть с линейным программированием, к нему не сводится, для нее существуют эффективные полиномиальные алгоритмы (модификации метода внутренней точки для ЛП), находящие приближенное решение с некоторой аддитивной ошибкой ϵ и временем, ограниченным полиномом по длине входа и $O(\log(\frac{1}{\epsilon}))$.

Заметим, что, так как решение задачи 21 «SDP» может быть иррациональным числом, от численных методов точного (рационального) решения ждать и невозможно, хотя продолжаются попытки построить эффективный алгоритм нахождения точного решения алгебраическими методами.

Далее нам также пригодится эквивалентная формулировка задачи 21 «SDP» в виде задачи 22:

Задача 22. «Векторное программирование»³.

$$\begin{aligned} \sum_{i,j} c_{ij} (\bar{v}_i \cdot \bar{v}_j) &\rightarrow \max(\min) \\ \forall k \sum_{i,j} a_{ijk} (\bar{v}_i \cdot \bar{v}_j) &= b_k, \\ \forall i \bar{v}_i &\in R^n. \end{aligned}$$

Эквивалентность задач 21 «SDP» и 22 «VP» следует из факторизации положительно полуопределенной матрицы X в виде $X = V^T V$, т. е. $x_{ij} = \bar{v}_i \cdot \bar{v}_j$, где \bar{v}_i и \bar{v}_j — соответствующие колонки матрицы V .

Преобразование решения задачи 22 «VP» в решение задачи 21 «SDP» тривиально (одно матричное умножение), обратное не совсем — требуется разложение Холецкого⁴, и это преобразование неоднозначно, но ничего принципиально сложного в этом нет — это классическая задача линейной алгебры.

³В англоязычной литературе VP, vector programming.

⁴Cholesky factorization или Cholesky decomposition.

Вероятностное округление при нахождении аппроксимации максимального разреза

Определение 4.4.3. Пусть есть неориентированный граф $G = (V, E)$. **Разрезом** (сечением, *cut*) называется разбиение множества вершин V на непересекающиеся множества S и T . т. е. $V = S \cup T$ и $S \cap T = \emptyset$.

Определение 4.4.4. Для неориентированного графа $G = (V, E)$ и разреза (S, T) ребро $e = (v, t)$ считается **пересекающим разрез**, если $v \in S$, а $t \in T$.

Определение 4.4.5. Для графа $G = (V, E)$ **размером** разреза (S, T) считается число ребер, пересекающих этот разрез.

Если граф — взвешенный, т. е. каждому ребру $e \in E$ соответствует некоторый вес w_e , то размером разреза (S, T) считается сумма весов ребер пересекающих этот разрез:

$$R(S, T) = \sum_{e=(v,t) \in E: v \in S, t \in T} w_e.$$

Задача 23. «Максимальный разрез/MAX-CUT».

Для взвешенного неориентированного графа $G = (V, E)$ с весами $w_e > 0$ найти разрез (S, T) с максимальным весом $R(S, T)$.

Упражнение 4.4.2. Докажите, что для простого, невзвешенного графа в задаче 23 «MAX-CUT» можно применить простую стратегию, дающую вероятностный 0.5-приближенный алгоритм: для каждой вершины с вероятностью $1/2$ отнести ее к множеству S и с вероятностью $1/2$ — к множеству T .

Упражнение 4.4.3. Студент предлагает для задачи 23 «MAX-CUT» приближенный алгоритм с точностью $\frac{1}{2}$: положить первую вершину в одну часть, последнюю — в другую, затем по-очереди добавлять оставшиеся вершины, к которым у этой вершины меньше ребер-связей.

Прав ли студент?

Наша цель — построить алгоритм с лучшими оценками точности приближения. Для этого сформулируем задачу 23 «MAX-CUT» как задачу целочисленного программирования.

Задача 24. «MAX-CUT(ЦП)»

$G = (V, E)$ — входной граф, $|V| = n$;

$W = (w_{ij})$ — веса ребер, $n \times n$ матрица. Для отсутствующего между v_i и v_j ребра — $w_{ij} = 0$;

y_i — принадлежность вершины части разреза:

$v_i \in S \rightarrow y_i = 1, v_i \in T \rightarrow y_i = -1$.

Ребро $(v_i, v_j) \in (S, T) \Leftrightarrow y_i y_j = -1$.

$R(S, T)$ — вес разреза (S, T) . $R(S, T) = \sum_{i < j} \frac{1 - y_i y_j}{2} w_{ij}$.

Задача целочисленного квадратичного программирования:

$$\begin{aligned} Z_{ЦП} = \sum_{i < j} \frac{1 - y_i y_j}{2} w_{ij} \rightarrow \max \\ \forall i \quad y_i \in \{-1, 1\}. \end{aligned}$$

Видно, что постановка задачи 24 «MAX-CUT(ЦП)» внешне похожа на задачу 22 «VP», однако целочисленные ограничения в задаче 24 «MAX-CUT(ЦП)» делают ее существенно сложнее для решения. Рассмотрим следующую релаксацию задачи 24 «MAX-CUT(ЦП)».

Задача 25. «MAX-CUT(VP)»

$$\begin{aligned} Z_{VP} = \sum_{i < j} \frac{1 - \bar{v}_i \cdot \bar{v}_j}{2} w_{ij} \rightarrow \max \\ \forall i \quad \bar{v}_i \cdot \bar{v}_i = 1, \\ \forall i \quad \bar{v}_i \in R^n. \end{aligned}$$

Эту задачу мы уже можем решать эффективно (см. выше), осталось разобраться, можно ли использовать решение задачи 25 «MAX-CUT(VP)» для нахождения решения (возможно приближенного) задачи 24 «MAX-CUT(ЦП)».

Сначала заметим, что для оптимумов задач 24 «MAX-CUT(ЦП)» и 25 «MAX-CUT(VP)» выполняется

$$Z_{ЦП}^* \leq Z_{VP}^*.$$

Этот факт следует из того, что задача 25 «MAX-CUT(VP)» содержит задачу 24 «MAX-CUT(ЦП)» как частный случай.

Используя полученное решение задачи 25 «MAX-CUT(VP)» для вероятностного округления, получаем алгоритм 34 (концептуально аналогичный рассмотренному ранее алгоритму 33 «вероятностный MAX-SAT»).

Нам понадобится небольшой технический факт.

Лемма 23.

$$\min_{0 < \theta \leq \pi} \frac{2\theta}{\pi(1 - \cos \theta)} \geq 0.878.$$

Доказательство. Рассмотрим график нашей функции на различных интервалах (Рис. 4.5). Функция имеет сингулярности в точках 0 и 2π , а на интервале $(0; 2\pi)$ функция выпуклая и имеет один минимум. Судя по построенным графикам, на интервале $(0; 2\pi)$ функция не меньше 0.878 .

Чтобы убедиться в этом, найдем минимум (по равенству нулю производной) и вычислим значение функции в точке минимума. Воспользуемся системой компьютерной алгебры Maxima: возьмем производную и с помощью метода Ньютона найдем ее единственный нуль в интересующем нас интервале.

```
< load("newton");
< y:2*x/%pi/(1-cos(x));
< x0:newton(diff(y,x),3);
> 2.331122370414422B0
< y(x0),numer;
> 0.87856720578485
```

Таким образом, можно даже утверждать, что

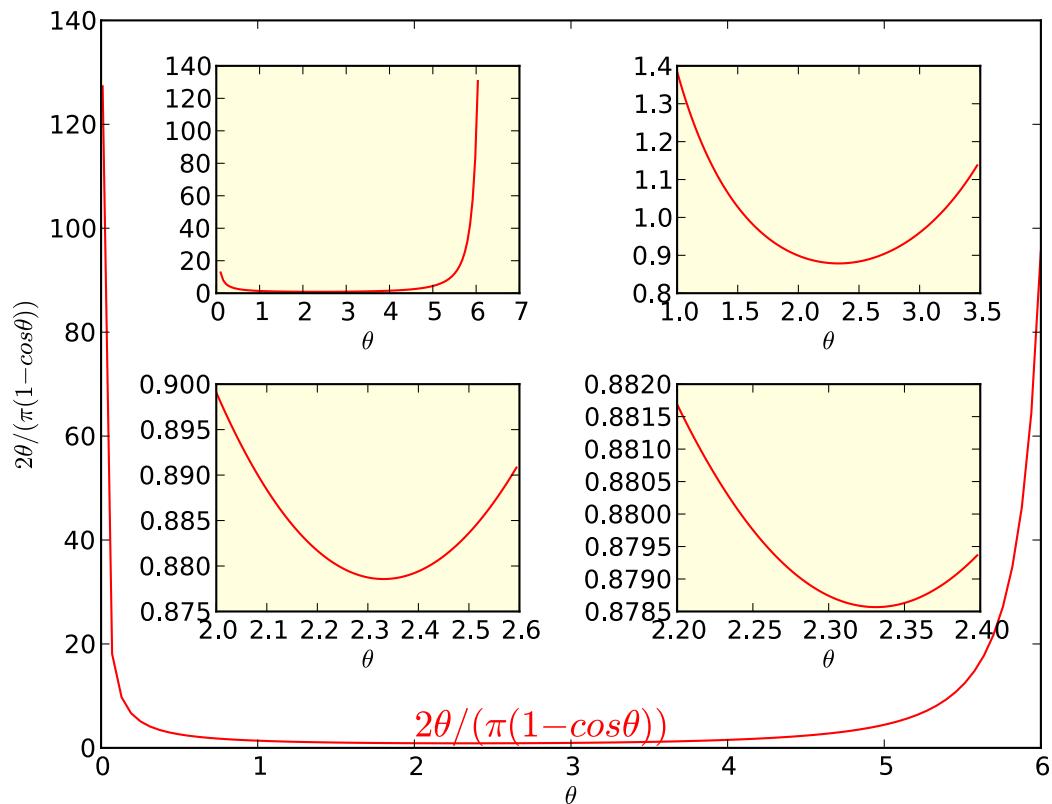
$$\min_{0 < \theta \leq \pi} \frac{2\theta}{\pi(1 - \cos \theta)} > 0.8785672057848.$$

□

Теперь оценим качество алгоритма.

Теорема 20. Пусть (S^*, T^*) — оптимальный разрез для задачи 23 «MAX-CUT», тогда для математического ожидания величины разреза (S', T') , полученного вероятностным алгоритмом 34 «SDP-округление MAX-CUT», выполняется

$$\mathbf{E}[R(S', T')] \geq 0.878 \cdot R(S^*, T^*).$$

Рис. 4.5: График функции $\frac{2\theta}{\pi(1-\cos\theta)}$

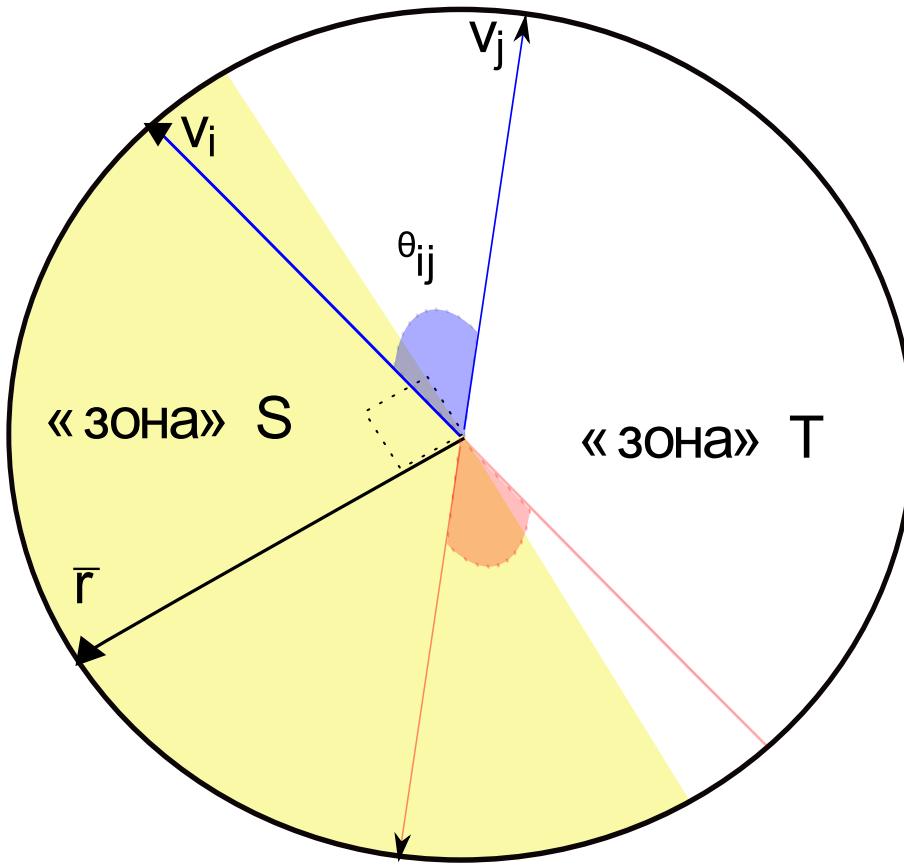


Рис. 4.6: Вектора в вероятностном округлении «MAX-CUT»

Алгоритм 34 «SDP-округление MAX-CUT»

Вход: Формулировка задачи 23 «MAX-CUT» в виде задачи 24 «MAX-CUT(ЦП)»

$(v_1, \dots, v_n) \leftarrow$ решения релаксации 25 «MAX-CUT(VP)»

случайно выбираем \bar{r} из равномерного распределения векторов единичной длины

$S \leftarrow T \leftarrow \emptyset$

for all $i \in \{1..n\}$ **do**

if $\bar{v}_i \cdot \bar{r} \geq 0$ **then**

$S \leftarrow S \cup \{i\}$

else

$T \leftarrow T \cup \{i\}$

end if

end for

Выход: (S, T) — приближенное решение (23).

Доказательство. Возьмем некоторые i и j и соответствующие им векторы решения релаксации v_i и v_j . Затем оценим вероятность того, что при вероятностном округлении соответствующие вершины попадут в разные части разреза. Получим

$$\mathbb{P}(y_i \neq y_j) = \mathbb{P}(y_i y_j = -1) = \frac{\theta_{ij}}{\pi},$$

где θ_{ij} — угол между векторами v_i и v_j .

Отсюда, используя определение математического ожидания:

$$\mathbf{E} \sum_{i < j} \frac{1 - y_i y_j}{2} w_{ij} = \sum_{i < j} \frac{2 \cdot \mathbb{P}(y_i \neq y_j)}{2} w_{ij} = \sum_{i < j} \frac{\theta_{ij}}{\pi} w_{ij}.$$

С другой стороны, значение решения релаксации 25 «MAX-CUT(VP)» тоже можно выразить через углы θ_{ij} :

$$Z_{VP} = \sum_{i < j} \frac{1 - \bar{v}_i \cdot \bar{v}_j}{2} w_{ij} = \sum_{i < j} \frac{1 - \cos \theta_{ij}}{2} w_{ij}.$$

Теперь оценим минимальное качество решения, используя лемму 23:

$$\frac{\mathbf{E}[R(S', T')]}{R(S^*, T^*)} \geq \frac{\mathbf{E}[R(S', T')]}{Z_{VP}} = \frac{\sum_{i < j} \frac{\theta_{ij}}{\pi} w_{ij}}{\sum_{i < j} \frac{1 - \cos \theta_{ij}}{2} w_{ij}} \geq \min_{0 < \theta \leq \pi} \frac{2\theta}{\pi(1 - \cos \theta)} \geq 0.878.$$

□

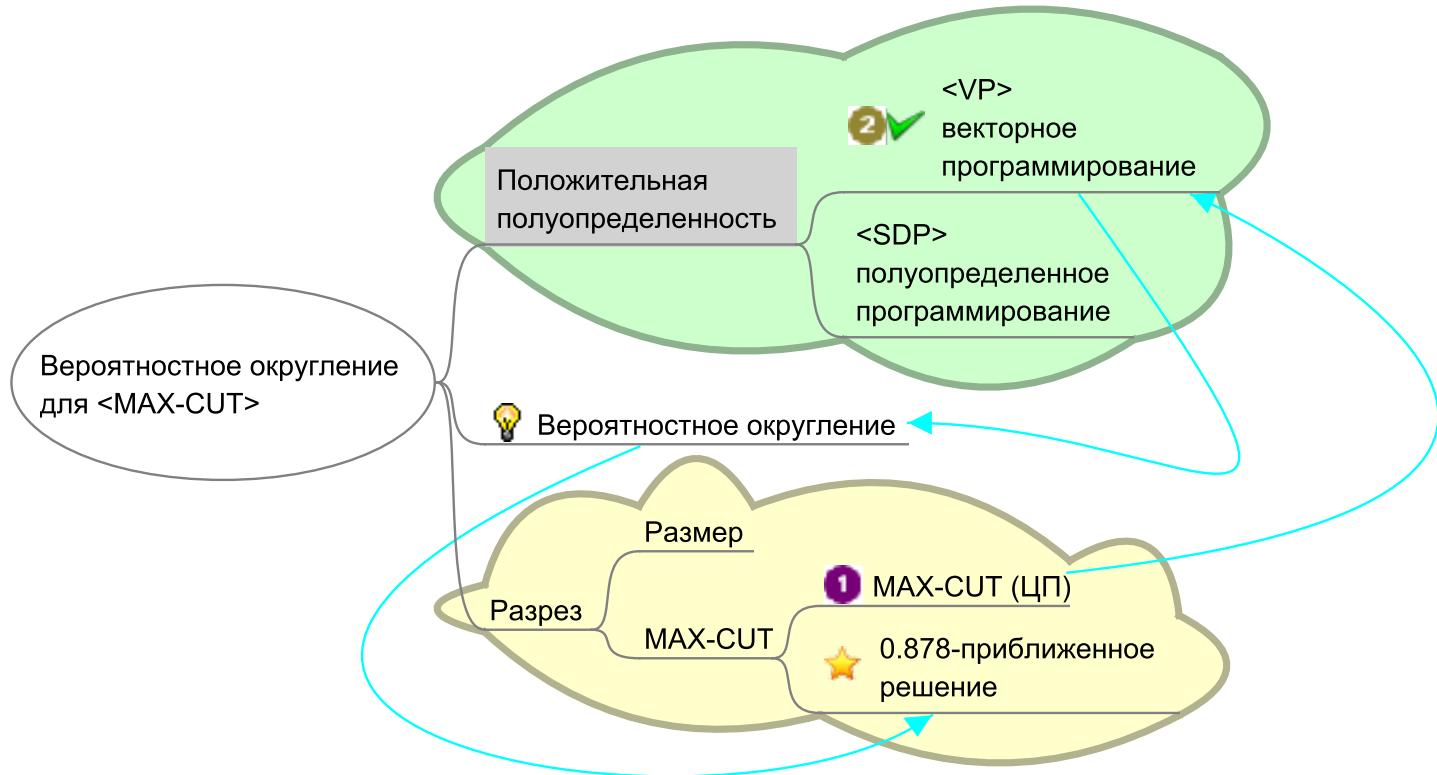


Рис. 4.7: Карта-памятка раздела 4.4.2

Глава 5

Методы дерандомизации

5.1 Метод условных вероятностей

Оказывается, в некоторых случаях вероятностные алгоритмы могут быть «дерандомизированы», т. е. конвертированы в детерминированные алгоритмы. Один из общих методов, позволяющих сделать это, называется методом условных вероятностей.

Проблемы, связанные с дерандомизацией вероятностных алгоритмов при построении хороших целочисленных решений, в настоящее время находятся в центре внимания многих исследователей (см. [MR95]). После работы [Rag88], в которой вероятностным методом было доказано существование хороших целочисленных решений в задаче аппроксимации на решетке, появилось много работ, в которых та же техника использовалась для других задач. Эта техника получила название метода *условных вероятностей*.

При этом подходе эффективный детерминированный алгоритм нахождения искомого целочисленно-

го вектора можно построить путем аппроксимации исходной задачи некоторым функционалом (обычно связанным с оценками вероятностей некоторых событий), и затем использовать покоординатное построение требуемого решения.

Опишем этот подход на примере следующей задачи: имеется величина $X(x)$, где в булевом векторе $x = (x_1, \dots, x_n)$ компоненты являются независимыми случайными величинами, причем $P\{x_i = 1\} = p_i$, $P\{x_i = 0\} = 1 - p_i$.

Так, в задаче 20 «MAX-SAT» $X(x_1, \dots, x_n)$ равно числу невыполненных скобок в КНФ при вероятностном округлении.

Требуется найти любой булев вектор \hat{x} , для которого выполнено неравенство

$$X(\hat{x}) \leq \mathbf{E} X. \quad (5.1)$$

Обозначим через $X(x| x_1 = d_1, \dots, x_k = d_k)$ новую случайную величину, которая получена из X фиксированием значений первых k булевых переменных.

Рассмотрим покомпонентную стратегию определения искомого вектора \hat{x} . Для определения его первой компоненты в $x = (x_1, \dots, x_n)$ вычисляем значения $f_0 \leftarrow \mathbf{E} X(x| x_1 = 0)$ и $f_1 \leftarrow \mathbf{E} X(x| x_1 = 1)$. Если $f_0 < f_1$, полагаем $x_1 = 0$, иначе полагаем $x_1 = 1$. При определенной таким образом первой компоненте (обозначим ее d_1) вычисляем значения функционала $f_0 \leftarrow \mathbf{E} X(x| x_1 = d_1, x_2 = 0)$ и $f_1 \leftarrow \mathbf{E} X(x| x_1 = d_1, x_2 = 1)$.

Если $f_0 < f_1$, полагаем $x_2 = 0$, иначе полагаем $x_2 = 1$.

Фиксируем вторую координату (обозначая ее d_2) и продолжаем описанный процесс до тех пор, пока не определится последняя компонента решения (см. рис. 5.1).

Почему найденный вектор будет удовлетворять (5.1)? Рассмотрим первый шаг алгоритма. Имеем

$$\begin{aligned}
 \mathbf{E} X &= \mathsf{P}\{x_1 = 1\} \mathbf{E} X(x \mid x_1 = 1) + \mathsf{P}\{x_1 = 0\} \mathbf{E} X(x \mid x_1 = 0) = \\
 &= p_1 \mathbf{E} X(x \mid x_1 = 1) + (1 - p_1) \mathbf{E} X(x \mid x_1 = 0) \geq \\
 &\geq p_1 \mathbf{E} X(x \mid x_1 = d_1) + (1 - p_1) \mathbf{E} X(x \mid x_1 = d_1) = \\
 &= (p_1 + 1 - p_1) \mathbf{E} X(x \mid x_1 = d_1) = \mathbf{E} X(x \mid x_1 = d_1).
 \end{aligned}$$

Продолжая эту цепочку неравенств для каждого шага, получаем на последнем n -м шаге

$$\mathbf{E} X \geq \mathbf{E} X(x \mid x_1 = d_1, \dots, x_n = d_n).$$

Но

$$\mathbf{E} X(x \mid x_1 = d_1, \dots, x_n = d_n) = X(d_1, \dots, d_n),$$

и для построенного вектора $\hat{x} = (d_1, \dots, d_n)$ выполнено неравенство (5.1).

Как оценить число шагов описанной процедуры? На каждой итерации вычисляются два условных математических ожидания (f_0 и f_1) и затем находится минимум этих величин. Если это делается за время $O(L)$, то при общем числе итераций n получим оценку $O(nL)$. Таким образом, изложенный общий метод позволяет осуществлять «дерандомизацию», если у нас есть эффективный алгоритм вычисления условных математических ожиданий (или условных вероятностей).

Теперь используем описанный выше метод условных вероятностей для задачи 20 «MAX-SAT».

Математическое ожидание X равно сумме вероятностей невыполнения скобок:

$$\mathbf{E} X = \sum_{j=1}^m P_j, \tag{5.2}$$

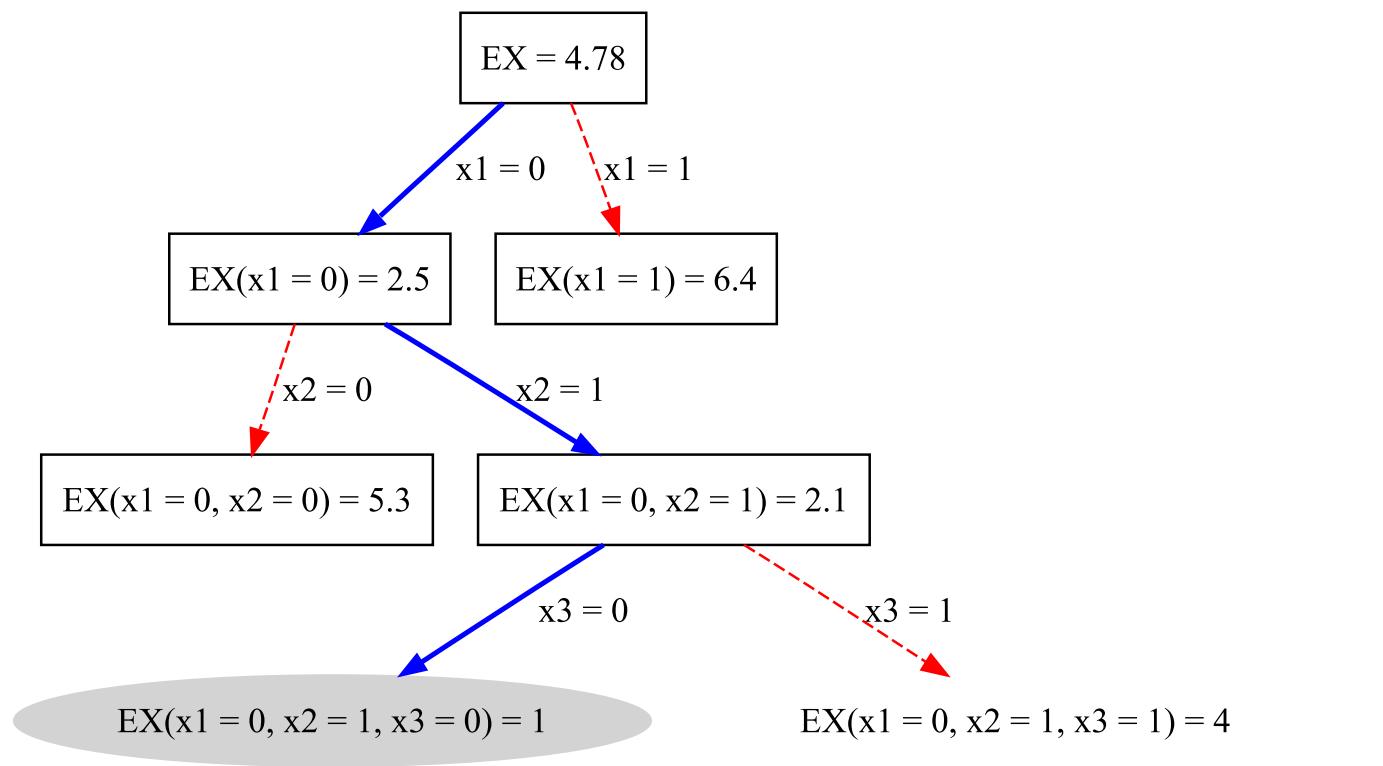


Рис. 5.1: Дерандомизация на основе минимизации оценок математического ожидания

Алгоритм 35 Дерандомизация вероятностного алгоритма 33 «вероятностный MAX-SAT» по методу условных вероятностей

Вход: Формулировка задачи 20 «MAX-SAT» в виде (4.1).

Выход: (x_1, \dots, x_m) — приближенное решение (4.1).

$(p_1, \dots, p_n) \leftarrow$ решения линейной релаксации (4.2).

for all $i \in \{1..n\}$ **do**

$f_0 = \mathbf{E} X(x | x_i = 0)$ {Вычисляется через (5.4)}

$f_1 = \mathbf{E} X(x | x_i = 1)$ {и (5.2)}

if $f_0 < f_1$ **then**

$x_i \leftarrow 0$

else

$x_i \leftarrow 1$

end if

end for

return \mathbf{x}

где вероятность невыполнения j -й дизъюнкции есть

$$P_j = \mathbf{P} \left\{ \sum_{i \in C_j^+} x_i + \sum_{i \in C_j^-} (1 - x_i) = 0 \right\}.$$

Для полученного вектора (d_1, \dots, d_n) выполнено неравенство

$$X(x_1 = d_1, \dots, x_n = d_n) \leq \mathbf{E} X. \quad (5.3)$$

Из теоремы 18 следует, что $\mathbf{E} X \leq (1/e)m$, а из (5.3) вытекает, что $X(x_1 = d_1, \dots, x_n = d_n) \leq (1/e)m$.

Таким образом, мы находим допустимый 0–1 вектор $\mathbf{x} = (x_1, \dots, x_n)$ с гарантированной верхней оценкой для целевой функции.

Важный вопрос заключается в том, как эффективно вычислять условные математические ожидания. Предположим, значения первых k переменных уже определены и I_0 — множество индексов переменных, значения которых равны 0, а I_1 — множество индексов переменных, значения которых равны 1.

Нетрудно проверить, что если $I_0 \cap C_j^- \neq \emptyset$ или $I_1 \cap C_j^+ \neq \emptyset$, то $P_j = 0$. В противном случае

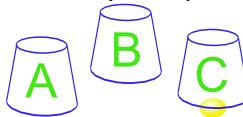
$$P_j = \prod_{i \in C_j^+ \setminus I_0} (1 - p_i) \cdot \prod_{i \in C_j^- \setminus I_1} p_i. \quad (5.4)$$

Запишем этот алгоритм более формально, в виде алгоритма 35 «MAX-SAT дерандомизация».

Упражнение 5.1.1. Покажите, что можно организовать вычисление f_0 и f_1 таким образом, что сложность алгоритма 35 «MAX-SAT дерандомизация» (кроме решения линейной релаксации) будет $O(mn)$.

Упражнение 5.1.2. Честный попутчик в поезде предлагает вам сыграть в следующую игру, вариант классического «наперстка». т. е. есть три наперстка, шарик, ваша задача обнаружить шарик — тогда вы выигрываете, иначе — выигрывает сдающий.

Каждый раз на кон, вы и сдающий, ставите по 50 рублей. Сдающий тасует три наперстка, и прячет под одним из них шарик. Затем он предлагает вам выбрать наперсток. После того, как вы выбрали наперсток (пусть это будет наперсток «A»), вы можете открыть его, либо, заплатив еще 10 рублей, потребовать от сдающего, открыть пустой наперсток из двух оставшихся (пусть открытый будет «B»), после чего выбрать один из двух закрытых наперстков (т. е. «A» или «C»).



Заметим, что сдающий и предметы честные (никаких «исчезающих» шариков и прочего мошенничества, наперсток для шарика выбирается совершенно случайно).

Какая стратегия оптимальна?

1. Не играть. Выигрыш — 0.
2. Выбрать наперсток «A» и открыть его.
3. Выбрать наперсток «A», «купить» открытие пустого наперстка «B», но выбрать наперсток «A».
4. Выбрать наперсток «A», «купить» открытие пустого наперстка «B», и выбрать наперсток «C».

Дорога длинная, играть можно много раз, правильная стратегия может привести к существенному обогащению, неправильная — к разорению...

Обоснуйте. Подсчитайте матожидание выигрыша для каждой из стратегий.



Рис. 5.2: Карта-памятка раздела 5.1.0

5.2 Метод малых вероятностных пространств

В разделе 4.3.1 мы уже рассматривали параллельный вероятностный алгоритм **Parallel MIS**, для поиска максимального по включению независимого множества (см. определение 4.3.2 «MAX-IND-SET»), причем у этого алгоритма средним числом итераций было логарифмическим от длины входа.

Идея дерандомизации этого алгоритма заключается в том, чтобы показать, что вероятностный анализ работает аналогичным образом, даже если пометки вершин делаются не полностью независимо, а попарно независимо. Отметим, единственное место в анализе алгоритма, где использовалась независимость пометок, была лемма 17.

Справедлива следующая

Лемма 24. *Если в алгоритме Parallel MIS случайные пометки вершин делаются попарно независимо, то вероятность того, что хорошая вершина принадлежит множеству $S \cup \Gamma(S)$, не меньше, чем $1/24$.*

Ключевое преимущество попарной независимости состоит в том, что только $O(\log n)$ случайных бит достаточно для порождения всех точек соответствующего вероятностного пространства.

Число точек в таком пространстве есть $O(n^c)$ и может быть просмотрено полным перебором за полиномиальное время.

Доказательство леммы. Единственное место, где использовалась полная независимость пометок вершин, была нижняя оценка вероятности, что хорошая вершина становится помеченной. Была использована следующая оценка.

Пусть X_i , $1 \leq i \leq n - \{0, 1\}$ —случайные величины и $p_i = P(X_i = 1)$. Если X_i независимы в совокупности, то

$$\mathbb{P} \left(\sum_1^n X_i > 0 \right) \geq 1 - \prod_1^n (1 - p_i).$$

Мы заменим эту оценку соответствующей оценкой для попарно независимых случайных величин.

Утверждение. Пусть X_i , $1 \leq i \leq n$ — $\{0, 1\}$ -случайные величины и $p_i = \mathbb{P}(X_i = 1)$. Если X_i попарно независимы, то

$$\mathbb{P} \left(\sum_1^n X_i > 0 \right) \geq \frac{1}{2} \min \left\{ \frac{1}{2}, \sum_1^n p_i \right\}.$$

Доказательство. Предположим, что $\sum p_i \leq 1/2$.

Обозначим через Y_i событие $X_i = 1$. Имеем по формуле включений-исключений:

$$\begin{aligned} \mathbb{P}(\cup_1^n Y_i) &\geq \sum_i \mathbb{P}(Y_i) - \frac{1}{2} \sum_{i,j} \mathbb{P}(Y_i \wedge Y_j) = \sum_i p_i - \frac{1}{2} \sum_{i,j} p_i p_j = \\ &= \sum_i p_i - \frac{1}{2} \left(\sum_i p_i \right)^2 \geq \sum_i p_i \left(1 - \frac{1}{2} \sum_i p_i \right) \geq \frac{1}{2} \sum_i p_i. \end{aligned}$$

Если $\sum_i p_i > 1/2$, то ограничим индексы суммирования по подмножеству $S \subseteq [n]$, такому, что $1/2 \leq \sum_i p_i \leq 1$, и повторим то же доказательство.

Завершим теперь доказательство леммы.

Напомним, что вершина $v \in V$ называется *хорошей*, если она имеет не менее $d(v)/3$ соседних вершин степени не более $d(v)$. В противном случае вершина называется *плохой*.

Покажем, что если v — хорошая вершина, то вероятность того, что найдется помеченная вершина из $\Gamma(v)$, не меньше $1/12$.

Обозначим эту вероятность через $P(\Gamma(v)_{marked})$.

Обозначим $\Gamma^*(v) = \{w \in \Gamma(v) \mid d(w) \leq d(v)\}$. По определению хорошей вершины $|\Gamma^*(v)| \geq \frac{1}{3}d(v)$. Используя доказанную выше оценку для попарно независимых случайных величин, имеем

$$\begin{aligned} P(\Gamma(v)_{marked}) &\geq \frac{1}{2} \min\left\{\frac{1}{2}, \sum_{w \in \Gamma(v)} \frac{1}{2d(w)}\right\} \geq \\ &\geq \frac{1}{2} \min\left\{\frac{1}{2}, \sum_{w \in \Gamma^*(v)} \frac{1}{2d(w)}\right\} \geq \frac{1}{2} \min\left\{\frac{1}{2}, \sum_{w \in \Gamma^*(v)} \frac{1}{2d(v)}\right\} \geq \\ &\geq \frac{1}{2} \min\left\{\frac{1}{2}, \frac{1}{3}d(v) \frac{1}{2d(v)}\right\} \geq \frac{1}{2} \min\left\{\frac{1}{2}, \frac{1}{6}\right\} = \frac{1}{12}. \end{aligned}$$

Используя лемму 18 из анализа вероятностного алгоритма Луби (*В течение каждой итерации, если вершина помечена, то она выбирается в S с вероятностью не менее $1/2$*) и учитывая тот факт, что в ее доказательстве использовалась только попарная независимость, получаем доказательство нашей леммы.

Попарная независимость для неодинаково распределенных случайных величин. Конструкция

Покажем теперь, как осуществить дерандомизацию путем полного перебора элементарных событий (точек) из вероятностного пространства и вычисления в каждой точке этого пространства. Элементарным событием будет приписывание пометок вершинам графа, задаваемое булевым вектором длины n . Размер построенного пространства будет полиномиальным. Для дерандомизации путем полного перебора

точек из построенного вероятностного пространства достаточно будет вычислить в каждой точке подмножество вершин R , соответствующее шагу 2.1 алгоритма Parallel MIS, затем для каждого ребра в R выбросить концевую вершину меньшей степени (шаг 2.2 алгоритма), подсчитать число N_R ребер, смежных с оставшимся множеством вершин. Затем взять множество R , максимизирующее показатель N_R . Оно обеспечит требуемые оценки на число итераций детерминированного алгоритма.

Итак, для осуществления указанного метода дерандомизации нужна конструкция небольшого вероятностного пространства со свойством попарной независимости соответствующих случайных величин. Опишем эту конструкцию.

Выберем наименьшее простое число p , такое, что $cn \leq p$, где точное значение константы $c > 1$ выберем далее. Для каждой вершины u выберем целое a_u , такое, что $a_u/p \approx 1/2d(u)$ и интервал A_u , $|A_u| = a_u$ в \mathbb{Z}_p . Более точно, положим $a_u = \lceil \frac{p}{2d(u)} \rceil$. Тогда

$$\frac{1}{2d(u)} \leq \frac{a_u}{p} \leq \frac{\frac{p}{2d(u)} + 1}{p} = \frac{1}{2d(u)} + \frac{1}{p} \leq \frac{1}{2d(u)} + \frac{1}{cn} \leq \frac{1}{2d(u)} + \frac{2}{c} \cdot \frac{1}{2d(u)} = \frac{1}{2d(u)} \left(1 + \frac{2}{c}\right).$$

Достаточно выбрать $c = 10$, чтобы для вероятности вершины u быть помеченной на шаге 2.1 алгоритма Parallel MIS было выполнено неравенство

$$\frac{1}{2d(u)} \leq \frac{a_u}{p} \leq \frac{6}{5} \cdot \frac{1}{2d(u)}.$$

Упражнение 5.2.1. Покажите, что при указанном выборе вероятностей пометки вершин анализ алгоритма Parallel MIS, проведенный в разделе 4.3.1, остается в силе с небольшим изменением констант. В частности, лемма 17 справедлива без изменений, в лемме 18 оцениваемая вероятность не меньше $2/5$, в лемме 19 — соответственно не менее $(2/5) \cdot (1 - e^{-1/6})$.

Упражнение 5.2.2. Покажите, что при указанном выборе вероятностей пометки вершин анализ алгоритма Parallel MIS, проведенный в данном разделе выше, остается в силе с небольшим изменением констант. В частности, лемма 17 справедлива без изменений, в лемме 18 оцениваемая вероятность не меньше $2/5$, в лемме в данном разделе — соответственно не менее $(2/5) \cdot (1/12) = \frac{1}{30}$.

Пусть $X(u)$ — случайная величина, полученная путем равномерного выбора $x, y \in \mathbf{Z}_p$ следующим образом: $X(u) = 1$, если $xu + y \in A_u$, и нулю в противном случае.

Это значит, что $\mathsf{P}(X(u) = 1) = a_u/p$, поскольку

$$\mathsf{P}_{x,y}(X(u) = 1) = \mathsf{P}(\exists a \in A_u : xu + y = a) = \sum_{a \in A_u} \mathsf{P}(x = (a - y)u^{-1}) = \sum_{a \in A_u} \frac{1}{p} = \frac{a_u}{p}.$$

Для доказательства попарной независимости достаточно показать, что

$$\mathsf{P}(X(u) = 1, X(v) = 1) = \mathsf{P}(X(u) = 1) \mathsf{P}(X(v) = 1) = \frac{a_u a_v}{p^2}.$$

Имеем

$$\begin{aligned} \mathsf{P}(X(u) = 1, X(v) = 1) &= \\ &= \mathsf{P}(\exists a \in A_u, b \in A_v : xu + y = a, xv + y = b) = \\ &= \sum_{a \in A_u, b \in A_v} \mathsf{P}(xu + y = a, xv + y = b) = \sum_{a \in A_u, b \in A_v} \frac{1}{p^2} = \frac{a_u a_v}{p^2}. \end{aligned}$$

5.3 Полиномиальная детерминированная проверка простоты

Проблема проверки простоты натурального числа является одной из древнейших и классических задач теории чисел.

Неэффективные тесты (типа решета Эратосфена) были известны еще до нашей эры.

С развитием теории алгоритмов возник вопрос о существовании эффективных (полиномиальных) алгоритмов проверки простоты числа, которые за полиномиальное от длины входа ($\lceil \log(n+1) \rceil$ для числа n , заданного в двоичной системе) проверяют, является ли n простым числом.

Мы описываем ниже модифицированный алгоритм проверки простоты, предложенный в 2002 г. тремя индийскими математиками. Более точно, мы опираемся на статьи [AKS02; AKS04].

Некоторые определения и обозначения

Определение 5.3.1. Через $\text{НОД}(a, n)$ обозначим наибольший общий делитель (НОД) чисел a и n .

Определение 5.3.2. Пусть $\text{НОД}(r, n) = 1$. Тогда $o_r(n)$ — **порядок** n по модулю r : минимальное натуральное k такое, что

$$n^k \equiv 1 \pmod{r}.$$

Через $\varphi(r)$ обозначается **функция Эйлера**, равная числу взаимно простых с r чисел, не превосходящих r .

Определение 5.3.3. Через Z_n обозначим кольцо целых чисел по модулю n , а через F_p — конечное поле из p элементов, где p — простое число. Через $Z_n[X]$ обозначим кольцо многочленов с коэффициентами из Z_n , а через $F_p[X]$ — кольцо многочленов с коэффициентами из F_p .

Определение 5.3.4. Напомним, что для простого p и неприводимого в F_p многочлена $h(x)$ степени d , $F_p[X]/(h(X))$ — конечное поле порядка p^d . Мы будем пользоваться обозначением

$$f(X) \equiv g(X) \pmod{h(X), n}$$

для обозначения уравнения $f(X) = g(X)$ в кольце $Z_n[X]/(h(X))$.

Идея и основная лемма

Лемма 25. Пусть a — целое число, n — натуральное, $n \geq 2$ и $\text{НОД}(a, n) = 1$. Тогда n простое тогда и только тогда, когда

$$(X + a)^n \equiv X^n + a \pmod{n}.$$

Доказательство. Заметим, что если $0 < i < n$, то коэффициент при X^i в $(X + a)^n - (X^n + a)$ равен $\binom{n}{i} a^{n-i}$.

Рассмотрим два случая.

1. n — простое. Тогда $\binom{n}{i} \pmod{n} = 0$ и все коэффициенты равны нулю.
2. n — составное. Рассмотрим его простой делитель q и пусть k — максимальная степень такая, что n делится на q^k . Тогда q^k не является делителем $\binom{n}{q}$ и взаимно просто с a^{n-q} (докажите это в качестве упражнения). Следовательно, коэффициент при X^q не равен нулю (\pmod{n}).

□

Что дает нам эта лемма? На первый взгляд ничего, т.к. вычисление левой части тождества требует вычисления n коэффициентов в худшем случае. Однако она дала идею нового простого вероятностного теста на простоту, которая и привела к построению детерминированного полиномиального алгоритма проверки простоты числа. В некотором смысле можно рассматривать этот результат как *дерандомизацию соответствующего вероятностного алгоритма проверки специального тождества*.

Основная идея дерандомизации — заменить основное тождество на другое:

$$(X + a)^n = X^n + a \pmod{X^r - 1, n},$$

для некоторого специального r , ограниченного полиномом от $\log n$. К сожалению, в таком виде этому тождеству будут удовлетворять и некоторые составные числа. Ключевая идея авторов алгоритма заключалась в проверке данного тождества для целой серии значений a , число которых ограничено полиномом от $\log n$.

Им удалось показать, что данной серии тождеств удовлетворяют только простые числа, и, таким образом, построить детерминированный полиномиальный алгоритм проверки простоты произвольного натурального числа. На вход алгоритма подается число n , записанное в двоичной системе, а результатом алгоритма является 1, если n — простое, и 0, если n — составное.

Время работы

Покажем, что алгоритм 36 полиномиален.

Приведем без доказательства известный факт из теории чисел.

Лемма 26. *Обозначим через $LCM(m)$ наименьшее общее кратное первых m натуральных чисел. Тогда $LCM(m)$ не меньше 2^m при $m \geq 7$.*

Алгоритм 36 Полиномиальный алгоритм проверки простоты

Вход: целое $n > 1$.**Выход:** целое 1 — если простое, и 0, если n — составное.

1. **if** ($n = a^b$ для $a \in N$ и $b > 1$) **return** 0.
 2. Найти наименьшее r такое, что $o_r(n) > \log^2 n$.
 3. **for** $a = 1$ **to** r
do if $1 < \text{НОД}(a, n) < n$ **return** 0.
 4. **if** $n \leq r$, **return** 1.
 5. **for** $a = 1$ **to** $\lfloor \sqrt{\varphi(r)} \log n \rfloor$ **do**
if $(X + a)^n \neq X^n + a \pmod{X^r - 1, n}$ **return** 0.
 6. **return** 1.
-

Лемма 27. Существует $r \leq \max(3, \lceil \log^5 n \rceil)$, такое, что $o_r(n) > \log^2 n$.**Доказательство.** При $n = 2$ значение $r = 3$ тривиально удовлетворяет всем условиям. Предположим, что $n > 2$.Тогда $\lceil \log^5 n \rceil > 10$, и мы можем воспользоваться леммой 26.Пусть r_1, r_2, \dots, r_t — все числа, такие, что $o_{r_i}(n) \leq \log^2 n$, а q_1, \dots, q_k — делители n . Пусть M — множество, состоящее из элементов r_i, q_j и всевозможных произведений $r_i q_j$. Каждое из чисел множества M должно делить произведение

$$n \cdot \prod_{i=1}^{\lfloor \log^2 n \rfloor} (n^i - 1) < n^{\log^4 n} \leq 2^{\log^5 n}.$$

По лемме наименьшее общее кратное первых $\lceil \log^5 n \rceil$ натуральных чисел не меньше $2^{\lceil \log^5 n \rceil}$, и, следовательно, существует число $s \leq \lceil \log^5 n \rceil$, такое, что $s \notin M$. Если $\text{НОД}(s, n) = 1$, то все доказано. Если же $\text{НОД}(s, n) > 1$, то, поскольку s не делит n и $\text{НОД}(s, n) = q_j$ при некотором j , то $r = \frac{s}{\text{НОД}(s, n)} = \frac{s}{q_j} \neq r_i$ ни при каком i , и, следовательно, $o_r(n) > \log^2 n$.

□

Упражнение 5.3.1. Покажите, что шаг 1 можно выполнить за полиномиальное время.

Из леммы 27 вытекает, что шаги 2–5 выполнимы за полиномиальное время.

Отсюда получаем следующую теорему.

Теорема 21. Алгоритм 36 полиномиален.

Упражнение 5.3.2. Для каких n можно исключить шаг 4 из алгоритма 36?

Корректность

То, что для простого n алгоритм выдает 1, очевидно следует из описания алгоритма.

Нам надо доказать, что если алгоритм выдает 1, то n является простым числом (т. е. если число составное, то сработает какая-либо из «проверок» алгоритма, и алгоритм вернет 0).

Итак, допустим, число n — составное, но шаги 1 и 3 его не выявили. Рассмотрим последний «фильтр» — самый содержательный шаг алгоритма, шаг 5, внимательно.

Поскольку $o_r(n) > 1$ (кстати, почему?), должно существовать простое p , делящее n , такое, что $o_r(p) > 1$ (заметим, что если n — простое, то $p = n$). Зафиксируем числа p и n . Пусть также $l = \lfloor \sqrt{\varphi(r)} \log n \rfloor$.

На шаге 5 алгоритма проверяется l уравнений. Поскольку по нашему предположению алгоритм не выдает 0, то для любого $0 \leq a \leq l$

$$(X + a)^n = X^n + a \pmod{X^r - 1, n},$$

откуда следует

$$(X + a)^n = X^n + a \pmod{X^r - 1, p}.$$

С другой стороны, из основной леммы 25 следует, что

$$(X + a)^p = X^p + a \pmod{X^r - 1, p}.$$

Из двух предыдущих равенств следует, что

$$(X + a)^{n/p} = X^{n/p} + a \pmod{X^r - 1, p}.$$

Упражнение 5.3.3. Докажите этот факт, используя тождество

$$(X + a)^p \equiv X^p + a^p \pmod{p}.$$

Доказательство. Отметим, что

$$\mathbb{Z}[X]/(X^r - 1, p) = \mathbb{F}_p[X]/(X^r - 1).$$

Пусть

$$(X + a)^{n/p} - X^{n/p} - a = q(X) \in \mathbb{F}_p[X].$$

Тогда в кольце $\mathbb{F}_p[X]$ выполняется равенство

$$\left((X + a)^{n/p} - X^{n/p} - a \right)^p = (q(X))^p.$$

Возводя в степень p в кольце $\mathbb{F}_p[X]$ левую часть равенства, получим

$$\begin{aligned} \left((X + a)^{n/p} - X^{n/p} - a \right)^p &= (X + a)^n - (X^{n/p} + a)^p = \\ &= (X + a)^n - (X^n + a). \end{aligned}$$

Следовательно, в кольце $\mathbb{F}_p[X]$ выполняется равенство

$$(X + a)^n - (X^n + a) = (q(X))^p.$$

С другой стороны, по условию задачи в кольце $\mathbb{F}_p[X]$ выполняется равенство

$$(X + a)^n - (X^n + a) = s(X)(X^r - 1).$$

Следовательно, в кольце $\mathbb{F}_p[X]$ выполняется равенство

$$s(X)(X^r - 1) = (q(X))^p.$$

Заметим, что многочлен $X^r - 1 \in \mathbb{F}_p[X]$ не имеет кратных корней в его поле разложения. Действительно, его производная равна rX^{r-1} , а согласно условию $o_r(p) > 1$ и, следовательно, $\text{НОД}(r, p) = 1$. Поэтому $r \neq 0$ в кольце \mathbb{Z}_p , и равенство rX^{r-1} возможно только при $X = 0$. Но $X = 0$ не является корнем многочлена $X^r - 1$, что и означает простоту его корней. Тогда из теоремы единственности разложения на

множители в кольце многочленов $\mathbb{F}_p[X]$ следует, что многочлен $q(X)$ делится на многочлен $X^r - 1$, т. е. $q(X) = q_1(X)(X^r - 1)$. Следовательно, в кольце $\mathbb{F}_p[X]$ выполняется равенство

$$(X + a)^{n/p} - X^{n/p} - a = q(X) = q_1(X)(X^r - 1),$$

т. е. выполняется равенство

$$(X + a)^{n/p} - X^{n/p} - a = q_1(X)(X^r - 1) \equiv 0 \pmod{X^r - 1, p}.$$

□

Определение 5.3.5. Для полинома $f(X)$ натуральное число m **принадлежит** множеству $I[f(X)]$, если

$$f(X)^m = f(X^m) \pmod{X^r - 1, p}.$$

Лемма 28. Множество $I[f(X)]$ замкнуто относительно умножения. Если $m_1, m_2 \in I[f(X)]$, то $m_1 \cdot m_2 \in I[f(X)]$.

Доказательство. Поскольку $m_1 \in I[f(X)]$, то

$$[f(X)]^{m_1 \cdot m_2} \equiv [f(X^{m_1})]^{m_2} \pmod{X^r - 1, p}.$$

Поскольку $m_2 \in I[f(X)]$, то заменяя X на X^{m_1} , имеем

$$\begin{aligned} [f(X^{m_1})]^{m_2} &\equiv f(X^{m_1 \cdot m_2}) \pmod{X^{m_1 r} - 1, p} \equiv \\ &\equiv f(X^{m_1 \cdot m_2}) \pmod{X^r - 1, p}, \end{aligned}$$

поскольку $X^r - 1$ является делителем многочлена $X^{m \cdot r} - 1$.

Из этих двух соотношений получаем

$$[f(X)]^{m_1 \cdot m_2} \equiv f(X^{m_1 \cdot m_2}) \pmod{X^r - 1, p}.$$

□

Лемма 29. Если $m \in I[f(X)]$ и $m \in I[g(X)]$, то

$$m \in I[f(X) \cdot g(X)].$$

Доказательство. Имеем

$$\begin{aligned} [f(X) \cdot g(X)]^m &= [f(X)]^m \cdot [g(X)]^m = \\ &= f(X^m) \cdot g(X^m) \pmod{X^r - 1, p}. \end{aligned}$$

Определим два множества, играющие ключевую роль в доказательстве корректности алгоритма:

$$I = \left\{ \left(\frac{n}{p} \right)^i \cdot p^j \mid i, j \geq 0 \right\},$$

$$P = \left\{ \prod_{a=0}^l (X + a)^{e_a} \mid e_a \geq 0 \right\}.$$

□

Простым следствием лемм 28 и 29 является

Лемма 30. Для любого $m \in I$ выполнено $m \in I[P]$.

Определим теперь две группы, связанные с введенными выше двумя множествами. Пусть G обозначает мультиликативную группу всех вычетов чисел из I по модулю r . Тогда G является подгруппой мультиликативной группы Z_r^* , поскольку, как отмечено выше, $\text{НОД}(n, r) = \text{НОД}(p, r) = 1$. Пусть $t = |G|$ — число элементов группы G . Группа G порождается элементами n и p по модулю r и, поскольку, $o_r(n) > \log^2 n$, то $t > \log^2 n$.

Воспользуемся далее некоторыми известными фактами из теории конечных полей.

Факт 1. Мультиликативная группа любого конечного поля циклическая.

Факт 2. Для любого поля F существует его алгебраическое замыкание \overline{F} .

Факт 3. Рассмотрим расширение поля \mathbb{F}_p , полученное присоединением всех корней многочлена $X^r - 1$.

Имеем $\mathbb{F}_p \subset \mathbb{F}_p(\alpha_1, \dots, \alpha_r) \subset \overline{\mathbb{F}_p}$.

Факт 4. Возьмем мультиликативную подгруппу расширенного поля, состоящую из корней многочлена $X^r - 1$. Эта группа циклическая (из факта 1, как подгруппа циклической группы). Она содержит примитивный элемент α (r -й корень из единицы, порождающий всю циклическую группу). Тогда $\mathbb{F}_p \subseteq \mathbb{F}_p(\alpha) = F$.

Возьмем минимальный многочлен элемента α над полем \mathbb{F}_p и обозначим его $h(X)$. Этот многочлен неприводим в $\mathbb{F}_p[X]$ и имеет степень $o_r(p)$ (см. [ЛН88]). Такой многочлен всегда существует, если $\text{НОД}(p, r) = 1$, причем его степень $o_r(p) > 1$. Пусть H обозначает множество всех вычетов многочленов из P по модулю $h(X)$ и p .

Упражнение 5.3.4. Доказать, что фактор-множество H является группой.

Группа H порождается элементами $X, X + 1, X + 2, \dots, X + l$ в поле $F = \mathbb{F}_p[X]/(h(X))$ и является подгруппой мультиликативной группы поля F .

Следующие две леммы дают нижнюю и верхнюю оценку размера группы H .

Лемма 31. $|H| \geq \binom{t+l}{t-1}$.

Доказательство. Заметим сначала, что X — примитивный r -й корень из единицы в F (см. факт 4 выше).

Покажем теперь, что любые два различных многочлена в множестве P степени меньше, чем t , соответствуют различным элементам H . Пусть $f(X)$ и $g(X)$ — такие многочлены из P . Предположим $f(X) = g(X)$ в поле F , и пусть $m \in I$. Имеем $[f(X)]^m = [g(X)]^m$ в F . Поскольку $m \in I[f]$, $m \in I[g]$, и $h(X)$ является делителем $X^r - 1$, получаем, что в поле F выполнено

$$f(X^m) = g(X^m).$$

Отсюда вытекает, что X^m является корнем многочлена $Q(Y) = f(Y) - g(Y)$ для любого $m \in G$. Поскольку $\text{НОД}(m, r) = 1$ (т.к. G — подгруппа Z_r^*), любое такое X^m является примитивным r -м корнем из единицы. Следовательно, должно быть $|G| = t$ различных корней многочлена $Q(Y)$ в поле F . Однако степень $Q(Y)$ меньше t в силу выбора f и g . Это противоречие доказывает, что $f(X) \neq g(X)$ в F .

Заметим, что $i \neq j$ в F_p для $1 \leq i \neq j \leq l$, поскольку $l = \lfloor \sqrt{\varphi(r)} \log n \rfloor < \sqrt{r} \log n < r$, и $p > r$.

Значит все элементы $X, X+1, X+2, \dots, X+l$ различны в поле F . Также, поскольку степень h больше единицы, $X+a \neq 0$ в F для всех a , $0 \leq a \leq l$. Значит, существует не менее $l+1$ различных полиномов первой степени в H . Отсюда вытекает, что существует не менее $\binom{t+l}{t-1}$ различных полиномов степени $< t$ в H . \square

Лемма 32. Если n не является степенью p , то

$$|H| \leq n^{\sqrt{t}}.$$

Доказательство. Рассмотрим следующее подмножество множества I :

$$I_1 = \left\{ \left(\frac{n}{p} \right)^i \cdot p^j \mid 0 \leq i, j \leq \lfloor \sqrt{t} \rfloor \right\}.$$

Если n не является степенью простого числа, то

$$|I_1| = (\lfloor \sqrt{t} \rfloor + 1)^2 > t.$$

Поскольку $|G| = t$, по крайней мере два числа из I_1 должны быть равны по модулю r . Пусть это будут m_1 и m_2 и $m_1 > m_2$. Имеем

$$X^{m_1} = X^{m_2} \pmod{X^r - 1}.$$

Пусть $f(X) \in P$. Тогда

$$\begin{aligned} [f(X)]^{m_1} &= f(X^{m_1}) \pmod{X^r - 1, p} = \\ &= f(X^{m_2}) \pmod{X^r - 1, p} = \\ &= [f(X)]^{m_2} \pmod{X^r - 1, p}. \end{aligned}$$

Отсюда вытекает, что в поле F

$$[f(X)]^{m_1} = [f(X)]^{m_2}.$$

Следовательно, $f(X) \in H$ является корнем многочлена $Q_1(Y) = Y^{m_1} - Y^{m_2}$ в поле F . Поскольку $f(X)$ — произвольный элемент H , многочлен $Q_1(Y)$ имеет не менее $|H|$ различных корней в поле F . Однако степень многочлена $Q_1(Y)$ равна

$$m_1 \leq \left(\frac{n}{p} \cdot p\right)^{\lfloor \sqrt{t} \rfloor} \leq n^{\lfloor \sqrt{t} \rfloor}.$$

Это доказывает, что $|H| \leq n^{\lfloor \sqrt{t} \rfloor}$.

□

Завершающей является лемма, основанная на сравнении нижней и верхней оценок из лемм 31 и 32.

Лемма 33. Если алгоритм выдает ПРОСТОЕ, то n является простым числом.

Доказательство. Предположим алгоритм выдает ПРОСТОЕ. Из леммы 31 вытекает, что для $t = |G|$ и $l = \lfloor \sqrt{\varphi(r)} \log n \rfloor$:

$$\begin{aligned}
 |H| &\geq \binom{t+l}{t-1} = \frac{1}{(l+1)!} \cdot t \cdot \dots \cdot (t+l) \geq \\
 &\geq \frac{1}{(l+1)!} \cdot (\lfloor \sqrt{t} \log n \rfloor + 1) \cdot \dots \cdot (\lfloor \sqrt{t} \log n \rfloor + l + 1) \\
 &\quad (\text{поскольку } t > \sqrt{t} \log n) = \\
 &= \binom{l+1 + \lfloor \sqrt{t} \log n \rfloor}{\lfloor \sqrt{t} \log n \rfloor} \geq \binom{2\lfloor \sqrt{t} \log n \rfloor + 1}{\lfloor \sqrt{t} \log n \rfloor} \\
 &\quad (\text{поскольку } l = \lfloor \sqrt{\varphi(r)} \log n \rfloor \geq \lfloor \sqrt{t} \log n \rfloor) > \\
 &> 2^{\lfloor \sqrt{t} \log n \rfloor + 1} \geq n^{\sqrt{t}},
 \end{aligned}$$

поскольку $\lfloor \sqrt{t} \log n \rfloor > \lfloor \log^2 n \rfloor \geq 1$ и $\binom{2m+1}{m} > 2^{m+1}$ при $m \geq 1$.

По лемме 32 $|H| \leq n^{\sqrt{t}}$, если n не степень p . Следовательно, $n = p^k$ для некоторого $k > 0$. Если $k > 1$, то алгоритм выдаст 0 на шаге 1. Значит, $n = p$. \square

Окончательно получаем следующую теорему.

Теорема 22. Алгоритм выдает ПРОСТОЕ $\iff n$ — простое.

Глава 6

Основы теории сложности вычислений

При написании этой главы использовались курсы лекций
[Lov99], [Gol99], а также отчет [KP96] и книга [КШВ99].

6.1 Сложность вычислений

6.1.1 Машины Тьюринга и вычислимость

Неформально, машина Тьюринга (далее **МТ**) представляет собой автомат с конечным числом состояний и неограниченной памятью, представленной набором одной или более лент, бесконечных в обоих направлениях. Ленты поделены на бесконечное число ячеек, и на каждой ленте выделена стартовая ячейка. В каждой ячейке может быть записан только один символ из некоторого конечного алфавита Σ , где

предусмотрен символ $*$ для обозначения пустой ячейки.

На каждой ленте имеется головка чтения-записи, и все они подсоединенны к «управляющему модулю» МТ — автомата с конечным множеством состояний Γ . Имеется выделенное стартовое состояние «START» и состояние завершения «STOP». Перед запуском МТ находится в состоянии «START», а все головки позиционированы на нулевые ячейки соответствующих лент. На каждом шаге все головки считывают информацию из своих текущих ячеек и посылают ее управляющему модулю МТ. В зависимости от этих символов и собственного состояния управляющий модуль производит следующие операции:

1. посылает каждой головке символ для записи в текущую ячейку каждой ленты;
2. посылает каждой головке одну из команд «LEFT», «RIGHT», «STAY»;
3. выполняет переход в новое состояние (которое, впрочем, может совпадать с предыдущим).

Теперь то же самое более формально.

Определение 6.1.1. *Машина Тьюринга* — это набор $T = \langle k, \Sigma, \Gamma, \alpha, \beta, \gamma \rangle$, где

- $k \geq 1$ — число лент;
- Σ — алфавит лент, $* \in \Sigma$ — символ-пробел;
- Γ — конечное множество состояний, $S, Q \in \Gamma$ — выделенные состояния: запуск машины и завершение работы;

- α, β, γ — произвольные отображения:

$$\begin{aligned}\alpha : \Gamma \times \Sigma^k &\rightarrow \Gamma, \\ \beta : \Gamma \times \Sigma^k &\rightarrow \Sigma^k, \\ \gamma : \Gamma \times \Sigma^k &\rightarrow \{-1, 0, 1\}^k.\end{aligned}$$

т. е. α задает новое состояние, β — символы для записи на ленты, γ — перемещение головок. Таким образом, машина Тьюринга задается таблицей команд размером $|\Sigma|^k \times |\Gamma|$, задающей правила работы машины в соответствии с функциями α, β, γ . Удобно считать, что алфавит Σ содержит кроме «пробела» $*$ два выделенных символа — 0 и 1¹.

Под *входом* для МТ подразумевается набор из k слов (k -кортеж) из Σ^* , записанных справа от стартовых позиций на k лентах МТ. Обычно входные данные записывают только на первую ленту, и под входом x подразумевают k -кортеж $\langle x, \emptyset, \dots, \emptyset \rangle$ — так мы и будем считать дальше в этом разделе.

Результатом работы МТ на некотором входе X считается слово, записанное на последней ленте после остановки МТ (слова, записанные на остальных лентах, принято игнорировать).

Алфавит входного слова будем обозначать $\Sigma_0 = \Sigma \setminus \{\star\}$. Да, мы будем считать без потери общности, что входное слово не содержит пробелов $*$ — иначе возникнут технические сложности, как определить, где кончается входное слово и т. п.

Если что-то осталось непонятным, можно посмотреть на алгоритм 37 — симулятор работы МТ, который мы будем использовать, чтобы проиллюстрировать процесс работы машин Тьюринга. Он принимает на вход описание машины Тьюринга в виде таблицы команд (см. рис. 6.1).

Обратите также внимание на альтернативное представление машин Тьюринга в виде ориентированных графов, где вершины являются состояниями, а дуги — возможными сменами состояний, причем на-

¹Обычно вовсе ограничиваются алфавитом $\Sigma \equiv \{\star, 0, 1\}$.

чало дуги помечено символом, который должен быть на ленте для активации перехода, а конец дуги помечен символом, который пишется на ленту, и командой перемещения головки: «L» (влево), «R» (вправо), «» (на месте).

Рассмотрим несколько примеров машин Тьюринга:

1. «удвоение строки» (рис. 6.1);
2. «унарное сложение» (рис. 6.2);
3. «распознавание четных строк» (рис. 6.3);
4. «распознавание строки с одинаковым количеством 0 и 1» (рис. 6.4 и 6.5).

Для каждой машины показано:

- табличное описание МТ;
- граф переходов;
- история выполнения для различных входных слов, где для каждого такта выделено положение головки и указано текущее состояние.

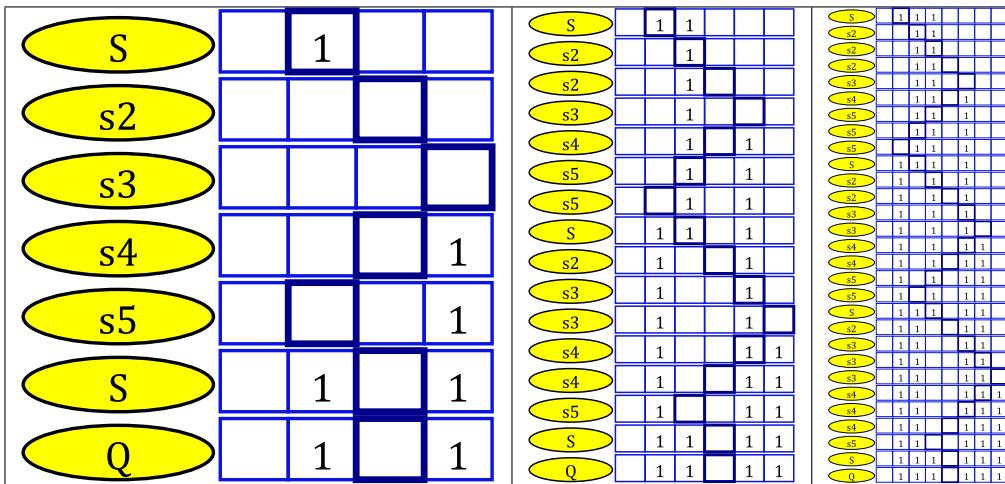
Упражнение 6.1.1. Постройте машину Тьюринга, которая записывает входное двоичное слово в обратном порядке.

Упражнение 6.1.2. Постройте машину Тьюринга, которая складывает два числа, записанные в двоичной системе. Для определенности считайте, что записи чисел разделены специальным символом алфавита «+».

Алгоритм 37 Симулятор работы машины Тьюринга

```
def execute_turing_machine(mt, tape_in):
    T = mt["program"]           # программа/таблица переходов
    tape = ["*"] + tape_in       # дописываем пробел слева от входа
    state = mt["start"]         # начальное состояние
    position = 1                # положение головки
    step = 0                     # счетчик тактов
    while state != mt["stop"] and step < 1000:
        step += 1
        if position >= len(tape): # потенциальная бесконечность
            tape.append("*")      # ленты
        symbol = tape[position]
        state, (symbol_to_write, move) = T[(state, (symbol))][:2]
        tape[position] = symbol_to_write
        if move == "L":
            position -= 1
        if move == "R":
            position += 1
```

$\langle S \rangle$	*	\Rightarrow	[Q]	*		
$\langle S \rangle$	1	\Rightarrow	s2	*	R	
s2	*	\Rightarrow	s3	*	R	
s2	1	\Rightarrow	s2	1	R	
s3	*	\Rightarrow	s4	1	L	
s3	1	\Rightarrow	s3	1	R	
s4	*	\Rightarrow	s5	*	L	
s4	1	\Rightarrow	s4	1	L	
s5	*	\Rightarrow	$\langle S \rangle$	1	R	
s5	1	\Rightarrow	s5	1	L	



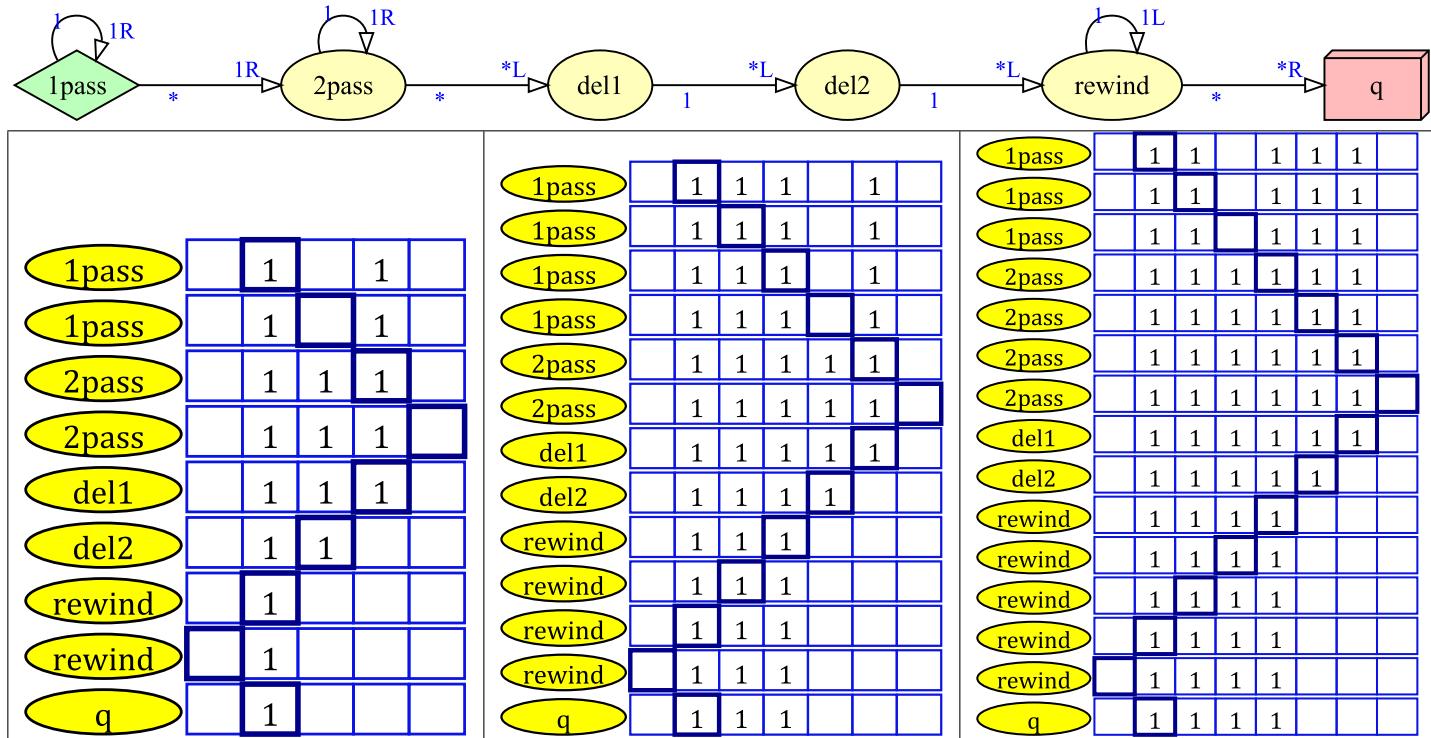
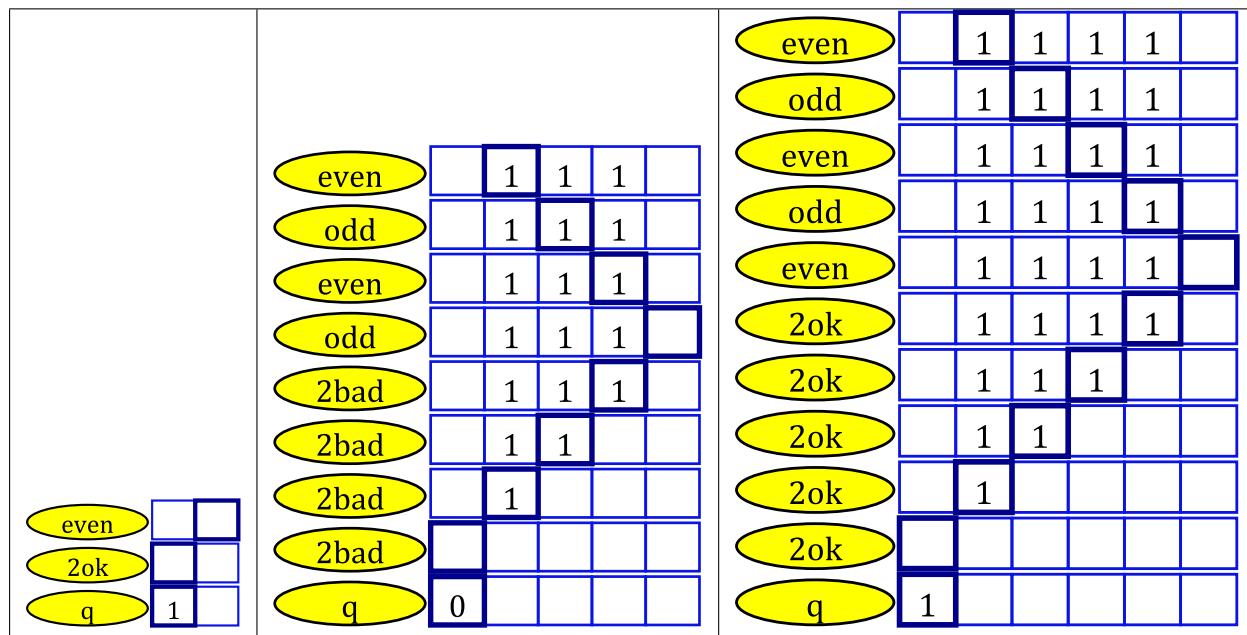
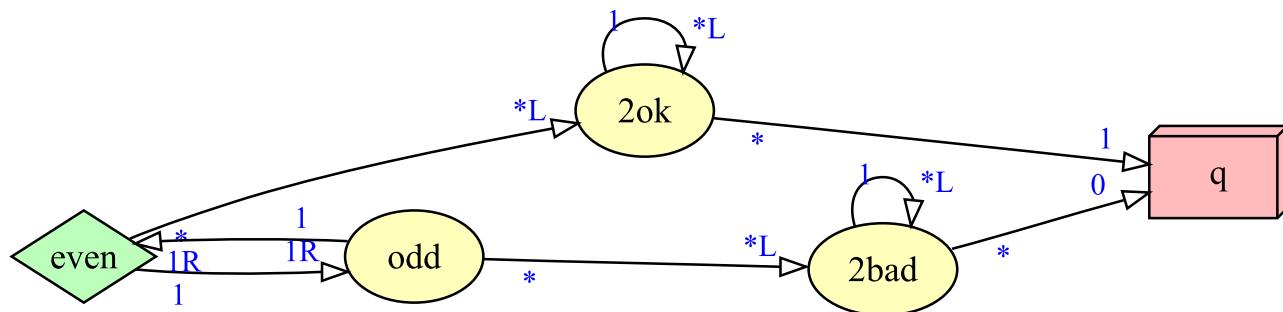
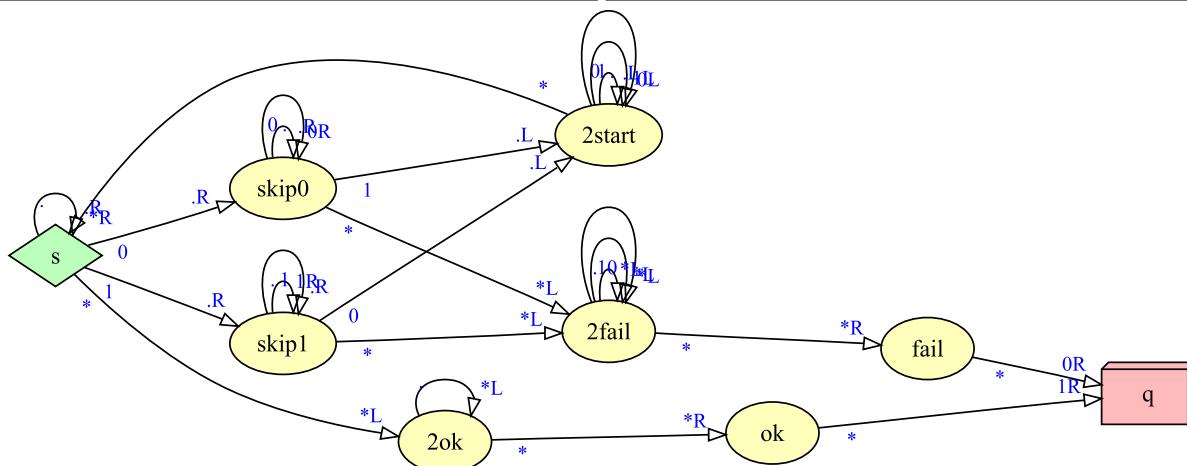


Рис. 6.2: Машина Тьюринга: унарное сложение



2fail	*	\Rightarrow	fail	*	R	
2fail	.	\Rightarrow	2fail	*	L	
2fail	0	\Rightarrow	2fail	*	L	
2fail	1	\Rightarrow	2fail	*	L	
2ok	*	\Rightarrow	ok	*	R	
2ok	.	\Rightarrow	2ok	*	L	
2start	*	\Rightarrow	<s>	*	R	
2start	.	\Rightarrow	2start	.	L	
2start	0	\Rightarrow	2start	0	L	
2start	1	\Rightarrow	2start	1	L	
fail	*	\Rightarrow	[q]	0	R	
ok	*	\Rightarrow	[q]	1	R	<1>
<s>	*	\Rightarrow	2ok	*	L	1 0 - Ok
<s>	.	\Rightarrow	<s>	.	R	
<s>	0	\Rightarrow	skip0	.	R	
<s>	1	\Rightarrow	skip1	.	R	
skip0	*	\Rightarrow	2fail	*	L	
skip0	.	\Rightarrow	skip0	.	R	
skip0	0	\Rightarrow	skip0	0	R	0-
skip0	1	\Rightarrow	2start	.	L	
skip1	*	\Rightarrow	2fail	*	L	
skip1	.	\Rightarrow	skip1	.	R	
skip1	0	\Rightarrow	2start	.	L	
skip1	1	\Rightarrow	skip1	1	R	1-

2fail	*	\Rightarrow	fail	*	R	
2fail	.	\Rightarrow	2fail	*	L	
2fail	0	\Rightarrow	2fail	*	L	
2fail	1	\Rightarrow	2fail	*	L	
2ok	*	\Rightarrow	ok	*	R	
2ok	.	\Rightarrow	2ok	*	L	
2start	*	\Rightarrow	<s>	*	R	
2start	.	\Rightarrow	2start	.	L	
2start	0	\Rightarrow	2start	0	L	
2start	1	\Rightarrow	2start	1	L	
fail	*	\Rightarrow	[q]	0	R	
ok	*	\Rightarrow	[q]	1	R	<1>
<s>	*	\Rightarrow	2ok	*	L	1 0 - Ok
<s>	.	\Rightarrow	<s>	.	R	
<s>	0	\Rightarrow	skip0	.	R	
<s>	1	\Rightarrow	skip1	.	R	
skip0	*	\Rightarrow	2fail	*	L	
skip0	.	\Rightarrow	skip0	.	R	
skip0	0	\Rightarrow	skip0	0	R	0-
skip0	1	\Rightarrow	2start	.	L	
skip1	*	\Rightarrow	2fail	*	L	
skip1	.	\Rightarrow	skip1	.	R	
skip1	0	\Rightarrow	2start	.	L	
skip1	1	\Rightarrow	skip1	1	R	1-



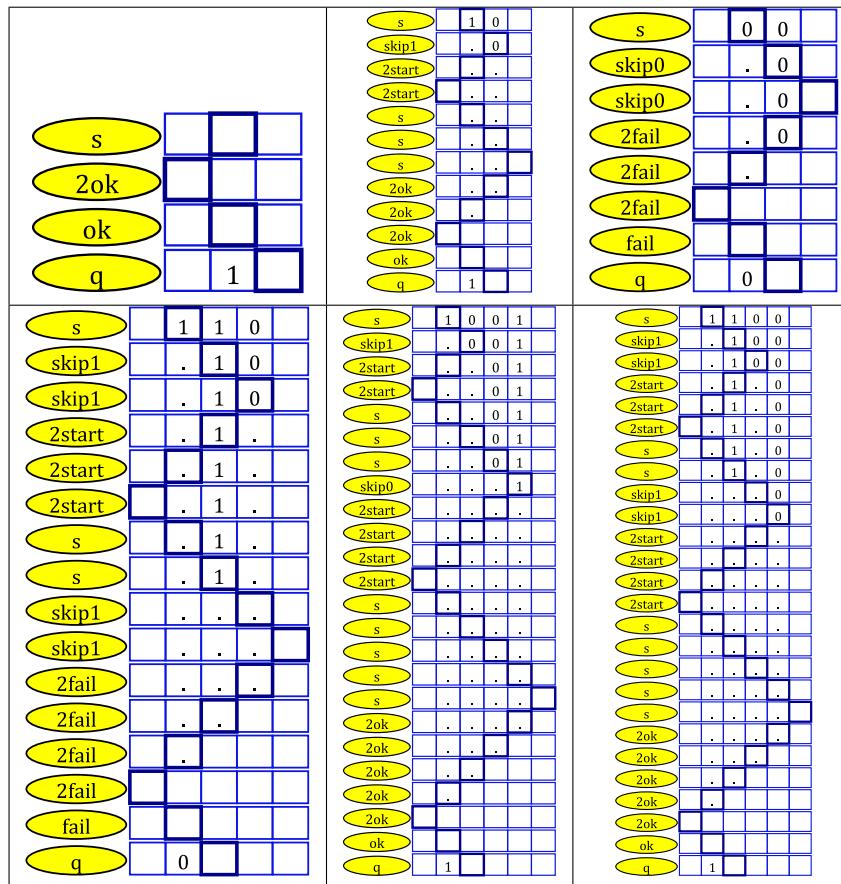


Рис. 6.5: Выполнение МТ «Количество 0 и 1 равно?»

Заметим, что в различной литературе встречается достаточно большое число разновидностей определений МТ — например, число лент ограничивается до одной, бесконечной только в одном направлении, или вводится «защита от записи» на все ленты, за исключением выходной, разумеется, и т. п.

Однако несложно, хотя и несколько утомительно, показать эквивалентность всех этих определений в смысле вычислительной мощности. Мы тоже слегка затронем эту тему.

Для начала введем понятие *универсальной машины Тьюринга*, которая уже напоминает больше современный программируемый компьютер, чем механическую шкатулку.

Пусть $T = \langle k + 1, \Sigma, \Gamma_T, \alpha_T, \beta_T, \gamma_T \rangle$ и $S = \langle k, \Sigma, \Gamma_S, \alpha_S, \beta_S, \gamma_S \rangle$, $k \geq 1$, две МТ, а $p \in \Sigma_0^*$. Итак, Т с программой p *симулирует* S , если для произвольного кортежа $\langle x_1, \dots, x_k \rangle \in \Sigma_0^{*k}$:

1. Т останавливается на входе (x_1, \dots, x_k, p) тогда и только тогда, когда S останавливается на (x_1, \dots, x_k) ;
2. в момент остановки Т на первых k лентах такое же содержимое, как и на лентах S после остановки на том же входе.

Определение 6.1.2. *$k + 1$ ленточная МТ Т **универсальна**, если для любой k -ленточной МТ S (над алфавитом Σ) существует программа $p \in \Sigma_0^*$, на которой Т симулирует S .*

Теорема 23. Для любого $k \geq 1$ и любого алфавита Σ существует $(k + 1)$ ленточная универсальная МТ.

Доказательство. Приведем конструктивное построение универсальной МТ. Основная идея заключается в том, чтобы разместить на дополнительной ленте универсальной МТ описание и текущее состояние моделируемой машины S .

Для начала опишем построение с $k + 2$ лентами. Для простоты будем считать, что алфавит Σ содержит символы «0», «1» и «-1». Пусть $S = \langle k, \Sigma, \Gamma_S, \alpha_S, \beta_S, \gamma_S \rangle$ — произвольная k -ленточная машина Тьюринга. Будем кодировать каждое состояние машины S словом фиксированной длины r над алфавитом Σ_0^* . Тогда

каждую строку из табличного представления машины S можно записать строкой-кодом фиксированной длины:

$$gt_1 \dots t_k \alpha_S(g, t_1, \dots, t_k) \beta_S(g, t_1, \dots, t_k) \gamma_S(g, t_1, \dots, t_k), \\ g \in \Gamma; t_i \in \Sigma; \forall i = 1, \dots, k.$$

Итак, на « $k + 1$ »-ой ленте у нас будет записано все табличное представление машины S в виде фиксированного размера кодов, на « $k + 2$ »-ой ленте изначально будет записано стартовое состояние S , на первых k лентах — входные данные.

Наша УМТ Т будет пробегать по « $k + 1$ »-ой ленте, пока текущее состояние моделируемой машины S и символы $t_1 \dots t_k$ на лентах $1 \dots k$ не совпадут с записанным на « $k + 2$ »-ой ленте кодом. Тогда из кода извлекаются инструкции, что делать с первыми k лентами, новое состояние, которое записывается на « $k + 2$ »-ую ленту, и все повторяется.

Для наглядности на рис. 6.6 приведен граф переходов для 3-х ленточной УМТ, эмулирующей одноленточную МТ. Большие прямоугольники обозначают состояния, вложенные прямоугольники — условия переходов в другие состояния, на ребрах-переходах прописаны совершаемые с лентами действия. $T(k)$, $k = 1, 2, 3$ — обозначают ленты, в контексте сравнения — символ под головкой данной ленты.

Изначально машина находится в состоянии $START$, на ленте $T(3)$ записан код стартового состояния S . В состоянии $START$ мы пытаемся сравнить текущий код на ленте 2 и текущее состояние S , записанное на ленте 3. Если они совпадают, и совпадает с ожидаемым символом на первой ленте, то «перематываем» к началу третью ленту ($REWIND_T3$), записываем на нее новое состояние из второй ленты ($WRITE_STATE$), записываем на первую ленту символ из второй ленты, двигаем куда надо головку первой ленты ($MOVE$), «перематываем» к началу вторую и третью ленту ($REWIND_T2_T3$), и возвращаемся в исходное состояние.

Иначе, по цепочке

`SKIP_T2_STATE → SKIP_T2_TRANSITION → SKIP_T2_MOVE`

(или более короткой) переходим к следующему коду (строке моделируемой МТ) на второй ленте.

Переход от $k+2$ лент к $k+1$ несложен — например, достаточно хранить содержимое « $k+2$ »-й ленты в отрицательной области « $k+1$ »-й ленты и моделировать 2 головки на « $k+1$ »-й ленте (одна из которых будет работать с состоянием S , а другая с программой S) методом, описанным в доказательстве следующей теоремы. \square

Следующая теорема утверждает, что в некотором смысле неважно, сколько лент в определении МТ.

Теорема 24. Для любой k -ленточной МТ S существует одноленточная МТ T , такая, что для любого $x \in \Sigma_0^*$, T останавливается на x тогда и только тогда, когда на x останавливается S , причем на ленте T после остановки записано то, что записано после остановки на последней ленте S .

На входе x , на котором S будет работать N шагов, время работы машины T будет $O(N^2)$.

Доказательство. Первым делом мы осуществляем «упаковку» всех лент моделируемой машины S на одну ленту машины T . Мы добиваемся соответствия i -й ячейки j -й ленты моделируемой машины S четной $2(ki + j - 1)$ ячейке единственной ленты машины T . Вернее, «упаковывается-растягивается» только входная, первая лента машины S , т.к. остальные ленты S по определению пусты перед запуском.

Позиции, соответствующие всем лентам S , кроме первой, заполняются пробелами.

Нечетные позиции $2(ki + j - 1) + 1$ на ленте мы используем для хранения информации о головках машины S — если у машины S в некотором состоянии в i -й позиции j -й ленты стояла головка, то в $2(ki + j - 1) + 1$ ячейку мы запишем 1, иначе пробел *. Также пометим 0 первые четные ячейки, соответствующие концам лент моделируемой машины S .

Теперь рассмотрим, как T непосредственно моделирует S . Во-первых, T «помнит» (за счет своих собственных состояний, а не дополнительных лент), в каком состоянии должна находиться моделируемая машина S . Также T помнит, какую «ленту» она в данный момент читает. За один проход по своей ленте T «выясняет», какие символы видны под каждой головкой моделируемой машины S , в какое состо-

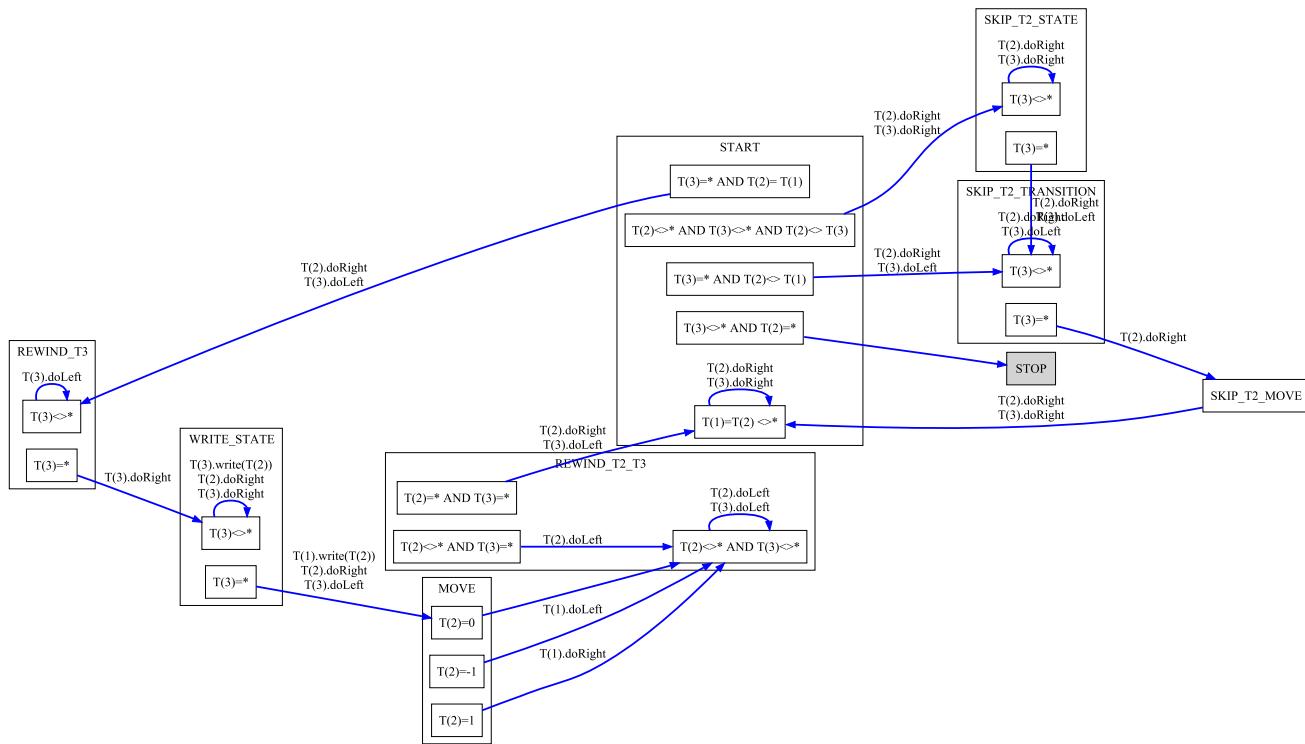


Рис. 6.6: Трехленточная универсальная МТ для одноленточных МТ

яние нужно перевести S , куда нужно двигать головки каждой ленты и что записывать на каждую ленту. Следующим проходом соответственно двигаются маркеры головок, пишутся символы на моделируемые ленты S .

После окончания моделирования вычисления S получившийся результат должен быть «сжат», что аналогично начальному «растяжению».

Очевидно, что описанная таким образом машина T вычисляет то же, что и моделируемая машина S . Теперь оценим число шагов T . Пусть M — число сканированных машиной T ячеек.

Упражнение 6.1.3. Покажите, что число сканированных машиной T ячеек $M = O(N)$.

Моделирование каждого шага S требует $O(M)$ шагов, таким образом, весь процесс моделирования состоит из $O(MN)$ шагов. Начальное «растяжение» и конечное «сжатие» требуют по $O(M^2)$ шагов.

Окончательно получаем, что все моделирование требует не более $O(N^2)$ шагов. \square

Таким образом, при рассуждениях можно ограничиваться рассмотрением одноленточных МТ, причем подразумевать под МТ ее программу и использовать выражения вроде «подать на вход машины Тьюринга T_1 машину Тьюринга $T_2 \dots$ ».

Теперь уже можно ввести строгое определение вычислимости.

Определение 6.1.3. Функция $f : N \rightarrow N$ является **вычислимой**, если существует такая машина Тьюринга T , что если на вход ей подать представленный в некоторой кодировке x , то

1. если функция f определена на x , и $f(x) = y$, то машина T останавливается на входе x , и на выходе у нее записано y ;
2. если функция f не определена на x , то машина T зацикливается (не останавливается за любое конечное число шагов) на входе x .

Аналогичным образом определяется понятие разрешимости и вычислимости для языков и других множеств².

Определение 6.1.4. Множество S (язык L) является *разрешимым*, если существует такая машина Тьюринга T , что если на вход ей подать элемент $x \in S$ (слово $l \in L$), то она остановится и выведет «1».

Иначе ($x \notin S, l \notin L$), T останавливается и выводит «0».

Итак, мы познакомились с формализацией понятия алгоритма и вычислимости через определение машины Тьюринга. Со времени первого определения понятия алгоритма было предложено множество различных универсальных моделей вычислений, зачастую весьма далеких от машин Тьюринга, RAM или даже реальных ЭВМ, однако никому еще не удалось предъявить пример процесса, который можно было бы признать алгоритмическим, но который невозможно было бы смоделировать на машине Тьюринга. Иными словами, любой вычислительный процесс может быть смоделирован на подходящей машине Тьюринга. Это так называемый тезис Тьюринга, также упоминаемый как тезис Черча или тезис Черча-Тьюринга, разделяется большинством специалистов. Таким образом, если мы принимаем этот тезис, то можем смело говорить о вычислимости, не указывая конкретную модель.

Сразу возникает вопрос: любую ли функцию $y = f(x)$ ³, можно вычислить на МТ?

Как проще всего убедиться в существовании невычислимых функций? Ответом служит формулировка следующего упражнения.

²Обратите внимание, что в теории формальных языков (теории реализации языков программирования, теории автоматных языков и регулярных выражений) принято говорить, что язык распознается автоматом, если автомат останавливается на словах из этого языка и не останавливается на остальных. В теории сложности для распознавания языка требуется, чтобы распознавающая машина Тьюринга останавливалась на всех словах.

³Можно подразумевать функции на множестве натуральных чисел, или преобразования строк — одно равнозначно другому.

Упражнение 6.1.4. Докажите, что существуют невычислимые по Тьюрингу функции $y = f(x)$. Использовать мощностные соображения.

Несмотря на то, что ответ получен в предыдущем упражнении, он несколько неконструктивный и не дает возможности «познакомиться» с представителем неразрешимой задачи (невычислимой функции).

Рассмотрим классическую неразрешимую задачу.

Задача 26. Проблема остановки (halting problem). Для данной машины Тьюринга M и входа x определить, остановится ли машина Тьюринга M , начав работу на x ?

Теорема 25. Проблема остановки алгоритмически неразрешима.

Доказательство. От противного. Предположим, что есть такой алгоритм, т. е. существует машина Тьюринга T , которая на входе $(M, x)^4$ дает ответ «да», если машина M останавливается на входе x , в противном случае дает ответ «нет». Тогда есть и такая машина $T^{diag}(X) \equiv T(X, X)$, которая на входе X моделирует работу T на «диагональном»⁵ входе (X, X) .

Надстроим над $T^{diag}(X)$ машину $T^{co}(X)$, которая если ответ машины T^{diag} — «да», то T^{co} начинает двигать головку вправо и не останавливается (зацикливается), а если ответ T^{diag} — «нет», то T^{co} останавливается.

Остановится ли T^{co} на входе T^{co} ?

1. Если да, то T^{diag} дает ответ «нет» на входе T^{co} , т. е. утверждает, что T^{co} не должна останавливаться на T^{co} .

⁴Под машиной Тьюринга M на входе подразумевается ее описание.

⁵Если у функции два однотипных параметра, то входы с равными параметрами общепринято называть *диагональными*.

2. Если не остановится, то T^{diag} дает ответ «да» на входе T^{co} , т. е. утверждает, что T^{co} должна останавливаться на T^{co} .

Противоречие. □

На самом деле в этой теореме в терминах машин Тьюринга переформулируется известный «парадокс брадобрея»:

Рассмотрим множество M тех брадобреев, которые бреют тех и только тех, которые не бреют сами себя. Если такой брадобрей не бреет сам себя, то он себя должен брить (по определению множества M). Если же он бреет сам себя, то он себя не должен брить (опять же по определению M).

Упражнение 6.1.5. Докажите, что также неразрешима версия задачи 26 «HALT» — «остановка на пустом слове», т. е. для данной МТ Т определить, остановится ли она на пустом слове.

Упражнение 6.1.6. Докажите, что неразрешима «Проблема недостижимого кода» — нет алгоритма, который для заданной машины Тьюринга Т и ее состояния q_k выясняет: попадет ли машина в это состояние хотя бы для одного входного слова x ?

Упражнение 6.1.7. Докажите, что не существует алгоритма, который выписывает одну за другой все машины Тьюринга, которые не останавливаются, будучи запущенными на пустой ленте.

Упражнение 6.1.8. Существует ли алгоритм, который выписывает одну за другой все машины Тьюринга, которые останавливаются, будучи запущенными на пустой ленте?

Упражнение 6.1.9. Докажите, для разрешимых языков L_1 и L_2 , язык $L = L_1 \cup L_2$ также разрешим.

Упражнение 6.1.10. Пусть M — машина Тьюринга с единственной лентой, которая копирует входное слово (приписывая его копию справа от самого слова). Пусть $T(n)$ — максимальное время ее работы на входах длины n . Докажите, что $T(n) \geq \varepsilon n^2$ для некоторого ε и для всех n . Что можно сказать про $T'(n)$, которое есть минимальное время ее работы на входах длины n ?

Упражнение 6.1.11. Пусть $T(n)$ — максимальное время, которое может пройти до остановки машины Тьюринга с n состояниями и n символами алфавита, если ее запустить на пустой ленте. Докажите, что. функция $T(n)$ растет быстрее любой вычислимой всюду определенной функции $b(n)$, то есть $\lim[T(n)/b(n)] = +\infty$.

Упражнение 6.1.12. Докажите, что разрешим язык L , состоящий из $n \in N$, таких, что строка «4815162342» встречается в десятичном разложении числа π не менее чем n раз подряд.

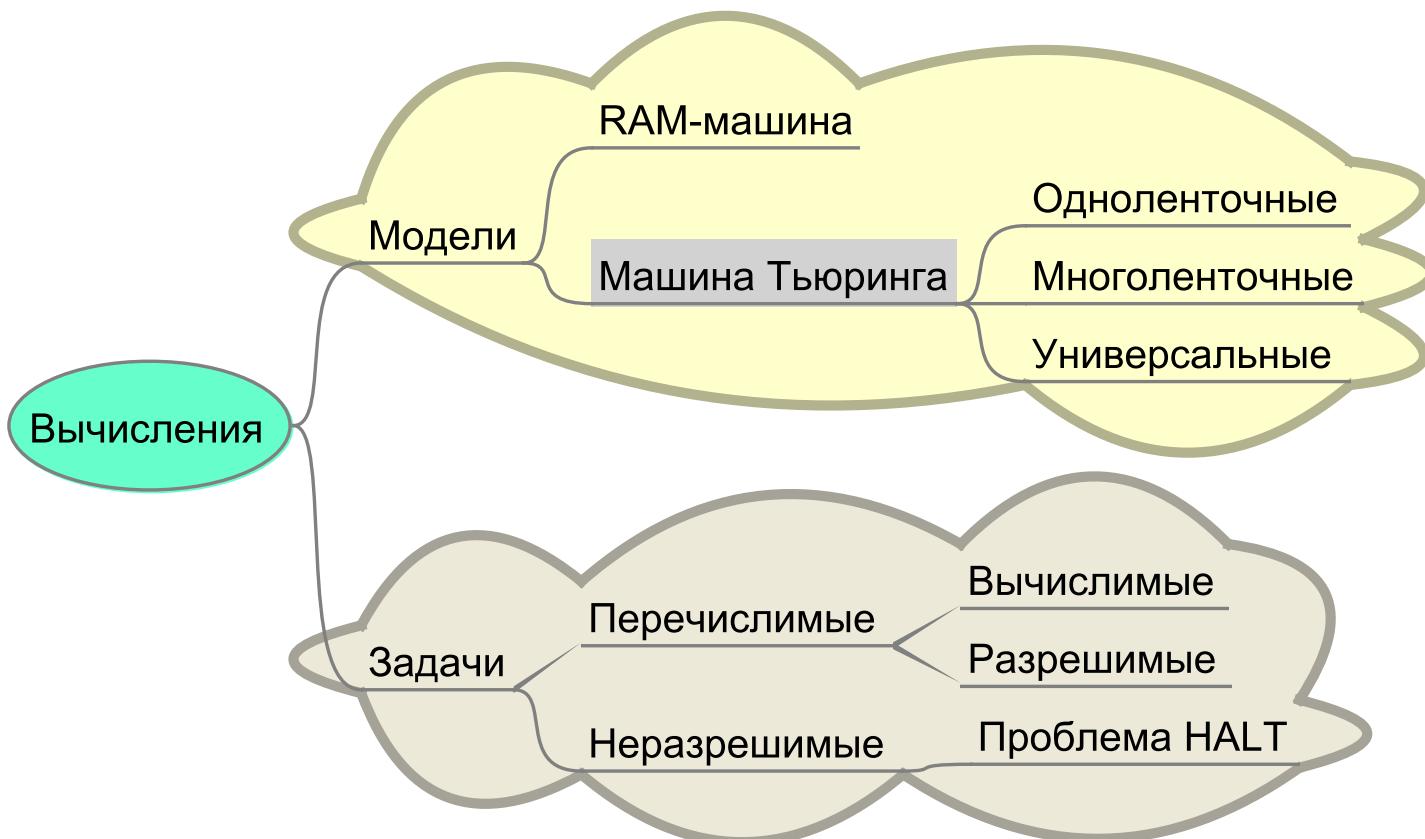
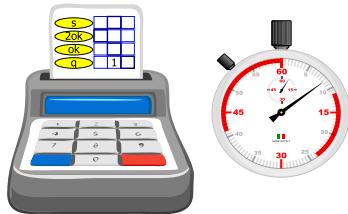


Рис. 6.7: Карта-памятка раздела 6.1.1

6.1.2 Классы \mathcal{DTIME} , \mathcal{DSPACE}



*Временная и пространственная сложность алгоритма.
Теорема об ускорении. Классы задач \mathcal{DTIME} и \mathcal{DSPACE} .*

После определения разрешимости хочется иметь меру сложности вычисления. Здесь и дальше мы будем рассматривать только разрешимые задачи и всюду определенные (не зацикливающиеся ни на одном входе) машины Тьюринга.

Под временем вычисления будем понимать число шагов машины Тьюринга до получения результата.

Определение 6.1.5. Пусть $t : N \rightarrow N$. Машина Тьюринга T имеет **временную сложность** (time complexity) $t(n)$, если для каждого входного слова длины n T выполняет не больше $t(n)$ шагов до остановки. Также будем обозначать временную сложность машины Тьюринга T , как $time_T(n)$.

Используемой памятью будем считать число ячеек на ленте, использованных для записи, не считая длины входа.

Определение 6.1.6. k -ленточная машина Тьюринга (произвольное $k > 0$) T имеет **пространственную сложность** $s(n)$, если для любого входного слова длины n T просматривает не более $s(n)$ ячеек на всех рабочих лентах (исключая входную ленту).

Обратите внимание, что пространственная сложность может быть меньше длины входа.

Упражнение 6.1.13. Дан неориентированный граф $G = (V, E)$, и вершины s и t . Придумайте алгоритм с пространственной сложностью $O(\log n)$ (n -длина входа), который возвращает «1», если есть путь из s в t длиной не больше $777 \log n$, проходящий только через вершины имеющие не больше 777 соседей, и «0» в противном случае.

Следующим шагом могло бы стать разумное определение «оптимального» алгоритма для данной алгоритмической задачи.

К сожалению, такой подход бесперспективен, и соответствующий результат (*теорема об ускорении*), установленный на заре развития теории сложности вычислений в работе [Blu67], послужил на самом деле мощным толчком для ее дальнейшего развития. Мы приведем этот результат (без доказательства) в ослабленной, но зато весьма наглядной форме.

Теорема 26. Существует разрешимая алгоритмическая задача, для которой выполнено следующее. Для произвольного алгоритма А, решающего эту задачу и имеющего сложность в наихудшем случае $time_A(n)$, найдется другой алгоритм В (для этой же задачи) со сложностью $time_B(n)$, такой, что

$$time_B(n) \leq \log_2 time_A(n)$$

выполнено для почти всех n (т. е. для всех n , начиная с некоторого).

Иными словами, любой алгоритм, решающий эту задачу, можно существенно ускорить, т. е. отыскать алгоритм намного меньшей асимптотической сложности. Следует сразу отметить, что задача, о которой идет речь в этой теореме, выглядит довольно искусственно, и, по-видимому, ничего подобного не происходит для задач, реально возникающих на практике. Тем не менее, теорема об ускорении не позволяет нам определить общее математическое понятие «оптимального» алгоритма, пригодное для всех задач, поэтому развитие теории эффективных алгоритмов пошло другим путем. Именно, одним из центральных

понятий этой теории стало понятие *класса сложности*. Так называется совокупность тех алгоритмических задач, для которых существует хотя бы один алгоритм с теми или иными сложностными характеристиками. Мы рассмотрим следующие классы сложности: \mathcal{P} , \mathcal{NP} , \mathcal{RP} , \mathcal{ZPP} , \mathcal{BPP} , \mathcal{PSPACE} , $\mathcal{EXPTIME}$, \mathcal{PCP} ; читателю, интересующемуся более глобальной картиной сложностной иерархии, мы рекомендуем обратиться к [Joh90; Aar07].

Для формальных определений классов сложности обычно рассматривают⁶ не произвольные алгоритмы, а алгоритмы для так называемых *задач разрешения* (*decision problem*), когда требуется определить, принадлежит или нет некоторый элемент некоторому множеству. Учитывая необходимость кодирования данных, подаваемых на вход машине Тьюринга, эти задачи абсолютно эквивалентны задачам распознавания языков, когда на некотором алфавите Σ рассматривается подмножество слов $L \subset \Sigma^*$, и для произвольного слова $l \in \Sigma^*$ нужно определить, принадлежит ли оно языку L .

Таким образом, каждый язык L определяет одну из задач разрешения, для которых мы сейчас более формально определим классы временной сложности.

Определение 6.1.7. Язык $L \subset \Sigma^*$ принадлежит классу $\mathcal{DTIME}(t(n))$, если существует машина Тьюринга T , разрешающая данный язык, и $\forall n : \text{time}_T(n) \leq t(n)$.

Определение 6.1.8.

$$\mathcal{P} \equiv \bigcup_{k \geq 0} \mathcal{DTIME}(n^k).$$

Определение 6.1.9.

$$\mathcal{EXPTIME} \equiv \bigcup_{k \geq 0} \mathcal{DTIME}(2^{n^k}).$$

Иными словами, класс \mathcal{P} состоит из тех алгоритмических задач, которые допускают решение хотя бы одним полиномиальным (в наихудшем случае) алгоритмом. Например, алгоритм 7 «Дейкстры» полино-

⁶О причинах будет говориться в следующих разделах.

миален, поэтому задача 5 «SPP» принадлежит классу \mathcal{P} . Этому же классу принадлежит и задача 7 «Minimum Spanning Tree».

Обычно обобщение классов сложности, введенных для задач разрешения, на произвольные вычислимые функции $f : N \rightarrow N$ происходит путем ассоциирования с произвольной вычислимой функцией $y = f(x)$ задачи разрешения «Для данных y, x проверить, правда ли, что $y = f(x)$ ».

В современной теории сложности вычислений понятие полиномиального алгоритма является адекватным математическим уточнением интуитивного понятия «эффективный алгоритм» (см. раздел 1.2.5), а класс \mathcal{P} представляет собой «класс эффективно решаемых задач».

Упражнение 6.1.14. Рассмотрим язык L состоящий из слов $\langle M, x, t \rangle$, для которых ДМТ M останавливается на x не позже, чем через t шагов. Докажите: $L \in \mathcal{EXPTIME}$.

Следующими по значимости после рассмотренных выше временных мер и классов сложности являются меры сложности, отражающие используемый алгоритмом объем памяти, т. е. максимальное число ячеек R_i , используемых алгоритмом (в наихудшем случае) на входах размера $\leq n$.

Определение 6.1.10. Язык $L \subset \Sigma^*$ принадлежит классу $\mathcal{DSPACE}(s(n))$, если существует машина Тьюринга T , разрешающая данный язык, и пространственная сложность T не превосходит $s(n)$.

Например, REG — регулярные языки (распознаваемые детерминированным конечным автоматом и задаваемые регулярными выражениями), принадлежат классу $DSPACE(O(1))$.

Определение 6.1.11.

$$\mathcal{PSPACE} \equiv \cup_{k \geq 0} \mathcal{DSPACE}(n^k).$$

Очевидно, что \mathcal{P} содержится в классе задач \mathcal{PSPACE} , разрешимых с полиномиальной памятью, просто в силу того, что за один такт можно просмотреть не больше одной новой ячейки памяти. Обратное,

по-видимому, неверно (хотя и не доказано строго — см. обсуждение в разделе 6.2). Наконец, стоит также упомянуть, что существуют различные меры и классы сложности, связанные с параллельными и распределенными вычислениями.

Упражнение 6.1.15. Покажите, что $\mathcal{P} \subseteq \mathcal{EXPTIME}$.

Упражнение 6.1.16. Покажите, что $\mathcal{PSPACE} \subseteq \mathcal{EXPTIME}$.

Упражнение 6.1.17.

$$\text{LOGSPACE} = \text{DSPACE}(O(\log n)).$$

Покажите, что $\text{LOGSPACE} \subseteq \mathcal{P}$.

Упражнение 6.1.18.

$$\text{LOGSPACE} = \text{DSPACE}(O(\log n)).$$

Рассмотрим язык «правильно вложенных скобок» L :

$$\in L \quad (), ()(), ((()), ((())()), \dots$$

$$\notin L \quad)(, \dots$$

Докажите, что $L \in \text{LOGSPACE}$.

6.2 Полиномиальные сводимости и \mathcal{NP} -полнота

Алгоритмическая задача называется *труднорешаемой*, если для нее не существует полиномиального алгоритма.

Известно, что бывают алгоритмические задачи, в принципе не разрешимые вообще никаким алгоритмом (см., например, задачу 26 «HALT»). Поэтому естественно задаться вопросом: а существуют ли *разрешимые* задачи, которые тем не менее не принадлежат классу \mathcal{P} ? Ответ на этот вопрос предоставляется теоремой об иерархии [ХС67], которая наряду с теоремой Блюма об ускорении является одним из краеугольных камней теории сложности вычислений. Так же, как и в случае с теоремой об ускорении, мы приводим ее упрощенный вариант.

Теорема 27. Существует алгоритмическая задача, разрешимая некоторым алгоритмом сложности $n^{O(\log n)}$, но не принадлежащая классу \mathcal{P} .

К сожалению, задачи, возникающие как в теореме об ускорении, так и в теореме об иерархии, носят довольно искусственный характер и по этой причине не могут быть использованы для сложностного анализа переборных задач дискретной оптимизации. Для этой цели используется теория \mathcal{NP} -полноты, изложение основ которой мы начинаем с понятия *полиномиальной сводимости*.

6.2.1 Сводимость по Куку

На неформальном уровне мы уже познакомились с этим понятием в разделе 1.1.

Определение 6.2.1. Алгоритмическая задача P_1 полиномиально сводится к задаче P_2 , если существует полиномиальный алгоритм для решения задачи P_1 , возможно, вызывающий в ходе своей работы процедуру для решения задачи P_2 .

При таком определении, однако, возникает один неприятный эффект. Например, алгоритм 13 «Переполнение памяти умножением», вне всякого сомнения, полиномиален, если последний оператор $R \leftarrow R \cdot R$ понимать как вызов процедуры умножения.

Мы уже отмечали в разделе 1.2.1, что для умножения также существует полиномиальный алгоритм. Тем не менее, составной алгоритм 13 «Переполнение памяти умножением» неполиномиален, и, более того, вычисляемая им функция 2^{2^t} по очевидным причинам не принадлежит классу \mathcal{P} . Таким образом, класс \mathcal{P} оказывается незамкнутым относительно полиномиальной сводимости, что, разумеется, ненормально. Понятны и причины этого: при повторных обращениях к функциональным процедурам может происходить лавинообразный рост значений числовых параметров. Наиболее кардинальный и в то же время общепринятый способ борьбы с этим явлением — с самого начала ограничиться рассмотрением лишь задач разрешения, т. е. таких, ответ на которые имеет вид ДА/НЕТ. При таком ограничении в полиномиальных сводимостях используются только предикатные процедуры, указанная выше проблема снимается, и класс \mathcal{P} уже будет замкнутым относительно полиномиальной сводимости.

Прежде чем двигаться дальше, отметим, что любую переборную задачу дискретной оптимизации можно без ограничения общности заменить на переборную же задачу разрешения, так что рассматриваемое ограничение не слишком обременительно. Мы проиллюстрируем, как осуществляется эта замена на примере задачи 4 «TSP», хотя видно, что тот же самый метод годится для любой задачи дискретной оптимизации.

А именно, давайте вместо *нахождения минимального значения суммы* (1.1) ограничимся более простым вопросом: верно ли, что $\text{Comm}(m, d) \leq B$, где $\text{Comm}(m, d)$ — минимально возможная стоимость пути коммивояжера, а B — новый числовой параметр?

Задача 27. «TSP-разрешимость». Заданы:

- n городов c_1, \dots, c_n ;
- $d_{ij} \equiv d(c_i, c_j) \in Z^+$ — расстояния между ними;
- B — положительное целое.

Верно ли, что минимально возможное значение суммы (1.1) меньше B ?

Тогда задача 4 «TSP» допускает простое полиномиальное сведение к этой задаче разрешения, осуществляемое с помощью *бинарного поиска*, когда интервал возможных решений делится пополам, а у алгоритма для решения задачи 27 «TSP-разрешимость» выясняется, в какой из половин лежит требуемое решение.

В теории сложности вычислений «процедурный» вариант полиномиальной сводимости называется *Тьюринговой сводимостью* или *сводимостью по Кук* в честь автора работы [Кук75].

6.2.2 Недетерминированные алгоритмы

Математическим уточнением понятия «переборная задача разрешения» служит «задача, разрешимая за полиномиальное время с помощью недетерминированного полиномиального алгоритма».

В недетерминированных алгоритмах дополнительно разрешаются *недетерминированные операторы перехода* вида

goto ℓ_1 or ℓ_2 .

Таким образом, для каждого массива входных данных имеется не один, а несколько (в общем случае — экспоненциальное число) путей, по которым может развиваться вычисление. Недетерминированный алгоритм по определению выдает окончательный ответ 1, если существует хотя бы один путь развития вычисления, на котором выдается ответ 1, и 0 — в противном случае (таким образом, ответы ДА и НЕТ в случае недетерминированных вычислений несимметричны).

Определение 6.2.2. *Недетерминированная машина Тьюринга (НМТ)* — это набор $T = \langle k, \Sigma, \Gamma, \Phi \rangle$, где

- $k \geq 1$ — натуральное число (число лент),

- Σ — «алфавит лент», конечное множество,
- $\star \in \Sigma$ — алфавит содержит «пробел»,
- Γ — конечное множество состояний, $START, STOP \in \Gamma$,
- Φ — произвольное отношение:

$$\Phi \subset (\Gamma \times \Sigma^k) \times (\Gamma \times \Sigma^k \times \{-1, 0, 1\}^k).$$

Переход из состояния g , с символами на лентах h_1, \dots, h_k будет допустим, если новое состояние g' , записанные символы h'_1, \dots, h'_k и смещения головок $\varepsilon_1, \dots, \varepsilon_k$ удовлетворяют соотношению

$$(g, h_1, \dots, h_k, g', h'_1, \dots, h'_k, \varepsilon_1, \dots, \varepsilon_k) \in \Phi.$$

Процесс вычисления недетерминированной машиной Тьюринга можно представлять таким образом, что в каждый момент «ветвления» — совершения недетерминированного перехода из нескольких возможных — происходит «клонирование» НМТ, и вычисление продолжается по всем возможным путям, пока одной из «клонированных» НМТ не будет получен утвердительный ответ («да», «1»), либо пока везде не будет получен ответ «0».

Очевидно, детерминированные машины Тьюринга являются частным случаем недетерминированных.

Сложностью (временем работы) недетерминированного алгоритма (НМТ) на входном слове x называется минимальная сложность вычисления, приводящего на этом входе к ответу «1», либо, если $x \notin L$, максимальная сложность, приводящая к «0».

Зная, что такое время работы НМТ для каждого входного слова x , можно ввести для НМТ и сложность в наихудшем случае.

По аналогии с определением 6.1.7 « \mathcal{DTIME} » и определением 6.1.10 « \mathcal{DSPACE} » определим классы $\mathcal{NTIME}(f(n))$ и $\mathcal{NSPACE}(f(n))$.

Важно отметить серьезное отличие классов сложности для НМТ и детерминированных МТ. Это замкнутость классов сложности языков относительно дополнения для детерминированных МТ и отсутствие этого свойства для НМТ.

т. е. для любого класса $\mathcal{DTIME}(f(n))$ класс

$$\text{co}\mathcal{DTIME}(f(n)) \equiv \{L | \bar{L} \in \mathcal{DTIME}(f(n))\},$$

состоящий из языков-дополнений⁷, совпадает с $\mathcal{DTIME}(f(n))$. Действительно, для любого языка $L \in \mathcal{DTIME}(f(n))$ существует детерминированная машина Тьюринга Т с временной сложностью $f(n)$, и из нее, инвертировав окончательный ответ, получаем машину \bar{T} , распознающую \bar{L} и также принадлежащую классу $\mathcal{DTIME}(f(n))$. Аналогично показывается замкнутость относительно дополнений и классов пространственной сложности детерминированных машин Тьюринга — $\mathcal{DSPACE}(f(n))$.

Для классов $\mathcal{NTIME}(f(n))$ и $\mathcal{NSPACE}(f(n))$ эти рассуждения не проходят из-за асимметрии положительных и отрицательных ответов в определении сложности НМТ, и, таким образом, нельзя утверждать о совпадении классов $\mathcal{NTIME}(f(n))$ и $\text{co}\mathcal{NTIME}(f(n))$, $\mathcal{NSPACE}(f(n))$ и $\text{co}\mathcal{NSPACE}(f(n))$.

Теперь определим важный в теории сложности класс задач разрешения (языков), разрешимых полиномиальными недетерминированными алгоритмами и его со-класс (класс-дополнение).

Определение 6.2.3.

$$\begin{aligned}\mathcal{NP} &= \cup_{k \geq 0} \mathcal{NTIME}(n^k), \\ \text{co}\mathcal{NP} &= \{L | \bar{L} \in \mathcal{NP}\}.\end{aligned}$$

⁷Определение дополнения языка можно найти в разделе 7.2.

Упражнение 6.2.1. Покажите, что $\mathcal{P} \subseteq \mathcal{NP} \cap \text{co}\mathcal{NP}$.

Следует подчеркнуть, что недетерминированные алгоритмы представляют собой лишь некоторую довольно удобную математическую абстракцию и, в отличие от детерминированных и вероятностных алгоритмов, не соответствуют реально существующим вычислительным или аналоговым машинам.

В частности, недетерминированные полиномиальные алгоритмы, вообще говоря, не являются эффективными, хотя и известно включение $\mathcal{NP} \subseteq \mathcal{PSPACE}$. Далее, все переборные задачи лежат в классе \mathcal{NP} . Например, задача 4 «TSP» в форме задачи разрешения (задача 27 «TSP-разрешимость») решается недетерминированным полиномиальным алгоритмом, который

1. недетерминировано генерирует последовательность городов для посещения (не обязательно цикл);
2. затем детерминировано проверяет:
 - является ли полученный маршрут гамильтоновым циклом;
 - правда ли, что стоимость маршрута меньше B ;
3. только в случае успешности обеих проверок возвращает «1», иначе — «0».

Справедливо и обратное: вычисление с помощью любого недетерминированного полиномиального алгоритма можно представить как переборную задачу (перебор ведется по всем двоичным словам, кодирующими направления ветвлений в недетерминированных операторах перехода). Тем самым, «задача из класса \mathcal{NP} » оказывается адекватным математическим уточнением понятия «переборная задача разрешения».

Более формально это формулируется в одном из альтернативных определений класса \mathcal{NP} через детерминированные машины Тьюринга.

Определение 6.2.4. Язык $L \subseteq \Sigma^*$ принадлежит классу \mathcal{NP} , если существуют полиномиальная детерминированная машина Тьюринга M и полином $p(\cdot)$, такие, что

$$L = \{x \in \Sigma^* : \exists y, |y| < p(|x|) \& M(x, y) = 1\}.$$

Слово y называется обычно «подсказкой», «свидетелем» (\mathcal{NP} -witness), «доказательством» (\mathcal{NP} -proof).

Например, пусть L описывает задачу 4 «TSP» в форме задачи разрешения 27 «TSP-разрешимость».

Тогда слово x представляет собой матрицу расстояний между городами и B — денежное ограничение затрат на посещение всех городов, а слово y будет представлять объезд всех городов с затратами $< B$. Проверить, что путь y при параметрах задачи x действительно является решением, можно полиномиальным алгоритмом. С другой стороны, если для задачи с некоторыми параметрами x' такого пути ($< B$) нет, то, соответственно, такой подсказки $y(x')$ не существует.

Существует несколько популярных определений класса \mathcal{NP} через сценарии совместного принятия решения двумя сторонами:

1. Честным, преследующим цель установить истину, но не гениальным, т. е. полиномиально ограниченным проверяющим (человеком, королем Артуром). Этую роль обычно обозначают *verifier*.
2. Всемогущим консультантом (магом Мерлином, оракулом), который в силах предложить правильное решение, но может предложить неправильное (например, преследуя свои интересы или любируя чужие). Этую роль обычно обозначают *oracle* или *prover*.

Соответственно, класс \mathcal{NP} — это класс языков, разрешимых такой командой, в которой оракул предлагает решения, но только будучи заверенными проверяющим, они приобретают «юридическую» силу (оракул может и обмануть).

Обратите внимание на следующие моменты:

1. полиномиальную ограниченность подсказки, — полиномиальности проверяющего алгоритма A недостаточно, т.к. он полиномиален относительно длины своих аргументов x и y . т. е. краткость подсказки важный момент — подсказки, вроде «а проверь все маршруты», которые, безусловно, можно проверить за полиномиальное время от длины описания «всех маршрутов», не подойдут, если длина описания этого множества маршрутов экспоненциальна;
2. если элемент не принадлежит языку, то подсказки с указанными свойствами не должно существовать, т. е. ничто не заставит проверяющего ошибиться.

Через определение 6.2.4 « $\mathcal{NP}/\text{ДМТ}$ » еще легче увидеть «переборную» сущность класса \mathcal{NP} , т.к. видно, что можно перебирать детерминированным алгоритмом множество вариантов, эффективно (полиномиальным алгоритмом) проверяя каждый вариант.

Теорема 28. Определение 6.2.3 « $\mathcal{NP}/\text{НМТ}$ » и определение 6.2.4 « $\mathcal{NP}/\text{ДМТ}$ » эквивалентны.

Доказательство. 6.2.3 « $\mathcal{NP}/\text{НМТ}$ » \rightarrow 6.2.4 « $\mathcal{NP}/\text{ДМТ}$ ». Для каждого слова $x \in L$ подсказкой y можно назначить закодированный кратчайший протокол выполнения-подтверждения НМТ Т, определяющей язык L , в смысле определения 6.2.3 « $\mathcal{NP}/\text{НМТ}$ ». Так как длина кратчайшего пути-подтверждения $T(x)$ полиномиально ограничена, то можно выбрать полиномиально ограниченную кодировку $y(x)$. Проверить целостность протокола (отсутствие противоречий с определением НМТ Т) $y(x)$ можно также за полиномиальное время.

6.2.4 « $\mathcal{NP}/\text{ДМТ}$ » \rightarrow 6.2.3 « $\mathcal{NP}/\text{НМТ}$ ». Еще проще. Пусть НМТ Т недетерминировано дописывает разделитель # и некоторое слово u к входному слову x , а затем работает над словом $x\#u$ как полиномиальная детерминированная машина Тьюринга из определения 6.2.4 « $\mathcal{NP}/\text{ДМТ}$ ». □

6.2.3 Сводимость по Карпу

Понятие полиномиальной сводимости по Куку⁸ оказывается чрезмерно общим для изучения переборных задач ввиду отмеченной выше асимметрии ответов 1 и 0.

Например, задача разрешения $\langle \text{Comm}(m, d) > B \rangle$ (в которой спрашивается «верно ли, что *любой* маршрут коммивояжера имеет длину по крайней мере $(B + 1)$?») принадлежит классу $\text{co}\mathcal{NP}$ и не принадлежит классу \mathcal{NP} при общепринятой гипотезе $\mathcal{P} \neq \mathcal{NP}$. В то же время она очевидным образом сводится по Куку к переборной задаче 27 «TSP-разрешимость», принадлежащей классу \mathcal{NP} .

Поэтому в теории сложности вычислений гораздо большее распространение получил ограниченный вариант полиномиальной сводимости по Куку, впервые рассмотренный в основополагающей работе [Kap75] и, соответственно, называемый *сводимостью по Карпу* (мы будем использовать для нее просто термин *полиномиальная сводимость*, также широко распространенный в литературе).

Определение 6.2.5. Задача разрешения P_1 **полиномиально сводится** к задаче разрешения P_2 , если

- существует полиномиально вычислимая функция $f : I_1 \rightarrow I_2$, (отображает входные данные I_1 для P_1 во входные данные $I_2 \equiv f(I_1)$ для задачи P_2),
- $\forall I_1$ совпадают ответы на вопросы « $P_1(I_1) ?$ » и « $P_2(f(I_1)) ?$ ».

Легко видеть, что относительно такого усеченного варианта полиномиальной сводимости класс переборных задач разрешения \mathcal{NP} уже оказывается замкнутым.

Фундаментальной открытой проблемой теории сложности вычислений (а к настоящему времени и одной из наиболее фундаментальных проблем всей современной математики — [Sma00]) является вопрос о совпадении классов сложности P и NP . Иными словами, все ли переборные задачи можно решить с

⁸см. определение 6.2.1 «Сводимость по Куку».

помощью эффективного алгоритма? В связи с этой емкой формулировкой $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ -проблему еще называют иногда *проблемой перебора*.

На первый взгляд $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ -проблема, по существу, представляет собой совокупность похожих, но формально между собой не связанных вопросов о том, поддается ли эффективному решению *данная конкретная* переборная задача. Тем не менее, оказывается, что рассмотренное выше понятие полиномиальной сводимости позволяет во многих случаях установить, что эти вопросы для целого ряда задач эквивалентны между собой, а также эквивалентны «глобальной» задаче $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$.

Определение 6.2.6. Задача разрешения называется \mathcal{NP} -полной⁹, если

- она принадлежит классу \mathcal{NP} ,
- произвольная задача из \mathcal{NP} сводится к ней полиномиально (См. определение 6.2.5 «Сводимость по Карпу»).

Класс \mathcal{NP} -полных задач обозначается \mathcal{NPC} .

Оставляя на секунду в стороне неочевидный *a priori* вопрос о существовании хотя бы одной \mathcal{NPC} -задачи, заметим, что они в точности обладают нужным нам свойством: $\mathcal{P} = \mathcal{NP}$ тогда и только тогда, когда некоторую \mathcal{NP} -полную задачу можно решить с помощью полиномиального алгоритма. Иными словами, \mathcal{NP} -полные задачи выполняют роль наиболее сложных, универсальных задач в классе \mathcal{NP} , и все они по своей сложности эквивалентны между собой.

Обратите внимание, что в определение задачи \mathcal{NPC} обязательно входит принадлежность классу \mathcal{NP} — если опустить это условие, получится класс \mathcal{NP} -трудных (\mathcal{NP} -hard) задач, включающих \mathcal{NP} , но выходящих за границы класса \mathcal{NP} (при гипотезе $\mathcal{P} \neq \mathcal{NP}$).

⁹ Чтобы не перегружать лекции излишней терминологией, мы будем называть в дальнейшем оптимизационную задачу \mathcal{NP} -полной, если \mathcal{NP} -полнна соответствующая задача разрешения.

Историю современной теории сложности вычислений принято отсчитывать с работ [Кук75; Кар75], в которых были заложены основы теории \mathcal{NP} -полноты и доказано существование вначале одной, а затем (в работе [Кар75]) достаточно большого числа (а именно, 21) естественных \mathcal{NP} -полных задач. В неоднократно цитировавшейся монографии [ГД82] (вышедшей на английском языке в 1979 г.) приводится список известных к тому времени \mathcal{NP} -полных задач, насчитывающий уже более 300 наименований. К настоящему времени количество известных \mathcal{NP} -полных задач выражается четырехзначным числом, и постоянно появляются новые, возникающие как в самой математике и теории сложности, так и в таких дисциплинах, как биология, социология, военное дело, теория расписаний, теория игр и т. д. (см. [07; СК]). Более того, как мы уже отмечали в разделе 1.1, для подавляющего большинства задач из класса \mathcal{NP} в конечном итоге удается либо установить их принадлежность классу \mathcal{P} (т. е. найти полиномиальный алгоритм), либо доказать \mathcal{NP} -полноту. Одним из наиболее важных исключений являются задачи типа дискретного логарифма и факторизации, на которых основаны многие современные криптоватоколы.

Именно этим обстоятельством объясняется важность проблемы $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$. Безуспешным попыткам построения полиномиальных алгоритмов для \mathcal{NP} -полных задач были посвящены усилия огромного числа выдающихся специалистов в данной области (в конце раздела мы вкратце расскажем о некоторых частичных результатах, полученных в обратном направлении, т. е. попытках доказать неравенство $\mathcal{P} \neq \mathcal{NP}$). Ввиду этого можно считать, что \mathcal{NP} -полные задачи являются *труднорешаемыми* со всех практических точек зрения, хотя, повторяем, строгое доказательство этого составляет одну из центральных открытых проблем современной математики.

Прежде всего необходимо иметь хотя бы одну \mathcal{NP} -полную задачу. Честь быть первой выпала задаче 16 «SAT», «открытой» в [Кук75].

Теорема 29. Задача 16 «SAT» — \mathcal{NP} -полнна.

Доказательство. Очевидно, задача 16 «SAT» принадлежит \mathcal{NP} , так как для любого слова x , представляющего выполнимую входную КНФ, существует подсказка y — значения переменных, при которых эта КНФ

выполняется, причем проверку легко выполнить за полиномиальное время, используя память не больше полиномиального объема.

Рассмотрим произвольный язык $L \in \mathcal{NP}$. Согласно определению 6.2.4 « $\mathcal{NP}/\text{ДМТ}$ »,

$$\forall x \in L, \exists y(x) : |y(x)| < \text{poly}(|x|),$$

и существует МТ M , распознающая $L_y = \{x \# y(x) \mid x \in L\}$ за полиномиальное время.

Рассмотрим таблицу вычисления (см. доказательство теоремы 42) для M .

Будем использовать те же переменные, что и в доказательстве теоремы 42 (коды состояний клеток таблицы вычисления). Чтобы таблица вычисления соответствовала правильно проведённому успешному (с ответом 1) вычислению, должны выполняться локальные правила согласования для каждой четвёрки клеток вида , и результат должен быть 1. Каждое такое правило задаётся формулой от переменных, отвечающих либо рассматриваемой четвёрке, либо нулевой ячейке самой нижней строки таблицы. Определим формулу φ_x как конъюнкцию всех этих формул, в которые подставлены значения переменных, кодирующих вход $x \# y$, дополненный символами $*$ до длины $|x| + 1 + q(|x|)$. Значения, соответствующие x и $\#$, — константы, поэтому переменные, от которых зависит эта формула, отвечают y и кодам внутренних ячеек таблицы. Так что можно считать, что формула φ_x зависит от y и ещё от каких-то переменных, которые мы обозначим z .

Итак, мы сопоставили слову x формулу $\varphi_x(y, z)$, которая по построению обладает следующим свойством. Если M принимает $x \# y$, то найдётся такой набор значений $z(x, y)$, при котором $\varphi_x(y, z(x, y))$ истинна (эти значения описывают работу M на входе $x \# y$). А если M не распознает $x \# y$, то $\varphi_x(y, z)$ всегда ложна (поскольку, по сути, утверждает, что вычисление на входе (x, y) даёт ответ 1).

□

Все остальные доказательства \mathcal{NP} -полноты (а уже следующая работа [Кар75] содержала 21 такое доказательство) основаны на следующем легко проверяемом замечании: если \mathcal{NP} -полнная задача P_1 по-

линомиально сводится к переборной задаче P_2 , то P_2 также \mathcal{NP} -полна. Поэтому общие рекомендации состоят в том, чтобы выбрать в списке уже известных \mathcal{NP} -полных задач «максимально похожую» и попытаться свести ее к интересующей нас задаче. Актуальные списки и каталоги \mathcal{NP} -полных задач (в дополнение к классическому труду [ГД82]) можно найти в [07; СК].

К сожалению, о том, как именно применять эту общую рекомендацию в каждом конкретном случае, что-либо определенное сказать довольно трудно. Доказательства \mathcal{NP} -полноты являются скорее искусством, и мы от всей души желаем нашим читателям больше полиномиальных алгоритмов, хороших и разных, для интересующих их задач, чтобы обращаться к этому специальному искусству им пришлось как можно реже.

В случае, если такая необходимость все же возникнет, мы рекомендуем обратиться к прекрасно написанной третьей главе монографии [ГД82], по которой можно ознакомиться, по крайней мере, с некоторыми ориентирами на этом пути.

Здесь же мы ограничимся простым примером.

Задача 28. «3-Выполнимость/3SAT».

Вариант задачи 16 «SAT», где каждая элементарная дизъюнкция (3.4) имеет длину $k \leq 3$. Соответствующие КНФ называются 3-КНФ.

Оказывается, задача 16 «SAT» сводится к своему ограниченному варианту 28 «3SAT». В самом деле, если дана некоторая КНФ (3.3), мы заменяем в ней каждую элементарную дизъюнкцию (3.4) с $k > 3$ на следующее булевское выражение:

$$(y_{i2} \equiv (x_{j_1}^{\sigma_1} \vee x_{j_2}^{\sigma_2})) \wedge (y_{i3} \equiv (y_{i2} \vee x_{j_3}^{\sigma_3})) \wedge \dots \wedge (y_{ik} \equiv (y_{i,k-1} \vee x_{j_k}^{\sigma_k})) \wedge y_{ik},$$

где y_{i2}, \dots, y_{ik} — новые булевые переменные, и трансформируем эквивалентности $z_1 \equiv z_2 \vee z_3$ в 3-КНФ стандартным способом. Во всяком выполняющем наборе для полученной таким образом 3-КНФ переменная y_{iv} обязательно должна принять значение $(x_{j_1}^{\sigma_1} \vee \dots \vee x_{j_\nu}^{\sigma_\nu})$ и, в частности, y_{ik} получает значение (3.4). Поэтому наше преобразование определяет полиномиальную сводимость задачи 16 «SAT» к задаче 28 «3SAT», и последняя тем самым оказывается \mathcal{NP} -полной.

Упражнение 6.2.2. Выразите логическое отношение эквивалентности в виде 3-КНФ формулы.

Упражнение 6.2.3. Рассмотрим ограниченную версию задачи 28 «3SAT», где на КНФ-формулу дополнительно наложено ограничение, что в ней каждая переменная может входить не больше трех раз, причем каждый литерал — не больше двух.

Покажите, что и эта задача \mathcal{NP} -полна.

Задача 29. «2-Выполнимость»(2SAT). Частный случай задачи 16 «SAT», в котором каждая элементарная дизъюнкция имеет длину $k \leq 2$. Соответствующие КНФ называются 2-КНФ.

Упражнение 6.2.4. Покажите, что задача 29 «2SAT» лежит в \mathcal{P} .

Упражнение 6.2.5. Рассмотрим максимизационную версию задачи 29 «2SAT», где дополнительно к 2SAT-формуле задается параметр K , и спрашивается, можно ли выполнить больше чем K , количество скобок.

Покажите, что эта задача лежит в \mathcal{NPC} .

Теперь рассмотрим специальный случай общей задачи о покрытии — 30 и покажем, что даже она \mathcal{NP} -полна.

Задача 30. «Вершинное покрытие»¹⁰.

¹⁰В англоязычной литературе — *Vertex Covering*.

Дан граф $G = (V, E)$ и положительное целое число K , $K \leq |V|$.

Имеется ли в графе G **вершинное покрытие** не более чем из K элементов, т. е. такое подмножество $V' \subseteq V$, что $|V'| \leq K$ и каждое ребро из E содержит хотя бы одну вершину из V' ?

Лемма 34. Задача 30 «Vertex Covering» лежит в \mathcal{NPC} .

Доказательство. Мы покажем, что задача 28 «3SAT» полиномиально сводится к задаче 30 «Vertex Covering».

Пусть имеется 3-КНФ (3.3) от n переменных x_1, \dots, x_n , в которой (без существенного ограничения общности) можно считать, что $k = 3$ для всех m элементарных дизъюнкций (3.4). Для каждой переменной x_j мы вводим отдельное ребро $(x_j, \neg x_j)$, а для каждой элементарной дизъюнкции (3.4) — треугольник с вершинами (v_{i1}, v_{i2}, v_{i3}) , после чего соединяем ребрами v_{i1} с $x_{j_1}^{\sigma_1}$, v_{i2} — с $x_{j_2}^{\sigma_2}$ и v_{i3} — с $x_{j_3}^{\sigma_3}$. На рисунке 6.8 приведен пример небольшой (всего три дизъюнкции) выполнимой ЗКНФ-формулы, отображенной указанным способом на графике.

В полученном графе (безотносительно к исходной ЗКНФ) всякое вершинное покрытие должно иметь размер не менее $(n + 2m)$ (по крайней мере нужно n вершин, чтобы покрыть ребра $(x_j, \neg x_j)$ и необходимо не меньше $2m$ вершин для покрытия треугольников (v_{i1}, v_{i2}, v_{i3})).

С другой стороны, если есть выполняющий набор для ЗКНФ, то существует вершинное покрытие размера $(n + 2m)$.

В вершинное покрытие из пар вершин $(x_j, \neg x_j)$ мы включаем вершину x_j , если в выполняющем наборе $x_j = 1$, и $\neg x_j$ в противном случае. Тогда, т.к. в каждой дизъюнкции у нас есть по крайней мере один истинный терм, то в каждом треугольнике должна быть хотя бы одна вершина, достигаемая одним ребром от какого-либо x_j или $\neg x_j$, а остальные две вершины мы включаем в вершинное покрытие.

Итак, мы убедились, что покрытия $(n + 2m)$ в точности соответствуют выполняющим наборам исходной ЗКНФ, и если положить параметр K в формулировке задачи 30 «Vertex Covering» равным $(n + 2m)$, то получим искомое полиномиальное сведение. \square

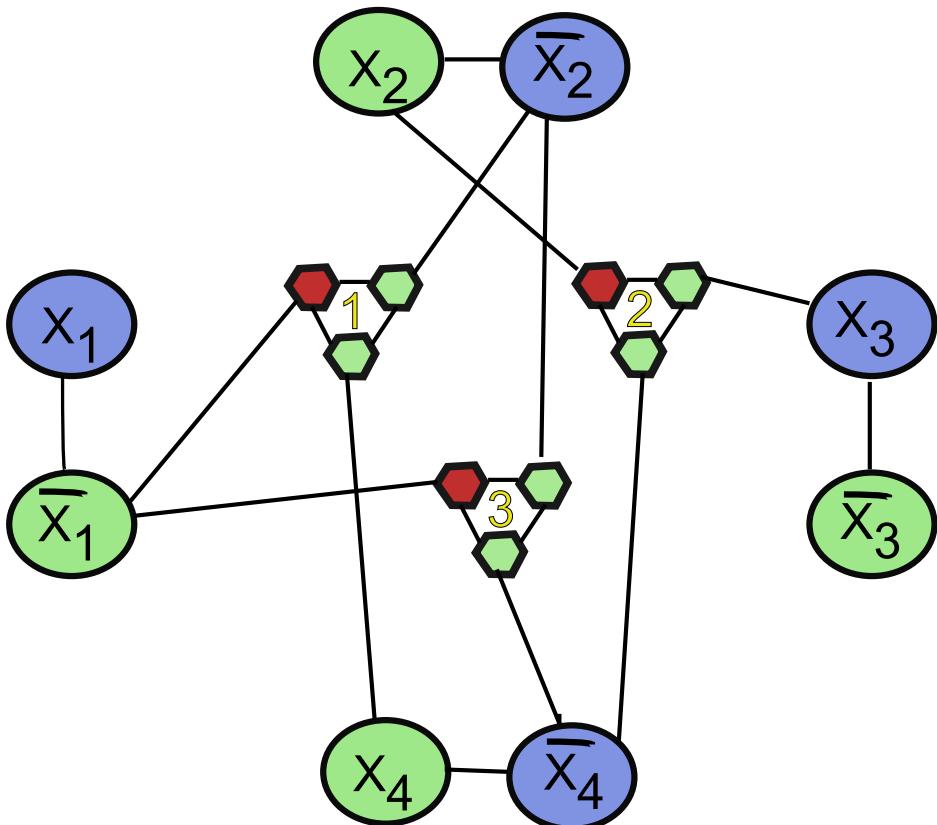


Рис. 6.8: Отображение выполнимой КНФ на граф

В настоящее время «экспертное мнение» склоняется к тому, что скорее всего $\mathcal{P} \neq \mathcal{NP}$. Таким образом, можно представить общепринятое представление об отношениях классов \mathcal{NP} , со \mathcal{NP} , \mathcal{P} , \mathcal{NPC} на рис. 6.9.

Для того чтобы доказать этот факт, необходимо научиться получать *нижние оценки* сложности любого алгоритма, предназначенного для решения некоторой задачи из класса \mathcal{NP} .

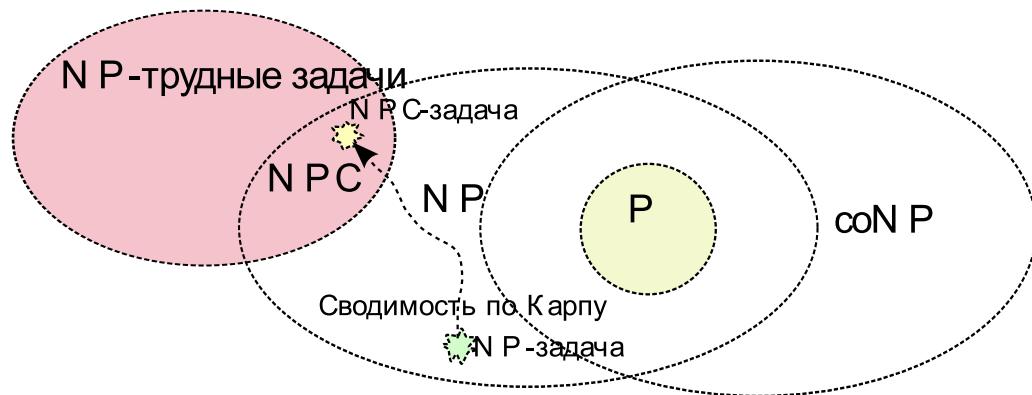
Соответствующая проблема так и называется *проблемой нижних оценок*. Мы уже видели ранее (и познакомимся с массой таких примеров в дальнейшем), что для самых разнообразных задач возможны весьма неожиданные алгоритмы, основанные на самых различных идеях. Для доказательства же нижних оценок необходимо найти некоторое *общее рассуждение*, которое учитывало бы все такие алгоритмы, как уже существующие, так и те, которые могут быть построены в будущем.

Поэтому проблема нижних оценок является крайне трудной, и решена она лишь в довольно частных случаях.

Традиционно эта проблема в подавляющем большинстве случаев рассматривается в контексте *схемной* (или *булевой*) сложности (см. раздел 6.4) ввиду исключительной внешней простоты и наглядности схемной модели. Из отмеченной взаимосвязи между алгоритмами и схемами вытекает, что проблема нижних оценок для последних должна быть столь же трудной, как и для алгоритмов.

Действительно, наилучшая известная оценка сложности схем для какой-либо задачи из класса \mathcal{NP} всего лишь линейна от числа переменных n и составляет $5n - o(n)$ (см. [IM02]).

Гораздо больших успехов в получении нижних оценок сложности удалось добиться для схем с различными ограничениями. К числу таких ограниченных моделей относятся, например, *монотонные схемы* (в которых запрещаются элементы *NOT*) или *схемы ограниченной глубины* (в которых вдоль любого пути число чередований элементов различных типов ограничено сверху произвольно большой, но фиксированной заранее константой). Для таких схем проблема нижних оценок практически полностью решена: см., например, [Раз85a; Раз85b].



С другой стороны, работа над верхними оценками — построение конкретных алгоритмов, как правило, производится в терминах машин с произвольным доступом (*RAM*).

Упражнение 6.2.6. Покажите, что задача распознавания гамильтоновых графов (т. е. графов, содержащих гамильтонов цикл) принадлежит \mathcal{NP} , а задача распознавания негамильтоновых графов принадлежит $\text{co}\mathcal{NP}$.

Упражнение 6.2.7. Придумайте полиномиальный алгоритм для проверки, есть ли в заданном графе хотя бы один «треугольник».

Упражнение 6.2.8. Рассмотрим язык $DFNT$, состоящий из полиномов от нескольких переменных, имеющих целочисленные корни.

Студент утверждает, что $DFNT \in NP$, т.к. если оракул-Мерлин предоставит решение v , доказывающее принадлежность полинома $p \in DFNT$, то верификатор Артур сможет легко проверить: $p(v) \stackrel{?}{=} 0$.

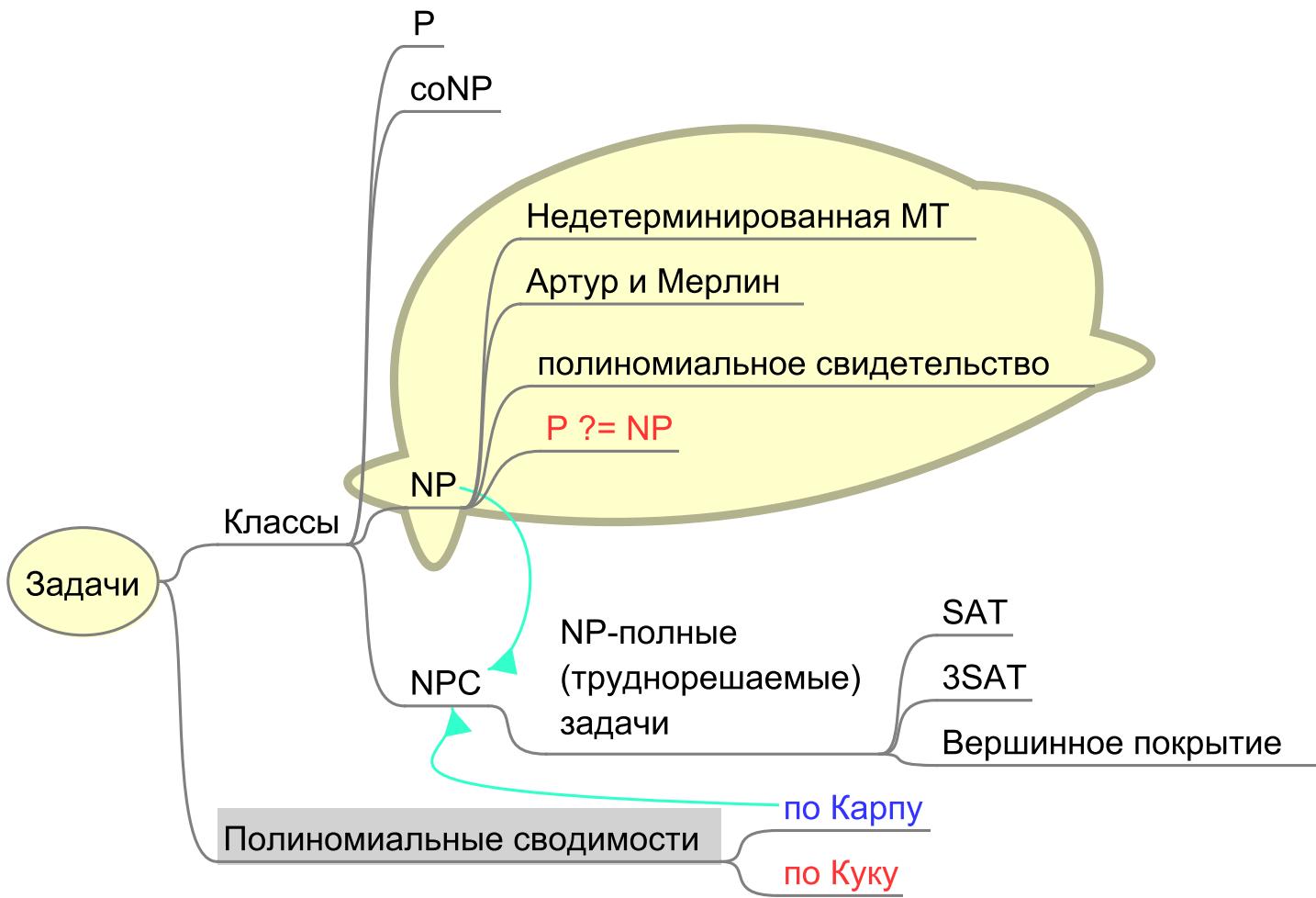
Прав ли студент?

6.3 Вероятностные вычисления



Вероятностные алгоритмы с односторонней ошибкой. Классы сложности \mathcal{RP} и $\text{co}\mathcal{RP}$ и отношение к классам \mathcal{NP} и $\text{co}\mathcal{NP}$. Вероятностная амплификация для \mathcal{RP} и $\text{co}\mathcal{RP}$. Вероятностные алгоритмы с двусторонней ошибкой. Класс сложности \mathcal{BPP} . Вероятностная амплификация для \mathcal{BPP} . Неамплифицируемый класс \mathcal{PP} .

Итак, в разделе 6.1.1 мы познакомились с детерминированными машинами Тьюринга, моделями, ко-



торые можно использовать для описания всех существующих вычислительных устройств, будь то карман-ный калькулятор или суперкомпьютер. В разделе [6.2.2](#) мы рассматривали недетерминированные машины Тьюринга — интересную, мощную модель вычислений, полезную для описания классов сложностей задач, но, увы, не соответствующую никаким реальным вычислительным устройствам. Однако оказалось, что можно частично использовать мощь «параллельного перебора», присущую НМТ, привнеся в детерминированный процесс вычисления вероятностную составляющую. Выяснилось, что такие устройства вполне можно построить физически, и что они способны эффективно решать больший класс задач, чем обыкновенные МТ.

Впрочем, по-порядку. Сначала определим ВМТ — вероятностную машину Тьюринга. Существует два подхода к определению ВМТ, приведем их оба.

Определение 6.3.1. *Вероятностная машина Тьюринга* аналогична недетерминированной машине Тьюринга, только вместо недетерминированного перехода в два состояния машина выбирает один из вариантов с равной вероятностью.

Определение 6.3.2. *Вероятностная машина Тьюринга* представляет собой детерминированную машину Тьюринга (см. определение [6.1.1](#) «Машина Тьюринга»), имеющую дополнительно источник случайных битов, любое число которых, например, она может «заказать» и «загрузить» на отдельную ленту и потом использовать в вычислениях обычным для МТ образом.

Оба определения — [6.3.1](#) «online ВМТ» и [6.3.2](#) «offline ВМТ» — очевидно, эквивалентны. Более того, они вполне соответствуют обычному компьютеру, к которому подключен внешний генератор случайных последовательностей на основе случайных физических процессов, таких, как, например, распад долгоживущего радиоактивного изотопа (замер времени между щелчками счетчика Гейгера рядом с образцом изотопа рубидия-85), снятие параметров с нестабильных электрических цепей и т. п.

Привнесение случайности в детерминированную вычислительную модель (как и привнесение недетерминизма) модифицирует понятие разрешимости. Теперь, говоря о результате работы ВМТ M на входе x , мы можем говорить о математическом ожидании результата $E[M(x)]$ или вероятности вывода того или иного ответа на заданном входе: $P[M(x) = 1]$, $P[M(x) = 0]$, но ограничение на время работы (как правило, полиномиальное) понимается так же, как и для ДМТ.

Вообще, есть вероятностные алгоритмы, не совершающие ошибок при вычислении. Они используют «вероятностную составляющую» для «усреднения» своего поведения на различных входных данных таким образом, чтобы избежать случая, когда часто встречаются «плохие» входные наборы. Такие алгоритмы иногда называют «шервудскими», в честь известного разбойника-перераспределителя ценностей, и мы уже рассматривали один из таких алгоритмов — алгоритм 12 «Quicksort» с вероятностным выбором оси.

ВМТ может быть полезна, как мы увидим дальше, даже если будет иногда ошибаться при вычислении некоторой функции. Разумеется, нас будут интересовать машины, ошибающиеся «нечасто» и, желательно, работающие эффективно, т. е. полиномиальное от длины входа время.

Мы сконцентрируемся на анализе ВМТ, применяющихся для задач разрешения, т. е. задач, в которых нужно получить ответ «0» или «1».

Рассматривая ВМТ, предназначенные для решения таких задач, можно их классифицировать на три основные группы:

1. «zero-error» — ВМТ, не допускающие ошибок. Правда, в этот же класс попадают алгоритмы, которые хоть и не допускают ошибок, могут и не выдать никакого ответа или отвечать «не знаю».
2. «one-sided error» — ВМТ с односторонними ошибками. Например, ВМТ-«автосигнализация» обязательно сработает, если стекло разбито, дверь открыта и происходит автоугон, но могут произойти

и ложные срабатывания (так называемые ошибки второго рода, напомним также, что ошибкой первого рода является «пропуск цели»).

3. «two-sided error» — ВМТ с двусторонними ошибками. Ошибки могут быть и первого, и второго рода, но вероятность правильного ответа должна быть больше (чем больше, тем лучше) $\frac{1}{2}$, иначе эта ВМТ ничем не лучше обычной монетки.

Далее мы более подробно рассмотрим перечисленные классы ВМТ и классы языков, эффективно (т. е. полиномиально) распознаваемых ими.

6.3.1 Классы $\mathcal{RP}/\text{co}\mathcal{RP}$. «Односторонние ошибки»

Итак, определим классы языков, эффективно распознаваемых на ВМТ с односторонней ошибкой.

Определение 6.3.3. Класс сложности \mathcal{RP} (*Random Polynomial-time*) состоит из всех языков L , для которых существует полиномиальная ВМТ M , такая, что:

$$\begin{aligned} x \in L &\Rightarrow P[M(x) = 1] \geq \frac{1}{2}, \\ x \notin L &\Rightarrow P[M(x) = 0] = 1. \end{aligned}$$

Сразу же представим аналогичное определение для класса-дополнения $\text{co}\mathcal{RP} \equiv \{L | \bar{L} \in \mathcal{RP}\}$.

Определение 6.3.4. Класс сложности $\text{co}\mathcal{RP}$ (*Complementary Random Polynomial-time*) состоит из всех языков L , для которых существует полиномиальная ВМТ M , такая, что:

$$\begin{aligned} x \in L &\Rightarrow P[\mathbf{M}(x) = 1] = 1, \\ x \notin L &\Rightarrow P[\mathbf{M}(x) = 0] \geq \frac{1}{2}. \end{aligned}$$

Попробуем «популярно представить» языки и ВМТ, упомянутые в этих определениях. Например, для определения 6.3.3 « \mathcal{RP} » можно представить автоматическую систему биометрической идентификации на входе в военный бункер, никогда не пускающую солдат противника ($x \notin L$), и с вероятностью $\geq \frac{1}{2}$ пускающей своих ($x \in L$), правда, возможно, своим потребуется сделать несколько попыток прохода.

Для определения 6.3.4 « $\text{co}\mathcal{RP}$ » можно представить автоматическую автомобильную сигнализацию, гарантированно подающую сигнал при попытке угона ($x \in L$), и с вероятностью $\geq \frac{1}{2}$ не поднимающую ложную тревогу, если происходит что-то другое.

Действительно, первое условие из определения 6.3.4 « $\text{co}\mathcal{RP}$ » и второе условие из определения 6.3.3 « \mathcal{RP} » называют «completeness»-условиями¹¹, а первое условие из определения 6.3.3 « \mathcal{RP} » и второе условие из определения 6.3.4 « $\text{co}\mathcal{RP}$ » — «soundness»-условиями¹². Для тех, кто считает, что сигнализация с вероятностью ложного срабатывания чуть меньше $\frac{1}{2}$ не отличается здравостью и ни на что не годна, просим немного подождать.

Вспомним, что в «offline»-определении ВМТ (См. определение 6.3.2 «offline ВМТ») подразумевается отделенность вероятностных данных от обычной ДМТ. Полиномиальная ВМТ делает не более чем полиномиальное (от длины входа) число переходов, и, следовательно, моделирующая ее ДМТ нуждается не более чем в полиномиальной строке случайных битов. Теперь перепишем еще раз определения классов \mathcal{RP} и $\text{co}\mathcal{RP}$, используя только ДМТ. Выделим элемент случайности в строку y , причем $|y| \leq p(|x|)$, где

¹¹completeness — полнота.

¹²soundness — корректность.

$p(\cdot)$ — некий полином, и заменим условия на вероятность условиями на долю строк y , на которых ДМТ дает тот или иной результат.

Определение 6.3.5. Класс сложности \mathcal{RP} состоит из всех языков L , для которых существуют некий полином $p(\cdot)$ и полиномиальная МТ $M(x, y)$, такая, что:

$$\begin{aligned} x \in L &\Rightarrow \frac{|\{y : M(x, y) = 1, |y| \leq p(|x|)\}|}{2^{p(|x|)}} \geq \frac{1}{2}, \\ x \notin L &\Rightarrow \forall y \quad M(x, y) = 0. \end{aligned}$$

Определение 6.3.6. Класс сложности $\text{co}\mathcal{RP}$ состоит из всех языков L , для которых существуют некий полином $p(\cdot)$ и полиномиальная МТ $M(x, y)$, такая, что:

$$\begin{aligned} x \in L &\Rightarrow \forall y \quad M(x, y) = 1, \\ x \notin L &\Rightarrow \frac{|\{y : M(x, y) = 0, |y| \leq p(|x|)\}|}{2^{p(|x|)}} \geq \frac{1}{2}. \end{aligned}$$

Теперь вспомним определение класса \mathcal{NP} через детерминированную машину Тьюринга (определение 6.2.4 « $\mathcal{NP}/\text{ДМТ}$ ») и сравним его с определением 6.3.5 « $\mathcal{RP}/\text{ДМТ}$ », опуская одинаковые описания M , y :

\mathcal{NP}	\mathcal{RP}
$x \in L \Rightarrow \exists y, M(x, y) = 1$	$x \in L \Rightarrow$ доля $y: M(x, y) = 1 \geq \frac{1}{2}$
$x \notin L \Rightarrow \forall y, M(x, y) = 0$	$x \notin L \Rightarrow \forall y, M(x, y) = 0$

Из этого сравнения видно, что класс \mathcal{RP} вложен в класс \mathcal{NP} . Если для распознавания языка L из \mathcal{NP} нам для каждого $x \in L$ нужен был хотя бы один полиномиальный «свидетель» y , который мы могли предъявить «верификатору», то в случае с классом \mathcal{RP} этих свидетелей должно быть достаточно много, согласно определению 6.3.5 « $\mathcal{RP}/\text{ДМТ}$ » не меньше половины. На самом деле, как мы увидим дальше, этих свидетелей может быть и меньше, а пока очевидны следующие утверждения (см. рис. 6.11):

Теорема 30. $\mathcal{RP} \subseteq \mathcal{NP}$.

Теорема 31. $\text{co}\mathcal{RP} \subseteq \text{co}\mathcal{NP}$.

Теперь вернемся к пресловутой «одной второй». Рассмотрим некоторый класс \tilde{C} .

Определение 6.3.7. Класс сложности \tilde{C} состоит из всех языков L , для которых существует полиномиальная ВМТ \tilde{M} , такая, что:

$$\begin{aligned} x \in L &\Rightarrow P[\tilde{M}(x) = 1] \geq \frac{3}{4}, \\ x \notin L &\Rightarrow P[\tilde{M}(x) = 1] = 0. \end{aligned}$$

Очевидно, т.к. с виду налагаемые условия кажутся более (и уж точно не менее) жесткими, $\tilde{C} \subseteq \mathcal{RP}$. Можем ли мы утверждать, что $\mathcal{RP} \subseteq \tilde{C}$ и, следовательно, $\tilde{C} \equiv \mathcal{RP}$?

Оказывается, да.

Стрелки показывают вложенность классов сложности

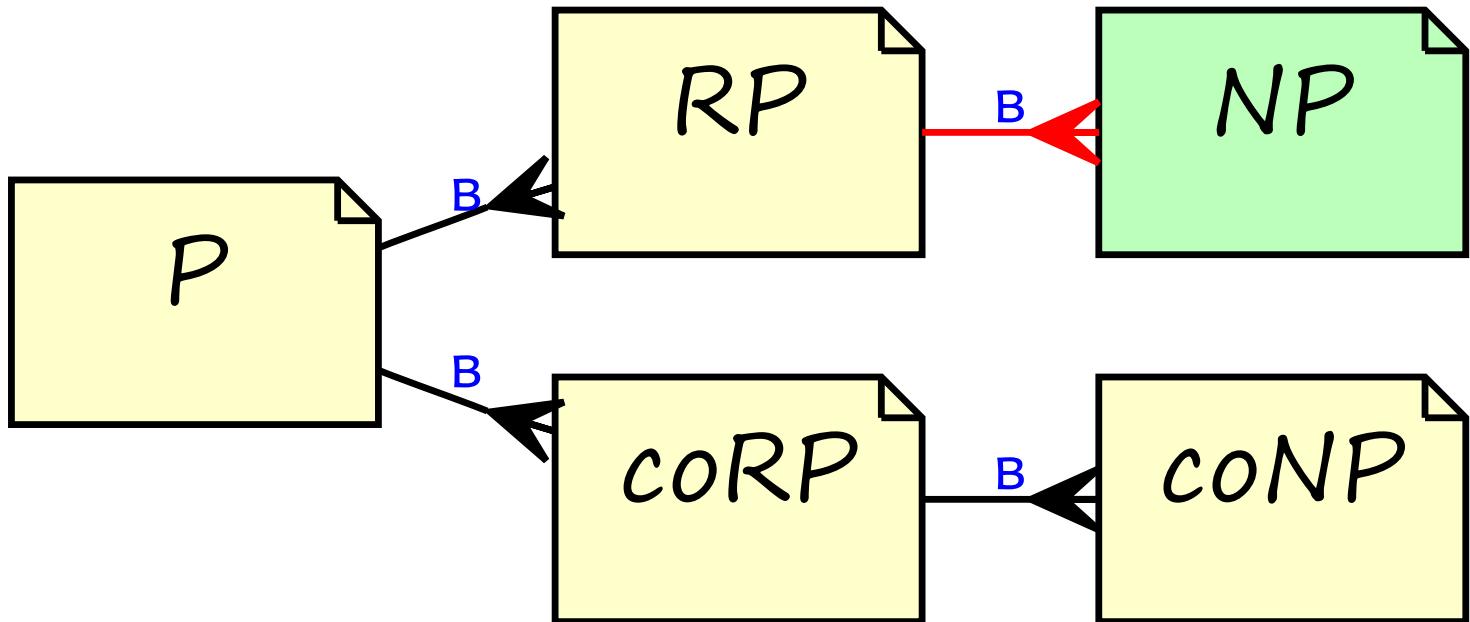


Рис. 6.11: Классы сложности \mathcal{RP} и $\text{co}\mathcal{RP}$

Лемма 35. $\mathcal{RP} \subseteq \tilde{C}$, и, следовательно, $\tilde{C} \equiv \mathcal{RP}$.

Доказательство. Допустим $L \in \mathcal{RP}$. Тогда существует ВМТ $M(x)$ со свойствами, описанными в определении 6.3.3 « \mathcal{RP} ». Для заданного x будем запускать машину M два раза, обозначая результаты запусков через $M_1(x)$ и $M_2(x)$. Построим машину \tilde{M} , дважды вызывающую машину M и выдающую в качестве результата $M_1(x) \vee M_2(x)$. Тогда вероятность ошибки первого рода (других ошибок машина M не допускает), случающейся, когда $x \in L$, а $M(x) = 0$ (обозначим эту вероятность p_M), будет меньше $\frac{1}{2}$. Машина \tilde{M} , по построению, тоже будет допускать только ошибки первого рода, но они будут происходить только в случае ошибки машины M на обоих запусках $M_1(x)$ и $M_2(x)$, и вероятность такой ошибки будет

$$p_{\tilde{M}} = \frac{1}{p_M} \cdot \frac{1}{p_M} < \frac{1}{4}.$$

Соответственно, для заданного языка L построенная нами машина \tilde{M} удовлетворяет описанию языка \tilde{C} . □

Примененный нами метод называется «вероятностным усилением» (или «вероятностной амплификацией» от *amplification*), и, применив его полиномиальное число раз, мы добиваемся экспоненциального уменьшения вероятности ошибки. Таким образом, уже очевидно, что в определениях \mathcal{RP} и $\text{co}\mathcal{RP}$ вместо «одной второй» можно использовать любые константы $\geq \frac{1}{2}$ — с помощью амплификации мы легко можем достичь заданного уровня «здравости».

Можно вполне использовать упоминающуюся шумную сигнализацию, срабатывающую ложно в половине случаев, если заставить ВМТ сигнализации выполнить проверку раз двадцать — тогда вероятность ложной тревоги была бы меньше $\frac{1}{10000}$. Это, к сожалению, пока является недостижимым уровнем для существующих автосигнализаций.

Более того, «порог», после которой можно «усиливать» вероятность, не обязательно должен быть больше одной второй, он может быть даже очень малым, главное, не быть пренебрежимо малым.

Определение 6.3.8. Класс сложности $\mathcal{RP}_{\text{weak}}$ состоит из всех языков L , для которых существуют полиномиальная ВМТ M и полином $p(\cdot)$, такие, что:

$$\begin{aligned} x \in L &\Rightarrow P[M(x) = 1] \geq \frac{1}{p(|x|)}, \\ x \notin L &\Rightarrow P[M(x) = 1] = 0. \end{aligned}$$

Определение 6.3.9. Класс сложности $\mathcal{RP}_{\text{strong}}$ состоит из всех языков L , для которых существуют полиномиальная ВМТ M и полином $p(\cdot)$, такие, что:

$$\begin{aligned} x \in L &\Rightarrow P[M(x) = 1] \geq 1 - 2^{-p(|x|)}, \\ x \notin L &\Rightarrow P[M(x) = 1] = 0. \end{aligned}$$

Поначалу кажется, что $\mathcal{RP}_{\text{weak}}$ — это некоторая релаксация, т. е. определение более широкого класса из-за ослабления ограничений, класса \mathcal{RP} , а $\mathcal{RP}_{\text{strong}}$, наоборот, усиление ограничений, и, по крайней мере:

$$\mathcal{RP}_{\text{strong}} \subseteq \mathcal{RP} \subseteq \mathcal{RP}_{\text{weak}}.$$

Однако оказывается, все это определения одного и того же класса.

Лемма 36. $\mathcal{RP}_{\text{weak}} = \mathcal{RP}_{\text{strong}} = \mathcal{RP}$.

Доказательство. Достаточно доказать $\mathcal{RP}_{\text{weak}} \subseteq \mathcal{RP}_{\text{strong}}$, для чего мы покажем, как для любого языка $L \in \mathcal{RP}_{\text{weak}}$ из машины M_{weak} (машина M из определения 6.3.8) сделать машину M_{strong} , соответствующую определению 6.3.9. Действуем, как и в более простом случае леммы 35 — для данного x машина

M_{strong} запускает t раз машину M_{weak} и возвращает *Логическое ИЛИ* от всех результатов запуска. Так же, как и в случае леммы 35, возможна только ошибка первого рода, причем ее вероятность (если $x \in L$):

$$P(M_{\text{strong}}(x) = 0) = (P(M_{\text{weak}}(x) = 0))^{t(|x|)} = \left(1 - \frac{1}{p(|x|)}\right)^{t(|x|)}.$$

Осталось найти необходимое «для разгона» количество запусков:

$$\left(1 - \frac{1}{p(|x|)}\right)^{t(|x|)} \leq 2^{-p(|x|)},$$

откуда получаем

$$t(|x|) \geq -\frac{\log 2 p(|x|)}{\log \left(1 - \frac{1}{p(|x|)}\right)} \geq \log 2 \cdot p^2(|x|).$$

т. е. за полиномиальное время мы усиливаем вероятность от «полиномиально малой» до «полиномиально близкой к единице». \square

6.3.2 Класс \mathcal{BPP} . «Двусторонние ошибки»

Теперь рассмотрим, что происходит, если допустить возможность двусторонних ошибок, т. е. ошибок первого и второго рода. Определим класс языков, эффективно распознаваемых на ВМТ с двусторонней ошибкой.

Определение 6.3.10. Класс сложности \mathcal{BPP} (*Bounded-Probability Polynomial-time*) состоит из всех языков L , для которых существует полиномиальная ВМТ M , такая, что:

$$\begin{aligned} x \in L &\Rightarrow P[M(x) = 1] \geq \frac{2}{3}, \\ x \notin L &\Rightarrow P[M(x) = 0] \geq \frac{2}{3}. \end{aligned}$$

Из определения видно, что класс \mathcal{BPP} замкнут относительно дополнения.

По опыту раздела 6.3.1 читатель ожидает обобщений определения класса \mathcal{BPP} без магической константы « $\frac{2}{3}$ », и мы его не разочаруем. Итак, предоставим «свободное» и «жесткое» определения класса \mathcal{BPP} .

Определение 6.3.11. Класс сложности $\mathcal{BPP}_{\text{weak}}$ состоит из всех языков L , для которых существуют:

$c, 0 < c < 1$ — константа;

$p(\cdot)$ — положительный полином;

M — полиномиальная ВМТ;

такие, что:

$$\begin{aligned} x \in L &\Rightarrow P[M(x) = 1] \geq c + \frac{1}{p(|x|)}, \\ x \notin L &\Rightarrow P[M(x) = 1] < c - \frac{1}{p(|x|)}. \end{aligned}$$

Определение 6.3.12. Класс сложности $\mathcal{BPP}_{\text{strong}}$ состоит из всех языков L , для которых существуют полиномиальная ВМТ M , и полином $p(\cdot)$, такие, что:

$$\begin{aligned} x \in L &\Rightarrow P[M(x) = 1] \geq 1 - 2^{-p(|x|)}, \\ x \notin L &\Rightarrow P[M(x) = 0] \geq 1 - 2^{-p(|x|)}. \end{aligned}$$

В следующей лемме мы используем следующие результаты из теории вероятностей.

Теорема 32. «Закон Больших Чисел для схемы Бернулли»

Пусть событие A может произойти в любом из t независимых испытаний с одной и той же вероятностью p , и пусть M_t — число осуществлений события A в t испытаниях. Тогда $\frac{M_t}{t} \rightarrow p$, причем для любого $\varepsilon > 0$:

$$P\left(\left|\frac{M_t}{t} - p\right| \geq \varepsilon\right) \leq \frac{p(1-p)}{t\varepsilon^2}.$$

Теорема 33. «Оценка Чернова» (Chernoff bounds).

Пусть M_1, \dots, M_t — независимые события, каждое из которых имеет вероятность $p \geq \frac{1}{2}$. Тогда вероятность одновременного выполнения более половины событий будет больше, чем $1 - \exp(-2(p - \frac{1}{2})^2 t)$.

Лемма 37. $\mathcal{BPP}_{\text{weak}} = \mathcal{BPP}$.

Доказательство. Очевидно, что $\mathcal{BPP} \subseteq \mathcal{BPP}_{\text{weak}}$, для этого достаточно положить $c \equiv \frac{1}{2}$ и $p(|x|) \equiv 6$.

Теперь покажем $\mathcal{BPP}_{\text{weak}} \subseteq \mathcal{BPP}$. Пусть $L \in \mathcal{BPP}_{\text{weak}}$, обозначим через M_{weak} ВМТ из определения 6.3.11, построим M из определения 6.3.10 « \mathcal{BPP} ».

На входе x машина M вычислит $p(|x|)$, затем будет запускать $t = 6p^2(|x|)$ раз машину M_{weak} (обозначим результат i -го запуска через M_{weak}^i) и возвращать «1», если $M_t = \sum_{i=1}^t M_{\text{weak}}^i > t \cdot c$, или «0» в противном случае.

Рассмотрим $P_{err} = P(M(x) = 0 | x \in L)$ — вероятность ошибки первого рода:

$$P_{err} = P[M_t < t \cdot c].$$

Заметим, что при $x \in L$ мат. ожидание «суммарного голосования» машин M_{weak}^i будет

$$\mathbf{E} M_t(x) = t \cdot \mathbf{E} [M_{weak}^i(x)] \geq t \cdot \left(c + \frac{1}{p(|x|)} \right),$$

откуда, применяя теорему 32 и учитывая, что $0 \leq \frac{\mathbf{E}(M_t)}{t} \leq 1$, получаем

$$P_{err} \leq P \left(\left| \frac{M_t}{t} - \frac{\mathbf{E}(M_t)}{t} \right| \geq \frac{1}{p(|x|)} \right) \leq \frac{p^2(|x|)}{t} \leq \frac{1}{3}.$$

Вероятность ошибки второго рода $P_{err} = P(M(x) = 1 | x \notin L)$ оценивается аналогично. \square

Лемма 38. $\mathcal{BPP}_{strong} = \mathcal{BPP}$.

Доказательство. Очевидно, что $\mathcal{BPP}_{strong} \subseteq \mathcal{BPP}$, осталось показать обратное вложение. Действуем аналогично лемме 37, строим машину M_{strong} , запуская $t = 2p(|x|) + 1$ раз (пусть будет нечетное число) обычную \mathcal{BPP} -машину M и принимая решения на основе «большинства» ее результатов. Если $x \in L$, то,

согласно теореме 33, вероятность правильного ответа:

$$\begin{aligned}
 \mathbb{P}(M_{strong}(x) = 1) &\geq 1 - \exp\left(-2\left(p - \frac{1}{2}\right)^2 \cdot t\right) = \\
 &= 1 - \exp\left(-2\left(\frac{2}{3} - \frac{1}{2}\right)^2 \cdot (2p(|x|) + 1)\right) = \\
 &= 1 - \exp\left(-\frac{2p(|x|) + 1}{18}\right) > 1 - e^{-\frac{p(|x|)}{9}} > 1 - 2^{-p(|x|)}.
 \end{aligned}$$

□

Упражнение 6.3.1. Вы разработчик военного программного комплекса, использующего сложновычислимую и секретную функцию $F : [0, \dots, N-1] \rightarrow [0, \dots, m-1]$, которую подключают к вашему алгоритму в виде отдельного массива длины N , т. е. функция задана таблично на внешней флеш-памяти огромного объема(только вес этой флэшки — 20 кг, которую носят и охраняют два майора-особиста). Функция гомоморфна, т. е.

$$F((x + y) \bmod N) = (F(x) + F(y)) \bmod m$$

Однако, утром перед приемными испытаниями, выяснилось, что «флешка побилась», т. е. некоторые значения этой функции стали неверными. Виновные майоры уже расстреляны, а вся команда разработчиков пытается решить проблему.

Инженеры исследовали сбой — и утверждают, что «побилось» не более $\frac{1}{6}$ ячеек, к сожалению, неизвестно каких.

С учетом этого факта вам поставлена задача реализовать простой и быстрый алгоритм, который правильно вычисляет $F(x)$ с вероятностью не меньше $\frac{2}{3}$.

Стрелки показывают вложенность классов сложности

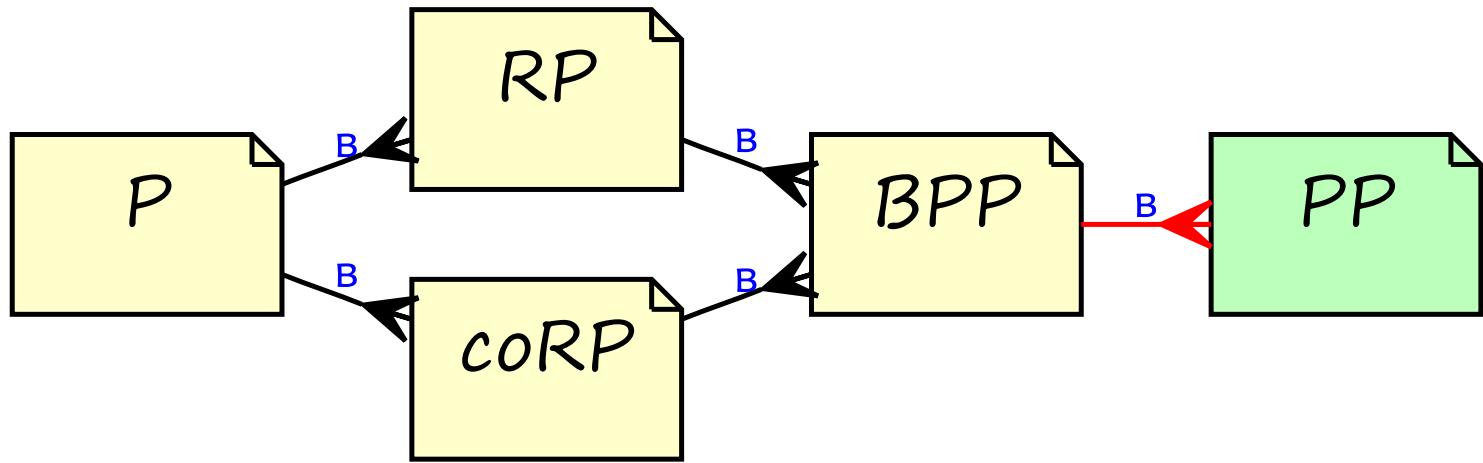


Рис. 6.12: Классы сложности: \mathcal{BPP} и его «соседи»

Упражнение 6.3.2. Все то же, что и в упражнении 6.3.1, но чтобы вероятность ошибки P_{err} можно было сделать произвольно малой, т. е. $2^{-p(|x|)}$, и при этом, чтобы выполнение алгоритма было замедлено не больше, чем в $O(p(|x|))$ раз.

Упражнение 6.3.3. Вводная, что и в упражнении 6.3.1, только теперь достаточно времени (испытания прошли, есть пара недель перед вводом в опытную эксплуатацию), и нужно восстановить значение этой функции за время $O(N^2)$.

6.3.3 Класс \mathcal{PP}

Теперь познакомимся с максимально широким классом языков, распознаваемых полиномиальной ВМТ.

Определение 6.3.13. Класс сложности \mathcal{PP} (*Probability Polynomial-time*) состоит из всех языков L , для которых существует полиномиальная ВМТ M , такая, что:

$$\begin{aligned} x \in L &\Rightarrow P[M(x) = 1] > \frac{1}{2}, \\ x \notin L &\Rightarrow P[M(x) = 0] > \frac{1}{2}. \end{aligned}$$

По опыту разделов 6.3.1 и 6.3.2 читатель может предположить, что константа « $\frac{1}{2}$ » также выбрана произвольно, «для красоты», но здесь это не так. В отличие от определений 6.3.3 « \mathcal{RP} », 6.3.10 « \mathcal{BPP} », в определении 6.3.13 « \mathcal{PP} » мы никак не можем заменить константу « $\frac{1}{2}$ » на любую большую константу, т.к. в этом случае нет гарантированной возможности амплификации вероятности за полиномиальное время.

Обратите внимание, что в определении 6.3.13 « \mathcal{PP} » важна даже строгость неравенства (что необязательно для определений 6.3.3 « \mathcal{RP} », 6.3.10 « \mathcal{BPP} »), т. е. нельзя заменить символ « $>$ » на « \geq », т.к. тогда определение полностью «выродится», ведь проверяющую машину можно будет заменить подбрасыванием одной монетки. Единственное ослабление, на которое мы можем пойти, — это пожертвовать строгостью одного из неравенств в определении.

Определение 6.3.14. Класс сложности $\mathcal{PP}_{\text{weak}}$ состоит из всех языков L , для которых существует полиномиальная ВМТ M , такая, что:

$$\begin{aligned} x \in L &\Rightarrow P[\mathbf{M}(x) = 1] > \frac{1}{2}, \\ x \notin L &\Rightarrow P[\mathbf{M}(x) = 0] \geq \frac{1}{2}. \end{aligned}$$

Упражнение 6.3.4. Что будет, если в определении 6.3.14 в обоих неравенствах поставить « \geq »? Какой класс языков будет определен?

Лемма 39. $\mathcal{PP}_{weak} = \mathcal{PP}$.

Доказательство. Очевидно, доказывать нужно только вложение $\mathcal{PP}_{weak} \subseteq \mathcal{PP}$, для чего мы покажем, как для любого языка $L \in \mathcal{PP}_{weak}$ из машины M_{weak} (машина M из определения 6.3.14) сделать машину M из определения 6.3.13 « \mathcal{PP} ». Пусть машина M_{weak} использует не больше $w(|x|)$ случайных бит (см. определение 6.3.2 «offline BMT»), машина M будет использовать $p(|x|) = 2 \cdot w(|x|) + 1$ случайных бит следующим образом:

$$M(x, \langle r_1, \dots, r_{p(|x|)} \rangle) \equiv (r_{w(|x|)+1} \vee \dots \vee r_{p(|x|)}) \wedge M_{weak}(x, \langle r_1, \dots, r_{w(|x|)} \rangle),$$

причем, «оптимизируя» вычисление этого выражения, мы даже не будем запускать M_{weak} , если $r_{w(|x|)+1} \vee \dots \vee r_{p(|x|)}$, что, очевидно, может произойти с вероятностью $2^{-(w(|x|)+1)}$.

Итак, рассмотрим случай $x \in L$. Тогда $P(M_{weak}(x) = 1) > \frac{1}{2}$, причем $P(M_{weak}(x) = 1) \geq \frac{1}{2} + 2^{-w(|x|)}$, т.к. всего вероятностных строк не больше $2^{w(|x|)}$, и минимальный «квант» вероятности, соответствующий одной вероятностной строке, будет не меньше $2^{-w(|x|)}$.

$$\begin{aligned}
 P(M(x) = 1) &= (1 - 2^{-(w(|x|)+1)}) \cdot P(M_{\text{weak}}(x) = 1) \geq \\
 &\geq (1 - 2^{-(w(|x|)+1)}) \cdot \left(\frac{1}{2} + 2^{-w(|x|)} \right) = \\
 &= \frac{1}{2} + 2^{-(w(|x|)+1)} \left(\frac{3}{2} - 2^{-w(|x|)} \right) > \frac{1}{2}.
 \end{aligned}$$

Если $x \notin L$, то $P(M_{\text{weak}}(x) = 0) \geq \frac{1}{2}$, а

$$\begin{aligned}
 P(M(x) = 0) &= (1 - 2^{-(w(|x|)+1)}) \cdot P(M_{\text{weak}}(x) = 0) + \\
 &\quad + 2^{-(w(|x|)+1)} \geq (1 - 2^{-(w(|x|)+1)}) \cdot \frac{1}{2} + 2^{-(w(|x|)+1)} > \frac{1}{2}.
 \end{aligned}$$

□

Исследуем отношение класса \mathcal{PP} к известным нам классам сложности.

Теорема 34. $\mathcal{PP} \subseteq \mathcal{PSPACE}$.

Доказательство. Для любого языка $L \in \mathcal{PP}$ из машины M_{PP} (машина M из определения 6.3.13 « \mathcal{PP} ») можно сделать машину M , которая последовательно запускает M_{PP} на x и всех $2^{p(|x|)}$ возможных вероятностных строках, и результат определяется по большинству результатов запусков. Машина M будет разрешать язык L и использовать не более полинома ячеек на ленте, т.к. каждый запуск M_{PP} может использовать один и тот же полиномиальный отрезок ленты. □

Теорема 35. $\mathcal{NP} \subseteq \mathcal{PP}$.

Доказательство. Покажем, как для любого языка $L \in \mathcal{NP}$ из машины M_{NP} (машина M из определения 6.2.4 « $\mathcal{NP}/\text{ДМТ}$ ») сделать машину M из определения 6.3.14. Пусть размер подсказки y для машины M_{NP} ограничен полиномом $p(|x|)$. Тогда машина M будет использовать $p(|x|) + 1$ случайных бит следующим образом:

$$M(x, \langle r_1, \dots, r_{p(|x|)+1} \rangle) \equiv r_{p(|x|)+1} \vee M_{NP}(x, \langle r_1, \dots, r_{p(|x|)} \rangle).$$

При $x \in L$, $P(M(x) = 1) = \frac{1}{2} + 2^{-p(|x|)} > \frac{1}{2}$. При $x \notin L$, $P(M(x) = 0) = \frac{1}{2}$. Таким образом, учитывая лемму 39, получаем $L \in \mathcal{PP}_{weak} = \mathcal{PP}$. \square

Аналогично получаем следующую лемму.

Лемма 40. $\text{co}\mathcal{NP} \subseteq \mathcal{PP}$.

6.3.4 Класс \mathcal{ZPP} . «Алгоритмы без ошибок»

Пока в разделах 6.3.1, 6.3.2 и 6.3.3 мы рассматривали «ошибающиеся» вероятностные алгоритмы распознавания. Еще один интересный класс вероятностных алгоритмов распознавания — алгоритмы, которым в дополнение к стандартным ответам «0» и «1» разрешено выдавать неопределенный ответ «?». Ответ «?» означает «не знаю» и не считается ошибочным в любом случае. Используя эти алгоритмы, мы можем определить еще один класс языков.

Определение 6.3.15. Класс сложности \mathcal{ZPP} состоит из всех языков L , для которых существует полиномиальная ВМТ M , возвращающая только ответы «0», «1», «?» («не знаю»), причем:

$$\begin{aligned} x \in L \Rightarrow P[M(x) = 1] &> \frac{1}{2} \quad \wedge \quad P[M(x) = 1] + P[M(x) = «?»] = 1, \\ x \notin L \Rightarrow P[M(x) = 0] &> \frac{1}{2} \quad \wedge \quad P[M(x) = 0] + P[M(x) = «?»] = 1. \end{aligned}$$

Оказывается, у этого класса есть и альтернативное определение:

Теорема 36. $\mathcal{ZPP} = \mathcal{RP} \cap \text{co}\mathcal{RP}$.

Доказательство. $\mathcal{ZPP} \subseteq \mathcal{RP}$, т.к. для любого языка $L \in \mathcal{ZPP}$ из машины M_{ZPP} (машина M из определения 6.3.15 « \mathcal{ZPP} ») можно сделать машину M из определения 6.3.3 « \mathcal{RP} »:

```
Answer = MZPP(x)
if Answer = «?» then
    Answer = 0
end if
RETURN Answer
```

Действительно, если $x \in L$, то $P[M_{ZPP}(x) = 1] > \frac{1}{2}$, и, следовательно, $P[M(x) = 1] \geq \frac{1}{2}$. Если $x \notin L$, то

$$P[M(x) = 0] = P[M_{ZPP}(x) = 0] + P[M_{ZPP}(x) = «?»] = 1.$$

Аналогично доказывается, что $\mathcal{ZPP} \subseteq \text{co}\mathcal{RP}$.

Теперь покажем, что $\mathcal{RP} \cap \text{co}\mathcal{RP} \subseteq \mathcal{ZPP}$. Изготовим машину M для распознавания \mathcal{ZPP} из машин M_{RP} (M из определения 6.3.3 « \mathcal{RP} ») и M_{coRP} (M из определения 6.3.4 « $\text{co}\mathcal{RP}$ »), используя «безошибочные» возможности обеих машин:

```
if MRP(x) = 1 then
    RETURN «1»
end if
if McoRP(x) = 0 then
    RETURN «0»
end if
```

RETURN «?»

□

Итак, кратко напомним отношения между уже знакомыми и новыми вероятностными классами сложности, определенными в этом разделе, на рис. 6.13, где стрелками отображаются отношения вложенности.

Упражнение 6.3.5. Класс сложности $\mathcal{ZPP}_{\text{NotNull}}$ состоит из всех языков L , для которых существует ВМТ M , всегда возвращающая правильные ответы (т. е. никаких «не знаю»), причем \exists полином $p(n)$, что для времени работы $T_M(x)$ машины M выполняется:

$$\mathbf{E} T_M(x) \leq p(|x|).$$

Докажите, что $\mathcal{ZPP}_{\text{NotNull}} = \mathcal{ZPP}$.

6.3.5 Вероятностно проверяемые доказательства

К середине 1990-х годов было доказано, что для многих задач полиномиальных алгоритмов с лучшими, чем известные, оценками точности не существует при стандартной гипотезе $\mathcal{P} \neq \mathcal{NP}$ (или сходных с ней). Результат, который позволил доказывать подобные теоремы, — одно из самых ярких достижений теории сложности. Он получил название \mathcal{PCP} -теоремы (Probabilistically Checkable Proofs).

Эта теорема устанавливает неожиданную связь между интерактивными доказательствами и приближенными алгоритмами. Как мы уже говорили, один из путей понимания класса \mathcal{NP} состоит в том, чтобы рассматривать его как класс языков, доказательства принадлежности к которым найти может и трудно, но зато эти доказательства легко проверить.

Стрелки показывают вложенность классов сложности

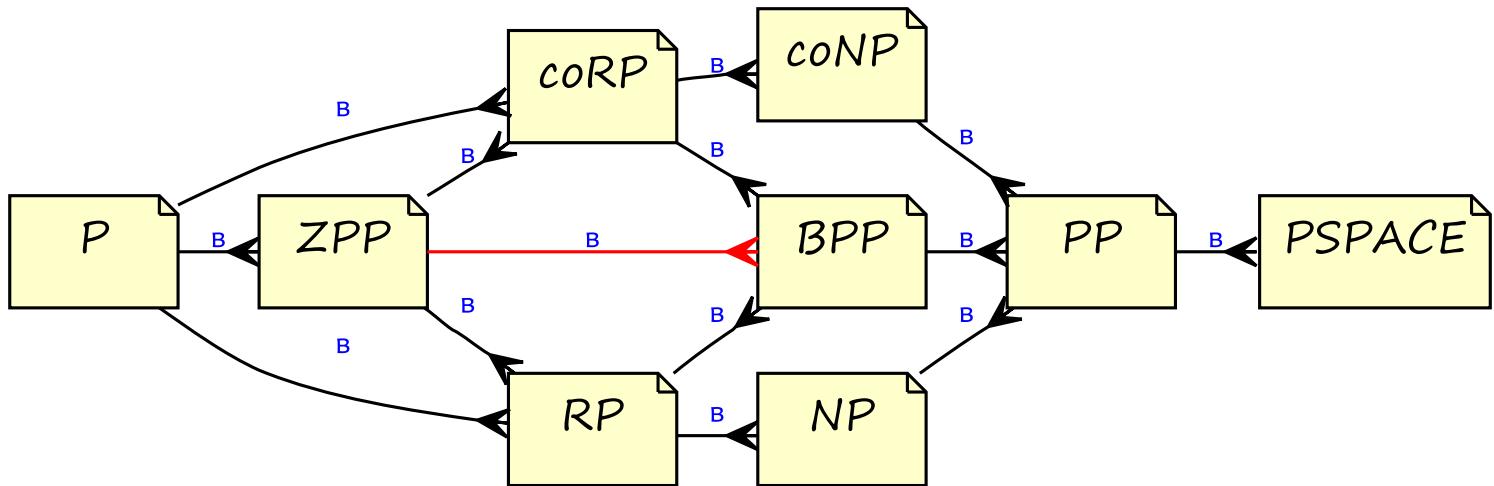


Рис. 6.13: Вероятностные классы сложности: ZPP, RP, BPP, NP

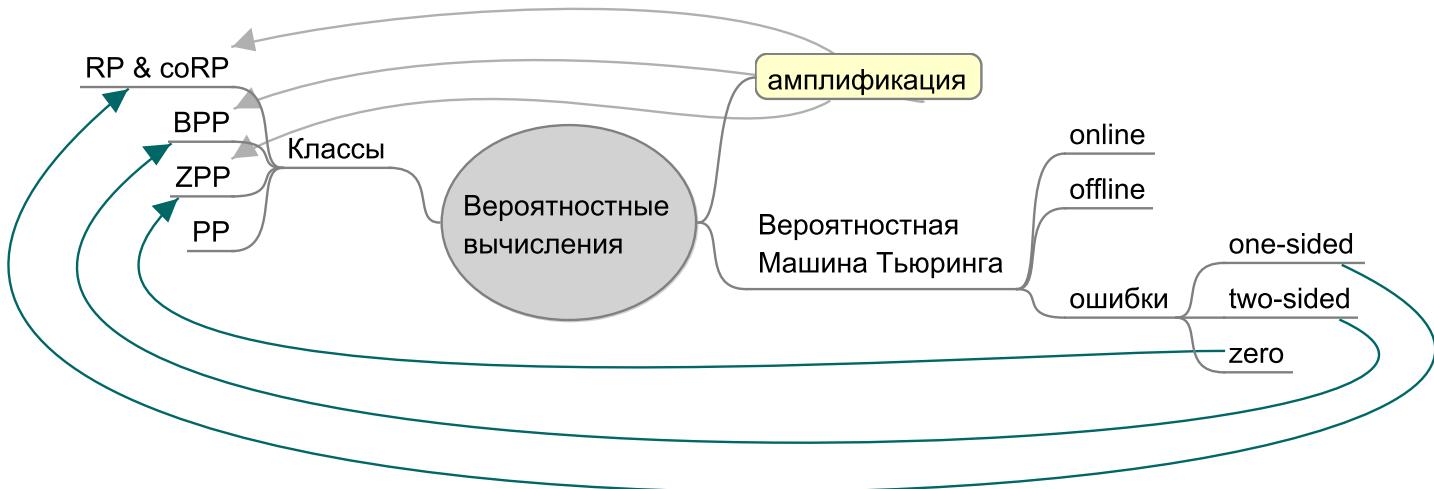


Рис. 6.14: Карта-памятка раздела 6.3

Предположим, однако, что при выяснении принадлежности $x \in L$ для некоторого языка $L \in \mathcal{NP}$ мы не хотим смотреть все доказательство.

Вместо этого мы хотим осуществить несколько случайных проверок и затем с достаточной уверенностью констатировать, что доказательство корректно.

Говоря неформально, система вероятностной проверки доказательств (*Probabilistically Checkable Proof System*, \mathcal{PCP} , мы будем называть ее \mathcal{PCP} -системой) для некоторого языка состоит из полиномиальной проверяющей вероятностной машины Тьюринга (ВМТ), имеющей специальный доступ к отдельным битам бинарной строки, представляющей доказательство. Предоставлением этой строки занимается специ-

альный оракул, понятие, часто используемое в теории сложности и обозначающее устройство, способное находить ответы на поставленные ему вопросы. Машины Тьюринга (детерминированные, недетерминированные, вероятностные) сопрягаются с этим устройством путем установки в каждую машину отдельной ленты. На этой ленте они пишут вопросы к оракулу и после перехода в специальное состояние «обращение к оракулу» за один такт работы получают на этой ленте ответ. Обычно ограничиваются оракулами, возвращающими один бит, например, оракул может заниматься распознаванием некоторого языка, возвращая «0/1» результаты проверки принадлежности слова на оракульной ленте.

В данном случае оракул является хранителем некоторой строки-доказательства π , состоящей из конкатенации индивидуальных строк-доказательств π_x , специфичных для каждого входного слова x , а проверяющая ВМТ, проверяя слово x , запрашивает у оракула отдельные биты π_x , посылая запросы типа «позиция в строке π_x ». Соответственно, в конце вычисления проверяющая ВМТ выносит вердикт о принадлежности слова языку, причем она должна «одобрить» все $x \in L$ и с вероятностью не меньшей $\frac{1}{2}$ «забраковать» $x \notin L$.

Можно представить судебный/следственный процесс над группой подозреваемых, где некоему суперкомпьютеру (Большой Брат, Матрица) известно абсолютно все, для каждого подозреваемого « x » в нем хранится полнейшее досье $\langle\pi_x\rangle$. Но следователь, допрашивая подозреваемого « x », не имеет сил и времени изучить абсолютно все досье $\langle\pi_x\rangle$ и задает суперкомпьютеру запросы по его содержимому, например, «где был x в такое-то время», «знаком ли x с y » и т. п.

Давайте сравним определение \mathcal{PCP} -системы с определением класса \mathcal{NP} через понятия «доказательства» и «верификации» (определение 6.2.4 « \mathcal{NP} /ДМТ»). Перечислим различия.

1. Верификатором для класса \mathcal{NP} была ДМТ, а у \mathcal{PCP} -системы — ВМТ.
2. Для каждого x строка доказательства у \mathcal{NP} была полиномиального размера, а у \mathcal{PCP} -системы каж-

дая строка π_x может быть экспоненциального размера¹³.

3. В случае \mathcal{NP} верификатор сразу же получает доступ ко всему доказательству, а \mathcal{PCP} -система при любой длине доказательства успеет просмотреть часть не более полиномиальной длины. Впрочем, \mathcal{PCP} -система может вполне «побрезговать» полным доказательством, даже если оно полиномиального размера, ограничившись просмотром константы битов из доказательства, или вовсе не смотреть на него, вынеся результат из исследования входного слова и вероятностного «подбрасывания монеток». Так же \mathcal{PCP} -система может обойтись и без «монеток».

Теперь дадим более формальное определение.

Определение 6.3.16. Системой вероятностной проверки доказательств (верифицирующей \mathcal{PCP} -системой) для языка L называется ВМТ M с оракулом, для которой выполняются следующие условия:

полнота (completeness): $\forall x \in L$ существует оракул π_x :

$$\mathbb{P}[M^{\pi_x}(x) = 1] = 1.$$

корректность (soundness): $\forall x \notin L$ и для любого оракула π :

$$\mathbb{P}[M^\pi(x) = 1] \leq \frac{1}{2}.$$

Заметим, что \mathcal{PCP} -системы подразделяют на *адаптивные* и *неадаптивные*. В адаптивных системах запросы к оракулу зависят от предыдущих его ответов. Неадаптивная система получает на вход слово x ,

¹³Больше, чем экспоненциального размера, она быть не может, т.к. тогда номера позиций в этой строке будут более чем полиномиальны, и полиномиальная ВМТ не успеет их даже запросить, т. е. написать эти номера на оракульной ленте.

загружает строку случайных битов. На основе этой строки и входного слова формулирует все свои запросы к оракулу и, получив на них ответы, больше к нему не обращается.

Понятно, что неадаптивные системы являются частным, и, следовательно, более слабым случаем адаптивных систем, и далее мы по умолчанию будем считать \mathcal{PCP} -системы адаптивными.

Теперь определим, какие ресурсы потребляет \mathcal{PCP} -система и как вводить специфические меры и классы сложности. Таких ресурсов будет два (мы исключаем время и память — объявив процесс верификации полиномиальным, мы более не интересуемся ни тем, ни другим): «число случайных бит» и «число запросов к оракулу».

Определение 6.3.17. Пусть $r, q : N \Rightarrow N$ — неотрицательные целочисленные функции.

Класс сложности $\mathcal{PCP}(r(\cdot), q(\cdot))$ состоит из языков, имеющих верифицирующую \mathcal{PCP} -систему, которая на входе x :

1. потребляет не более $r(|x|)$ случайных бит;
2. делает не более $q(|x|)$ запросов к оракулу.

Для множеств цепочисленных функций R, Q определим

$$\mathcal{PCP}(R, Q) \equiv \bigcup_{r \in R, q \in Q} \mathcal{PCP}(r(\cdot), q(\cdot)).$$

т. е. одним определением 6.3.17 « \mathcal{PCP} », в зависимости от этих двух параметров, определяется множество различных классов сложности. Например, класс $\mathcal{PCP}(\text{poly}, \text{poly})$ очень мощный — доказано, что он совпадает с очень широким классом $\mathcal{NEXP} \equiv \mathcal{NTIME}(2^{\text{poly}})$.

Если рассмотреть вырожденные случаи (когда один из параметров равен нулю), то определение 6.3.17 « \mathcal{PCP} » переходит в определение классов \mathcal{NP} (определение 6.2.4 « \mathcal{NP} /ДМТ»):

$$\mathcal{PCP}(0, \text{poly}) = \mathcal{NP}$$

и coRP (см. определение 6.3.6 « coRP /ДМТ»):

$$\mathcal{PCP}(\text{poly}, 0) = \text{coRP}.$$

Но ценность \mathcal{PCP} заключается не только в «сведении к единому знаменателю» определений классов coRP и \mathcal{NP} , а в демонстрации связи и взаимозаменяемости «вероятностного» и «информационного» ресурсов. Оказалось, что один и тот же класс языков может быть определен как $\mathcal{PCP}(P, Q)$ с разными парами (P, Q) . Проще говоря, можно обменивать «случайные биты» на «запросы к оракулу» и наоборот.

Попробуем увидеть, как это может происходить.

Лемма 41. $\mathcal{PCP}(\log, \text{poly}) \subseteq \mathcal{NP}$.

Доказательство. Пусть язык L лежит в $\mathcal{PCP}(\log, \text{poly})$. Покажем, как построить верификатор класса \mathcal{NP} , который сможет разрешить L . Рассмотрим M' — оракульную ВМТ из \mathcal{PCP} -системы, распознающей L (можно считать ее аддитивной). Введем обозначения:

1. $\forall i \in 1, \dots, m \quad r_i$ — i -я вероятностная строка, из m возможных случайных строк длины не больше $\log n$, потребляемых ВМТ (см. определение 6.3.2 «offline ВМТ»);
2. для каждой вероятностной строки i ($1 \leq i \leq m$) пусть

$\langle q_1^i, \dots, q_{n_i}^i \rangle$ — последовательность вопросов к оракулу, основанных только на входном слове x и r_i ;

$\langle \pi_{q_1^i}, \dots, \pi_{q_{n_i}^i} \rangle$ — ответы оракула на эти вопросы.

Сначала обеспечим « \mathcal{NP} -одобрение», т. е. проверим, что для любого $x \in L$ существует полиномиальное « \mathcal{NP} -доказательство» y , такое, что их вместе $(x\#y)$ распознает полиномиальная ДМТ. Мы не можем использовать как « \mathcal{NP} -доказательство» не только полное доказательство оракула π , но даже оракульное доказательство π_x для входного слова x , т.к. любое из этих доказательств может быть экспоненциального размера, но если мы рассмотрим массив $\langle \pi_{q_1^i}, \dots, \pi_{q_{n_i}^i} \rangle$, для $i \in (1, \dots, m)$, то мы видим, что его можно закодировать строкой y полиномиальной длины, т.к. число различных вероятностных строк $m \leq 2^{\log(|x|)} = \text{poly}(|x|)$, число ответов от оракула также не больше $\text{poly}(|x|)$, таким образом, y — полиномиальное « \mathcal{NP} -доказательство» для входного слова x .

Верификатор-ДМТ M на входе $x\#y$ просимулирует ВМТ M' на всевозможных значениях строк r , и в процессе симуляции M' вместо оракула предоставит M' ответы из « \mathcal{NP} -доказательства» y (в зависимости от «выпавшей» «случайной» строки r). Если для каждой строки r симулируемая M' примет x , то же сделает и M . Так как мы уже видели, что количество строк r есть полином от $|x|$, y — полиномиальная строка, симуляция полиномиальной M' — также полином, то и описанная ДМТ M также будет полиномиальной.

Теперь рассмотрим случай, когда $x \notin L$, и убедимся, что $\forall y : M(x, y) = 0$. Допустим, что $\exists y : M(x, y) = 1$ для $x \notin L$. Но тогда (см. описание работы машины M) мы можем «превратить» это « \mathcal{NP} -доказательство» y обратно в некоторого оракула π^y , для которого $P(M'^{\pi^y}(x) = 1) = 1$, что противоречит условию корректности («soundness») из определения 6.3.16 « \mathcal{PCP} -система».

□

Эта лемма показала, как «упаковывать» вероятностные доказательства \mathcal{PCP} -систем, преобразуя их в полиномиальные « \mathcal{NP} -доказательства». Теперь поинтересуемся обратным вложением — какие существуют субполиномиальные функции $q(\cdot)$, такие, чтобы выполнялось $\mathcal{NP} \subseteq \mathcal{PCP}(\log, q(\cdot))$? Говоря нефор-

мально — можем ли мы сэкономить на полной честной проверке полиномиальных доказательств, заменив ее вероятностной, но более быстрой и «дешевой» проверкой?

Оказалось — да. Соответствующий результат, являющийся, вероятно, величайшим достижением теории сложности за последние два десятилетия, формулируется следующим образом.

Теорема 37. $\mathcal{NP} \subseteq \mathcal{PCP}(\log, O(1))$.

Доказательство этой теоремы чрезвычайно сложно, первоначальный вариант доказательства занимает порядка сотни страниц, поэтому мы приводим ее без доказательства. Зато, комбинируя результат этой теоремы 37 «*PCP-theorem*» с ранее доказанной нами леммой 41, получаем новое, нетривиальное определение знакомого нам класса NP :

$$\mathcal{NP} = \mathcal{PCP}(\log, O(1)). \quad (6.1)$$

Интересно, что доказаны даже более конкретные определения класса \mathcal{NP} через \mathcal{PCP} с указанием не класса функций, описывающих число запросов к оракулу, а непосредственно минимального числа битов доказательства, которые необходимо запросить у оракула ([Gur+98]):

$$\mathcal{NP} = \mathcal{PCP}(\log, q = 5), \quad (6.2)$$

где обозначением $q = 5$ мы отметили, что требуется не просто константа, а конкретное (5) число проверочных битов.

То, что таких битов не может быть меньше трех, при гипотезе, что $\mathcal{P} \neq \mathcal{NP}$, мы докажем в следующей теореме.

Теорема 38. $P = \mathcal{PCP}(\log, q = 2)$.

Доказательство. Естественно, доказать нужно только утверждение $\mathcal{PCP}(\log, q = 2) \subseteq P$. Действуем аналогично лемме 41, воспользуемся ее обозначениями для имеющейся ВМТ, вероятностных строк, запросов к оракулу и ответов оракула. На каждое i -е бросание монет $\langle r_1, \dots, r_m \rangle$ ответов от оракула должно быть всего (не больше чем) два — $\langle \pi_1^i, \pi_2^i \rangle$. По схеме верификации машиной M' (см. теорему 29 « $SAT \in \mathcal{NP}\mathcal{C}$ ») можно построить 2SAT-формулу ϕ — КНФ, где будет m -дизъюнкций, соответствующих вероятностным строкам, где каждая дизъюнкция содержит не больше, чем две переменные, соответствующие ответам оракула $\langle \pi_1^i, \pi_2^i \rangle$.

Тогда вопрос о существовании оракульного доказательства π_x для заданного слова x эквивалентен выполнимости 2SAT-формулы ϕ , а эта задача полиномиально разрешима (см. упражнение 6.2.4). \square

6.3.6 \mathcal{PCP} и неаппроксимируемость

Введение и исследование понятия \mathcal{PCP} -системы помогло в доказательствах неаппроксимируемости некоторых оптимизационных задач. Действительно, многие известные оптимизационные задачи, т. е. задачи, в которых находится оптимальное решение, стоимость которого должна быть максимальна (или минимальна), принадлежат классу $\mathcal{NP}\mathcal{C}$ или \mathcal{NP} -трудны. Хотя это означает, что при гипотезе $P \neq \mathcal{NP}$ точное решение получить эффективно невозможно, эта принадлежность ничего не говорит о возможности аппроксимации задачи, т. е. нахождения приближенного решения, которого, возможно, будет достаточно во многих практических случаях.

Определение 2.1.1 « C -приближенный алгоритм» включает все виды приближенных алгоритмов с гарантированной оценкой точности — как алгоритмы с константной точностью, так и более частный случай алгоритмов, у которых точность C выступает в роли одного из параметров, и они способны добиться любой заданной точности, расплатившись за нее временем. Разумеется, нас будут интересовать в первую

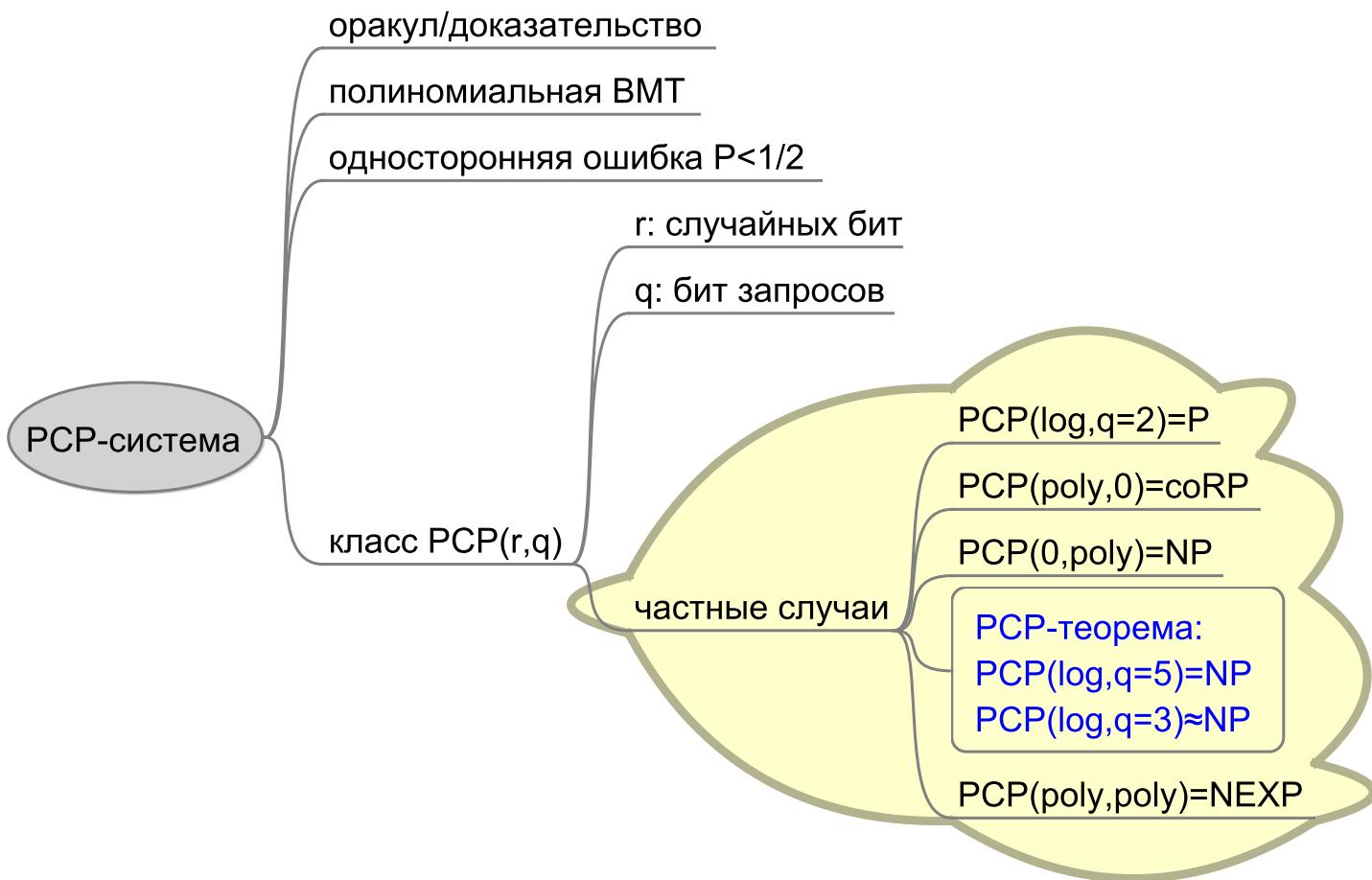


Рис. 6.15: Карта-памятка раздела 6.3.5

очередь эффективные, т. е. полиномиальные, приближенные алгоритмы. Отметим, что для задач максимизации целевой функции часто используют обратную величину и говорят $\frac{1}{C}$ -приближенный алгоритм (например, 0.878-приближенный алгоритм нахождения максимального разреза в графе).

Обратите внимание — если алгоритм «старается» найти хорошее решение, но не гарантирует точности на всех входных данных, то его обычно называют эвристикой (см. определение 1.1.1). В нашем курсе мы не рассматриваем такие алгоритмы, хотя они бывают популярными. Как правило, популярность ряда эвристик означает, что на большой доле входных данных они работают хорошо, а значит, можно доказать их положительные характеристики в среднем.

Далее в этом разделе мы будем рассматривать задачу 28 «3SAT» и ее оптимизационную версию — задачу 32.

Задача 31. «MAX-SAT (ε)».

Задача ε -разрешения для задачи 20 «MAX-SAT». Известно, что для данного $\varepsilon > 0$:

- либо доля невыполненных скобок в КНФ не меньше ε ,
- либо КНФ выполнима.

Определить, какая ситуация имеет место.

Задача 32. «MAX-3SAT(ε)».

Частный случай задачи 31 «MAX-SAT (ε)», в которой в каждой скобке-дизъюнкции не более трех переменных.

Оказалось, тесно взаимосвязаны следующие вопросы:

- Для любого ли $\varepsilon > 0$ существует полиномиальный алгоритм для задачи 32 «MAX-3SAT(ε)»?

- Можно ли при построении \mathcal{PCP} -системы для задачи 32 «MAX-3SAT(ε)» вероятностно выбирать одну дизъюнкцию и проверять ее выполнимость с помощью оракула, чтобы в случае, когда она выполнена, вероятность невыполнимости всей формулы уменьшилась на константу?

Размышляя над этими вопросами, рассмотрим два типа невыполнимых 3SAT-формул:

«Честные» — при любом выборе значений переменных невыполнимой сразу оказывается некоторая константная (скажем, ε) часть дизъюнкций, и, соответственно, вероятностная проверка любой выбранной дизъюнкции имеет константную вероятность выявить невыполнимую дизъюнкцию и «забраковать» всю формулу.

Также заметим, что для таких формул алгоритм, распознающий существование $\frac{1}{1-\varepsilon}$ -приближенного решения, будет, собственно, разрешающим алгоритмом.

«Хитрые» — в которых, например, при любом присваивании невыполнимой оказывается только одна дизъюнкция, и ее нельзя поймать с константной вероятностью, проверяя по одиночке случайно выбранные дизъюнкции.

Оказалось, есть путь вывести «хитрецов» на чистую воду. Это так называемые *усиливающие сводимости*¹⁴ (см. определение 6.3.18), представляющие собой сводимости по Карпу (см. определение 6.2.5 «Сводимость по Карпу») к «честным» 3SAT-задачам.

Определение 6.3.18. Усиливающая сводимость (*amplifying reduction*) по сведению произвольного языка $L \in \mathcal{NP}$ к языку 3SAT($\in \mathcal{NPC}$) для заданной константы $0 < \varepsilon < 1$ есть полиномиально вычислимая

¹⁴Пока еще нет устоявшегося русского перевода.

функция $\phi = f(x)$, преобразующая входное слово x в экземпляр (формулу ϕ) задачи 3SAT, для которой

$$\begin{aligned} x \in L &\iff \phi \in 3SAT, \\ x \notin L &\iff \phi \notin 3SAT, \end{aligned}$$

причем если $\phi \notin 3SAT$, то доля невыполненных (ложных) дизъюнкций будет не меньше ε .

Видно, что если усиливающие сводимости существуют, то их можно использовать для построения эффективных (особенно по запросам к оракулу) \mathcal{PCP} -систем.

Оказывается, да, такие сводимости существуют.

Теорема 39. $\mathcal{NP} \subseteq \mathcal{PCP}(\log, O(1))$ тогда и только тогда, когда существует усиливающая сводимость для 3SAT.

Доказательство. \Rightarrow : Рассмотрим $L \in \mathcal{NP}$. Нам дано, что для него существует $\mathcal{PCP}(\log, O(1))$ -система. Покажем существование усиливающей сводимости $L \rightarrow 3SAT$. Будем использовать те же обозначения (ВМТ, вероятностные строки, запросы/ответы оракула), как при доказательстве леммы 41 и теоремы 38.

Для каждого входного слова x (мы не показываем в индексах x , но неявно подразумеваем):

- существует полиномиальный (от длины входа, $2^{\log(|x|)}$) набор всех случайных строк $\langle r_1, \dots, r_m \rangle$;
- для каждой вероятностной строки r_i , запишем ответы оракула на возникшие у проверяющей ВМТ вопросы: $\langle \pi_1^i, \dots, \pi_t^i \rangle$.

Далее действуем так же, как в теореме 38, строим по «журналу» работы ВМТ-верификатора формулу ψ_i , выражающую его решение на i -м вероятностном наборе $\langle r_1, \dots, r_m \rangle$ и зависящую от t переменных $\langle \pi_1^i, \dots, \pi_t^i \rangle$. Как построить такую формулу за полиномиальное время, мы уже видели в теореме 29 « $SAT \in \mathcal{NPC}$ », здесь же построим ее через построение таблицы истинности (она будет полиномиального размера) и построении КНФ по этой таблице — тогда мы дополнительно получим, что в ней не больше, чем 2^t различных дизъюнкций.

После чего (т.к. дизъюнкции в формуле у нас еще могут содержать до t -переменных) преобразуем полученную КНФ ψ_i в ЗКНФ ϕ_i так же, как при доказательстве \mathcal{NP} -полноты задачи 28 «3SAT». При этом, т.к. каждая t -дизъюнкция ψ_i выражается с помощью введения дополнительных переменных конъюнкцией не более чем t дизъюнкций не более чем от трех переменных (назовем их 3-дизъюнкциями), получаем, что ϕ_i содержит не более чем $t \cdot 2^t$ 3-дизъюнкций.

Выполняя конъюнкцию над всеми ϕ_i , получаем ЗКНФ формулу, которую мы построили за полиномиальное время (напоминаем, что t — константа, и, следовательно, $t \cdot 2^t$ — тоже константа):

$$\phi \equiv \wedge_{i=1}^m \phi_i.$$

Осталось убедиться, что описанное преобразование $x \rightarrow \phi$ — действительно усиливающая сводимость.

Пусть $x \in L$, тогда есть оракул π , ответы $\langle \pi_1^i, \dots, \pi_t^i \rangle$ гарантированно убедят проверяющую ВМТ (при любой случайной последовательности $\langle r_1, \dots, r_m \rangle$), и эти ответы (+значения вспомогательных переменных) будут выполняющим набором для ϕ , т. е. $\phi \in 3SAT$.

Если же $x \notin L$, тогда для любого оракула π не меньше чем в половине исходов i проверяющая машина М «бракует» x , и, следовательно, должно быть невыполнимо не менее половины ϕ_i . Итак, общее число дизъюнкций в ϕ не больше $m \cdot t \cdot 2^t$, а число невыполненных дизъюнкций никак не меньше $\frac{m}{2}$, что не меньше доли $\varepsilon = \frac{1}{2 \cdot t \cdot 2^t}$.

\Leftarrow : Теперь пусть у нас есть ε -усиливающая сводимость $f : 3SAT \rightarrow 3SAT$, покажем, что выполняется условие теоремы 37 « \mathcal{PCP} -theorem». Рассмотрим любой $L \in NPC$, например, $3SAT$, и построим для него требуемую \mathcal{PCP} -систему.

Для входного слова — формулы ϕ — проверяющая ВМТ M будет работать следующим образом:

1. $\phi' = f(\phi)$;
2. случайно-равномерно выбираем $\lceil \frac{1}{\varepsilon} \rceil$ дизъюнкций для проверки. ($|r| = O(\log(|\phi'|)) = O(\log(|\phi|))$);
3. задаем не более $\lceil \frac{3}{\varepsilon} \rceil$ вопросов оракулу о том, какие значения нужно бы присвоить встретившимся в этих дизъюнкциях переменным для выполнимости формулы;
4. получив ответы, вычисляем конъюнкцию из выбранных дизъюнкций, бракуем ее, если она равна 0, и принимаем, если равна 1.

Проанализируем алгоритм M :

- Если формула выполнима, подходящий оракул покажет нам выполнимость любых выбранных нами дизъюнкций, и мы ее не забракуем — условие полноты («completeness») выполнено.
- Если формула невыполнима, то вероятность того, что мы $\lceil \frac{1}{\varepsilon} \rceil$ раз не угадаем ни одну невыполненную дизъюнкцию, будет не больше

$$(1 - \varepsilon)^{\lceil \frac{1}{\varepsilon} \rceil} \leq \frac{1}{e} \leq \frac{1}{2}.$$

т. е. выполнено и условие корректности («soundness»).



Из теоремы 39 и теоремы 37 « \mathcal{PCP} -theorem» немедленно следует существование усиливающей сводимости для задачи 3SAT, откуда следует важное утверждение о сложности аппроксимации (попросту говоря — неаппроксимируемости) задачи 32 «MAX-3SAT(ε)»:

Теорема 40. Для некоторой константы $\varepsilon > 0$ задача 32 «MAX-3SAT(ε)» \mathcal{NP} -трудна.

Доказательство. Покажем, что задача 32 «MAX-3SAT(ε)» \mathcal{NP} -трудна. Возьмем любую 3-КНФ ϕ , ($3SAT \in \mathcal{NPC}$) и с помощью ε -усиливающей сводимости сведем ее к ϕ^ε . Решив задачу 32 «MAX-3SAT(ε)» для ε из нашей сводимости, мы ответим на вопрос $\phi \stackrel{?}{\in} 3SAT$.

□

Таким образом, для задачи 3SAT доказано, что аппроксимация ее с произвольным ε не менее трудная задача, чем ее точное решение. Обратите внимание: «с произвольным ε »! Действительно, для некоторых ε задачу 3SAT удалось решить эффективно, т. е. полиномиально. Точнее, найден полиномиальный алгоритм для 3SAT аппроксимации с $\varepsilon = \frac{1}{8}$ для выполнимых 3SAT и доказано, что эта оценка — точная, т. е. аппроксимация 3SAT с точностью $\varepsilon < \frac{1}{8}$ есть \mathcal{NP} -трудная задача ([Hås97; KZ97]).

6.3.7 Класс \mathcal{APX} . Сводимости, сохраняющие аппроксимации

Многие известные оптимизационные задачи \mathcal{NP} -трудны, и хотя это означает, что при гипотезе $P \neq \mathcal{NP}$ точное решение получить эффективно невозможно, это ничего не говорит о возможности аппроксимации задачи, т. е. о нахождении приближенного решения, которого, возможно, будет достаточно во многих практических случаях. В частности, в последние годы для целого ряда \mathcal{NP} -трудных задач удалось построить полиномиальные алгоритмы, имеющие мультипликативную ошибку не более $1 + \varepsilon$ для любого фиксированного $\varepsilon > 0$. Для практических целей этого зачастую оказывается вполне достаточно, поэтому хотелось бы иметь некую классификацию задач по степени трудности поиска приближенных решений.

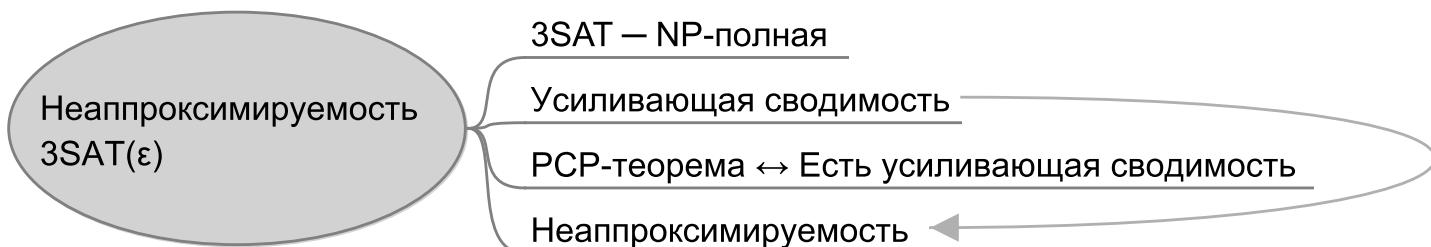


Рис. 6.16: Карта-памятка раздела 6.3.6

Ясно, однако, что хотя \mathcal{NP} -полные задачи сводятся друг к другу с помощью полиномиальной сводимости, они могут иметь различную сложность относительно нахождения приближенных решений. Для построения сводимостей, сохраняющих аппроксимации, требуется наложить более жесткие ограничения на такие сводимости. Это является косвенным объяснением того факта, что было предложено несколько сводимостей, сохраняющих аппроксимации, и далеко не сразу стали ясны их преимущества и недостатки. В настоящем параграфе мы кратко рассмотрим эти вопросы.

Прежде чем переходить к их описаниям, дадим формальное определение оптимизационной \mathcal{NP} -задачи.

Определение 6.3.19. *Оптимизационная NP проблема A — это четверка $(I, sol, m, type)$, такая, что:*

I — множество входов задачи A, распознаваемое за полиномиальное время.

sol(x) — обозначает множество допустимых решений для данного входа $x \in I$. Причем должен существовать полином p , такой, что для любого $y \in sol(x)$, $|y| \leq p(|x|)$ и для любых x и y с $|y| \leq p(|x|)$, за полиномиальное время можно проверить, принадлежит ли $y \in sol(x)$.

$m(x, y)$ — обозначает положительную целочисленную меру y (для входа x и соответствующего ему допустимого решения y), называемую также целевой функцией. Функция m должна быть вычислимая за полиномиальное время.

$type \in \{\max, \min\}$ — направление оптимизации.

Множество оптимизационных задач, для которых соответствующие задачи разрешения принадлежат классу \mathcal{NP} , обозначается через \mathcal{NPO} .

Чтобы прочувствовать данное определение, полезно проверить, как в таком виде представляется, например, рассмотренная нами ранее метрическая задача коммивояжера на минимум. С элементом $type$ все ясно — длину маршрута мы минимизируем. Входом I нашей задачи будет любой полный граф с заданными длинами ребер, причем должно выполняться неравенство треугольника. Ясно, что это можно проверить за полиномиальное время. Множество допустимых решений $sol(x)$ — это все гамильтоновы циклы графа с матрицей длин x . Полиномиальная ограниченность $y \in sol(x)$ очевидна: на самом деле длина битовой записи y не превосходит длины записи входа x . Проверить включение $y \in sol(x)$ можно эффективно естественным образом. Функция-мера $m(x, y)$ — это длина маршрута, вычисляемая как сумма длин входящих в маршрут ребер.

Определение 6.3.20. Пусть A — это \mathcal{NPO} -проблема. Для данного входа x и допустимого решения y для x мультипликативная ошибка решения y относительно x определяется следующим соотношением:

$$R(x, y) = \max \left\{ \frac{m(x, y)}{opt(x)}, \frac{opt(x)}{m(x, y)} \right\}.$$

Определение 6.3.21. Класс \mathcal{APX} состоит из всех оптимизационных задач, для которых существуют полиномиальные приближенные алгоритмы с мультипликативной ошибкой, не превышающей некоторой абсолютной константы (см. определение 2.1.1 « C -приближенный алгоритм»).

Класс \mathcal{APX} является естественным с теоретико-сложностной точки зрения классом достаточно хорошо приближаемых задач и, в свою очередь, содержит подкласс \mathcal{PTAS} .

Определение 6.3.22. Класс \mathcal{PTAS} состоит из всех оптимизационных задач, для которых существуют полиномиальные приближенные алгоритмы с мультипликативной ошибкой, не превышающей $1 + \varepsilon$ для любой константы $\varepsilon > 0$.

Рассмотрим понятие E -сводимости (error reducibility).

Определение 6.3.23. Пусть A и B — две \mathcal{NPO} проблемы. **E -сводимостью** задачи A к задаче B ($A \leq_E B$) называется пара функций f и g , вычислимых за полиномиальное время, и константа α со следующими свойствами:

- для любого входа $x \in I_A$, $f(x) \in I_B$ — вычислимо за полиномиальное время;
- для любого $x \in I_A$ и для любого $y \in sol_B(f(x))$, $g(x, y) \in sol_A(x)$ — вычислимо за полиномиальное время;
- для любого $x \in I_A$ и для любого $y \in sol_B(f(x))$:

$$R_A(x, g(x, y)) \leq 1 + \alpha(R_B(f(x), y) - 1).$$

Тройка (f, g, α) называется E -сводимостью от A к B и обозначается как « $A \leq_E B$ ».

Описанная сводимость сохраняет линейное отношение для оптимумов рассматриваемых задач. E -сводимость сохраняет свойство быть оптимальным решением: если y — оптимальное решение для входа $f(x)$, то $g(x, y)$ должно быть оптимальным решением для входа x .

Рассмотрим для иллюстрации простой пример для задач МАКСИМАЛЬНАЯ КЛИКА и МАКСИМАЛЬНОЕ НЕЗАВИСИМОЕ МНОЖЕСТВО. Напомним, что первая из этих задач заключается в нахождении в исходном графе максимального по числу вершин полного подграфа, а вторая — максимального по числу вершин пустого подграфа.

Определим E -сводимость от первой задачи ко второй. Для данного графа $G = (V, E)$, пусть $f(G) = \overline{G}$, где \overline{G} — граф, дополнительный к G . Ясно, что $V' \subseteq V$ — независимое множество в \overline{G} тогда и только тогда, когда V' — клика в G . Положив $g(G, V') = V'$, получим, что f и g удовлетворяют условиям E -сводимости с $\alpha = 1$.

Далеко не любая полиномиальная сводимость является сводимостью, сохраняющей аппроксимацию. Рассмотрим для иллюстрации еще один пример для задач МАКСИМАЛЬНАЯ КЛИКА и МИНИМАЛЬНОЕ ВЕРШИННОЕ ПОКРЫТИЕ. Определим полиномиальную сводимость от первой задачи ко второй. Для данного графа $G = (V, E)$, пусть $\overline{G} = (V, \overline{E})$, т. е. \overline{G} — граф, дополнительный к G . Ясно, что $V' \subseteq V$ — вершинное покрытие в \overline{G} тогда и только тогда, когда $V - V'$ — клика в G (если это не так, то в G есть непокрытое ребро). Положим $f(G) = \overline{G}$, $g(G, V') = V - V'$.

Упражнение 6.3.6. Покажите, что f и g не удовлетворяют условиям E -сводимости ни для какой константы $\alpha > 0$.

E -сводимости, на самом деле, являются сводимостями, сохраняющими аппроксимации.

Лемма 42. Если $A \leq_E B$ и $B \in \mathcal{APX}$, то $A \in \mathcal{APX}$. Если $A \leq_E B$ и $B \in \mathcal{PTAS}$, то $A \in \mathcal{PTAS}$.

Доказательство. Пусть (f, g, α) — указанная E -сводимость. Для данного входа $x \in I_A$ задачи A рассмотрим вход $f(x)$ задачи B и применим к нему r -приближенный алгоритм, который существует для задачи B для некоторой константы r . Получив некоторое решение y , построим по нему $g(x, y)$. Имеем

$$R_A(x, g(x, y)) \leq 1 + \alpha(R_B(f(x), y) - 1) \leq 1 + \alpha(r - 1).$$

Таким образом, мультипликативная ошибка построенного алгоритма ограничена константой $1 + \alpha(r - 1)$. Значит, $A \in \mathcal{APX}$.

При $r = 1 + \varepsilon$ получаем, что мультипликативная ошибка ограничена величиной

$$1 + \alpha(r - 1) = 1 + \alpha \cdot \varepsilon.$$

Значит, $A \in \mathcal{PTAS}$. □

Рассмотрим подкласс \mathcal{APX} , содержащий все задачи из \mathcal{APX} , все решения которых ограничены полиномом от длины входа (обозначение \mathcal{APX} РВ) ([[Kha+99](#)]).

Класс \mathcal{APX} РВ замкнут относительно E -сводимостей, и самые трудные задачи в этом классе — это \mathcal{APX} РВ-трудные задачи.

Определение 6.3.24. Задача называется \mathcal{APX} РВ-трудной, если любая задача из \mathcal{APX} РВ сводится к ней посредством E -сводимости.

Для многих задач была доказана их \mathcal{APX} РВ-трудность, в том числе для задач MAX-SAT, MAX-3SAT, MAX-2SAT, MAX-CUT, ВЕРШИННОЕ ПОКРЫТИЕ, метрической задачи коммивояжера на минимум и др. Справедливо следующее утверждение.

Лемма 43. Если хотя бы для одной из \mathcal{APX} РВ-трудных задач удастся построить \mathcal{PTAS} , то $\mathcal{P} = \mathcal{NP}$.

Это является прямым следствием леммы 42, несуществования \mathcal{PTAS} для задачи MAX-3SAT (см. теорему 40, следствие РСР-теоремы).

В заключение отметим важный факт, вытекающий из предыдущих рассмотрений. Чтобы показать несуществование \mathcal{PTAS} для некоторой задачи X (при условии $\mathcal{P} \neq \mathcal{NP}$), достаточно построить E -сводимость от некоторой \mathcal{APX} РВ-полной задачи к X . На этом мы заканчиваем краткий экскурс в теорию сводимостей, сохраняющих аппроксимации, однако для интересующихся историей вопроса приводим далее несколько более подробные комментарии и исторические замечания.

Сводимости, сохраняющие аппроксимации: немного истории. Здесь мы продолжаем обсуждение сводимостей, сохраняющих аппроксимации.

Как мы уже отмечали, сводимостей, сохраняющих аппроксимации, было предложено несколько, и далеко не сразу стали ясны их преимущества и недостатки.

Исторически, наибольшее продвижение в исследовании классов приближений было сначала достигнуто при синтаксическом взгляде на них.

Определение 6.3.25. Класс \mathcal{MAXSNP}_0 состоит из всех оптимизационных задач, которые могут быть определены в терминах выражения:

$$\max_S |\{(x_1, \dots, x_k) \in U^k : \phi(P_1, \dots, P_m, S, x_1, \dots, x_k)\}|,$$

где U — конечное множество, P_1, \dots, P_m и S — предикаты (т. е. отображения вида $U^r \rightarrow \{0, 1\}$ для некоторого натурального r) и ϕ — булево выражение, построенное из предикатов и переменных x_1, \dots, x_k . При этом k и r не зависят от входа, а U и P_1, \dots, P_m — зависят от входа.

Рассмотрим в качестве примера задачу МАКСИМАЛЬНЫЙ РАЗРЕЗ, в которой входом является граф $G = (V, E)$, и цель заключается в нахождении подмножества $V' \subseteq V$, максимизирующее число ребер с одной концевой вершиной в V' . Эта задача может быть записана в следующем виде:

$$\max_S |\{(x, y) : ((P(x, y) \vee P(y, x)) \wedge S(x) \wedge \neg S(y))\}|,$$

где V — универсум булевых переменных, $S : V \rightarrow \{0, 1\}$ и $P : V \times V \rightarrow \{0, 1\}$ — предикаты. Предположим, что граф $G = (V, E)$ — ориентированный. Описанная задача заключается в нахождении предиката, максимизирующего число пар (x, y) со свойством, что либо (x, y) , либо (y, x) является дугой (описанным

отношением P), а $S(x) = 1$ и $S(y) = 0$. Для такого предиката S , множество x , для которых $S(x) = 1$, задают максимальный разрез.

Важной компонентой классификации задач по трудности их аппроксимации является понятие L -сводимости (linear reducibility).

Определение 6.3.26. Пусть A и B — две оптимизационные NPO проблемы. L -сводимостью задачи A к задаче B называется пара функций f и g , вычислимых за полиномиальное время, со следующими двумя свойствами:

- Для любого входа $x \in I_A$ задачи A с величиной оптимума $OPT(x)$, $f(x) \in I_B$ — является входом задачи B с величиной оптимума $OPT(f(x))$, причем

$$OPT(f(x)) \leq \alpha \cdot OPT(x),$$

для некоторой положительной константы α .

- Для любого допустимого решения $y \in sol_B(f(x))$ для входа $f(x)$, $g(y)$ — является допустимым решением для входа $x \in I_A$ таким, что

$$|OPT(x) - m(x, g(y))| \leq \beta \cdot |OPT(f(x)) - m(f(x), y)|,$$

для некоторой положительной константы β ,

где $m(x, g(y))$ и $m(f(x), y)$ — значение целевой функции для допустимых решений $g(y)$ и y соответственно.

Описанная сводимость сохраняет линейное отношение как для оптимумов рассматриваемых задач, так и для абсолютных ошибок. L -сводимость сохраняет свойство быть оптимальным решением: если y — оптимальное решение для входа $f(x)$, то $g(y)$ должно быть оптимальным решением для входа x .

Определение 6.3.27. Класс \mathcal{MAP} определяется как класс всех оптимизационных проблем, L -сводимых к какой-нибудь проблеме из \mathcal{MAP}_0 .

Нетрудно проверить справедливость следующего утверждения.

Лемма 44. Если (f, g) является L -сводимостью задачи A к задаче B , а (f', g') — L -сводимостью задачи B к задаче C , то их композиция $(f \cdot f', g \cdot g')$ является L -сводимостью задачи A к задаче C .

Доказательство. Действительно, если x — вход задачи A , то $f(x)$ — вход задачи B , причем

$$OPT(f'(f(x))) \leq \alpha' \cdot OPT(f(x)) \leq \alpha' \alpha \cdot OPT(x).$$

Аналогично можно доказать, что второе неравенство выполнено с константой $\beta \cdot \beta'$. \square

Пападимитриу и Яннакакис ([PY91]) доказали, что L -сводимости на самом деле являются сводимостями, сохраняющими аппроксимации. Точнее, доказано следующее утверждение.

Лемма 45. Если имеется L -сводимость от оптимизационной задачи A к задаче B с параметрами α, β и существует полиномиальный алгоритм для задачи B с мультипликативной ошибкой не более $1 + \varepsilon$, то существует полиномиальный алгоритм для задачи A с мультипликативной ошибкой не более $1 + \varepsilon\beta\alpha$ для задач на минимум и с ошибкой не более $1 + \frac{\varepsilon\beta\alpha}{1 - \varepsilon\beta\alpha}$ — для задач на максимум.

Другими словами, если существует \mathcal{PTAS} для B , то существует \mathcal{PTAS} и для A !

Доказательство. Для данного входа $x \in I_A$ задачи A рассмотрим вход $f(x)$ задачи B и применим к нему $(1 + \varepsilon)$ -приближенный алгоритм. Получив некоторое решение y , построим по нему $g(y)$. Имеем для задачи минимизации:

$$\begin{aligned} |OPT(x) - m(x, g(y))| &\leq \beta \cdot |OPT(f(x)) - m(f(x), y)| \leq \\ &\leq \beta\varepsilon \cdot OPT(f(x)) \leq \beta\varepsilon\alpha \cdot OPT(x). \end{aligned}$$

□

Класс MAXNP замкнут относительно L -сводимостей, и самые трудные задачи в этом классе — это MAXNP -трудные задачи.

Определение 6.3.28. Задача называется MAXNP -трудной, если любая задача из MAXNP сводится к ней посредством L -сводимости.

Для многих задач была доказана их MAXNP -трудность, в том числе для задач MAX-SAT, MAX-3SAT, MAX-2SAT, MAX-CUT, ВЕРШИННОЕ ПОКРЫТИЕ и др. Если хотя бы для одной из MAXNP -трудных задач удастся построить PAS , то $\mathcal{P} = \mathcal{NP}$. Так же, как и в случае с E -сводимостями, это является прямым следствием леммы 45, несуществования PAS для задачи MAX-3SAT (как следствие PCP-теоремы).

К недостаткам L -сводимостей можно отнести тот факт, что класс APX не замкнут относительно них. Другими словами, существуют две задачи на максимум в классе NPO , такие, что $A \notin \text{APX}$, $B \in \text{APX}$ и $A \leq_L B$.

Оказывается, внутри класса APX определенная ранее E -сводимость является просто обобщением L -сводимости.

Лемма 46. Для любых двух NPO -задач, если $A \leq_L B$ и $A \in \text{APX}$, то $A \leq_E B$.

Определим замыкание класса MAXNP как множество всех NPO задач, которые E -сводятся к некоторой задаче из MAXNP .

Было доказано ([Kha+99]), что замыкание класса MAXNP совпадает с подклассом APX , содержащим все задачи из APX , все решения которых ограничены полиномом от длины входа (обозначение APXPB). В частности, следствием этого результата явился тот факт, что любой результат о MAXNP -полноте автоматически транслировался в утверждение об APX -полноте. Таким образом, задачи, полные в MAXNP относительно L -сводимостей, оказались полными в APXPB относительно E -сводимостей.

Несколько позднее, с использованием еще более общего понятия AP-сводимостей, удалось доказать, что замыкание класса \mathcal{MAXSNP} совпадает с \mathcal{APX} .

В настоящее время уже нет нужды использовать класс \mathcal{MAXSNP} , равно как и L -сводимости, поскольку класс \mathcal{APX} и AP-сводимости полностью их покрывают. Однако некоторые традиции и ряд полученных ранее результатов заставляют зачастую обращаться к этим понятиям. Кроме того, L -сводимости бывает проще строить, чем AP-сводимости.

В заключение приведем определение AP-сводимости (*approximation preserving reducibility*). На сегодняшний день эти сводимости являются наиболее общим типом сводимостей, сохраняющих аппроксимации.

Определение 6.3.29. Пусть A и B — две NPO проблемы. AP-сводимостью задачи A к задаче B ($A \leq_{AP} B$) называется пара функций f и g и константа α , такие что:

- Для любого входа $x \in I_A$ и для любого $r > 1$, $f(x, r) \in I_B$ — вычислимо за время $t_f(|x|, r)$.
- Для любого $x \in I_A$ и для любого $r > 1$ и для любого $y \in sol_B(f(x, r))$, $g(x, y, r) \in sol_A(x)$ — вычислимо за время $t_g(|x|, |y|, r)$.
- Для любого фиксированного r обе функции $t_f(\cdot, r)$ и $t_g(\cdot, \cdot, r)$ ограничены сверху некоторым полиномом.
- Для любого фиксированного n обе функции $t_f(n, \cdot)$ и $t_g(n, n, \cdot)$ являются невозрастающими функциями.
- Для любого $x \in I_A$, для любого $r > 1$ и для любого $y \in sol_B(f(x, r))$,

$$R_B(f(x, r), y) \leq r \times R_A(x, g(x, y, r)) \leq 1 + \alpha(r - 1).$$

Тройка (f, g, α) называется *AP-сводимостью от A к B*.

Нетрудно убедиться, что *AP-сводимости* являются обобщением *E-сводимостей* (проделайте это в качестве упражнения). Более того, в отличие от *E-сводимости* любая задача из *PTAS* *AP-сводится* к любой *NPO* задаче.

6.4 Схемы и схемная сложность

Этот раздел основан на соответствующей главе [КШВ99].

Схема (булева)¹⁵ — это способ вычислить функцию $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$.

Помимо исходных переменных x_1, \dots, x_n , для которых вычисляется значение f , схема использует некоторое количество вспомогательных переменных y_1, \dots, y_s и некоторый набор (базис) булевых функций \mathcal{F} .

Схема S в базисе \mathcal{F} определяется последовательностью присваиваний Y_1, \dots, Y_s .

Каждое присваивание Y_i имеет вид

$$y_i := f_j(u_{k_1}, \dots, u_{k_r}),$$

где $f_j(\cdot) \in \mathcal{F}$, а переменная u_{k_p} ($1 \leq p \leq r$) — это либо одна из исходных переменных x_t ($1 \leq t \leq n$), либо вспомогательная переменная y_l с меньшим номером ($1 \leq l < i$).

Таким образом, для каждого набора значений исходных переменных последовательное выполнение присваиваний, входящих в схему, однозначно определяет значения всех вспомогательных переменных. Результатом вычисления считаются значения последних m переменных y_{s-m+1}, \dots, y_s .

Схема вычисляет функцию f , если для любых значений x_1, \dots, x_n результат вычисления — $f(x_1, \dots, x_n)$.

¹⁵В русскоязычной литературе часто используется термин — схемы из функциональных элементов.

Определение 6.4.1. Схема называется *формулой*, если каждая вспомогательная переменная используется в правой части присваиваний только один раз.

Обычные математические формулы именно так задают последовательность присваиваний: «внутри» формул не принято использовать ссылки на их части или другие формулы.

Схему можно также представлять в виде ориентированного ациклического графа, у которого

- вершины входной степени 0 (*входы*) помечены исходными переменными;
- остальные вершины (*функциональные элементы*) помечены функциями из базиса;
- дуги помечены числами, указывающими номера аргументов;
- вершины выходной степени 0 (*выходы*) помечены переменными, описывающими результат работы схемы.

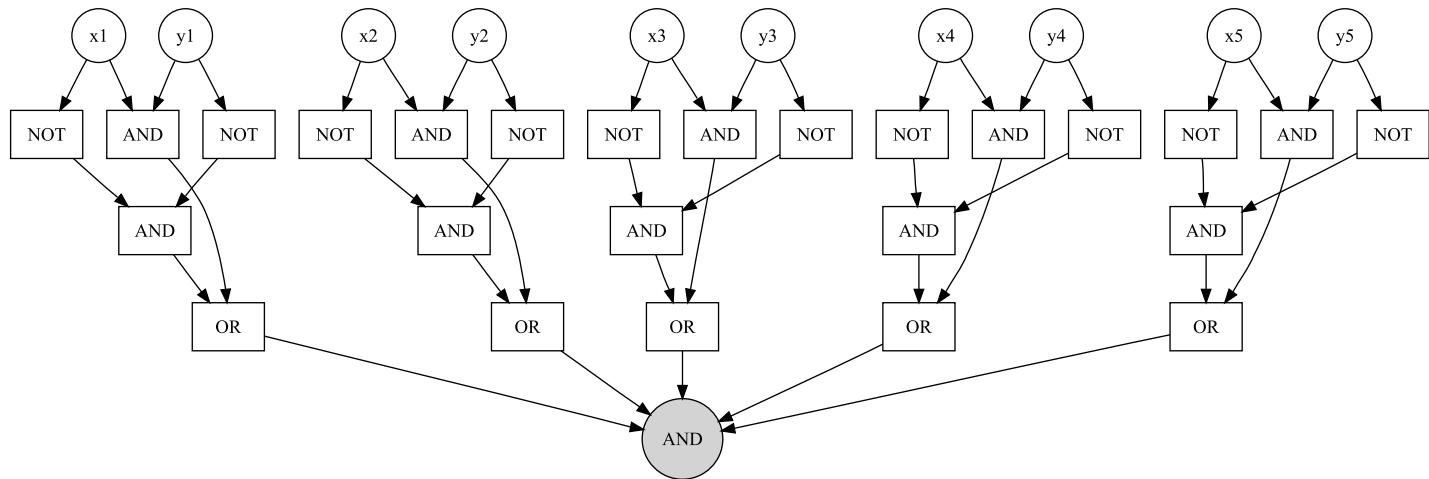


Рис. 6.17: Пример схемы: сравнение двух строк

Вычисление на графе определяется индуктивно: как только известны значения всех вершин y_1, \dots, y_{k_v} , дуги из которых ведут в данную вершину v , вершина v получает значение $y_v = f_v(y_1, \dots, y_{k_v})$, где f_v – базисная функция, которой помечена вершина.

При переходе к графу схемы мы опускаем *несущественные присваивания*, которые ни разу не используются на пути к выходным вершинам, так что они никак не влияют на результат вычисления.

Определение 6.4.2. Базис называется **полным**, если для любой булевой функции f есть схема в этом базисе, вычисляющая f .

Ясно, что в полном базисе можно вычислить произвольную функцию $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$ (такую функцию можно представить как упорядоченный набор из m булевых функций).

Булева функция может быть задана таблицей значений. Приведем таблицы значений для трех функций

$$\begin{aligned} NOT(x) &= \neg x, \\ OR(x_1, x_2) &= x_1 \vee x_2, \\ AND(x_1, x_2) &= x_1 \wedge x_2, \end{aligned}$$

(отрицание, дизъюнкция, конъюнкция), образующих полный базис, который будем считать стандартным. В дальнейшем имеются в виду схемы именно в этом базисе, если явно не указано что-либо иное.

x	NOT	x_1	x_2	OR	x_1	x_2	AND
0	1	0	0	0	0	0	0
1	0	0	1	1	0	1	0
		1	0	1	1	0	0
		1	1	1	1	1	1

Конъюнкция и дизъюнкция определяются для произвольного числа n булевых переменных аналогичным образом: конъюнкция равна 1 только тогда, когда все аргументы равны 1, а дизъюнкция равна 0 только тогда, когда все аргументы равны 0. В стандартном базисе они очевидным образом вычисляются схемами (и даже формулами), содержащими $n - 1$ элементарных двухходовых операций (конъюнкций или дизъюнкций).

Теорема 41. Базис $\{NOT, OR, AND\}$ — полный.

Доказательство. Литералом будем называть переменную или ее отрицание. Конъюнкцией литералов (это схема и даже формула) легко представить функцию $\chi_u(x)$, которая принимает значение 1 ровно один раз: при $x = u$. Если $u_i = 1$, включаем в конъюнкцию переменную x_i , если $u_i = 0$, то включаем в конъюнкцию $\neg x_i$.

Произвольная функция f может быть представлена в виде

$$f(x) = \bigvee_{u: f(u)=1} \chi_u(x). \quad (6.3)$$

В таком случае говорят, что f представлена в дизъюнктивной нормальной форме (ДНФ), т. е. как дизъюнкция конъюнкций литералов.¹⁶

Как уже говорилось, дизъюнкция нескольких переменных выражается формулой в стандартном базисе. \square

Определение 6.4.3. Размером схемы называется количество присваиваний в схеме.

Определение 6.4.4. Глубиной схемы называется максимальное число элементов на пути от входов к выходу.

Определение 6.4.5. Минимальный размер схемы в базисе \mathcal{F} , вычисляющей функцию f , называется схемной сложностью функции f в базисе \mathcal{F} и обозначается $c_{\mathcal{F}}(f)$.

Переход от одного полного конечного базиса к другому полному конечному базису меняет схемную сложность функций на множитель $O(1)$. Так что в асимптотических оценках выбор конкретного полного базиса неважен и поэтому будем использовать обозначение $c(f)$ для схемной сложности f в конечном полном базисе.

¹⁶Далее нам еще потребуется и конъюнктивная нормальная форма (КНФ) — конъюнкция дизъюнкций литералов.

Каждый предикат f на множестве $\{0, 1\}^*$ задает последовательность булевых функций $f_n: \{0, 1\}^n \rightarrow \{0, 1\}$ следующим образом (справа стоит предикат f):

$$f_n(x_1, x_2, \dots, x_n) = f(x_1, x_2, \dots, x_n).$$

Определение 6.4.6. Предикат f принадлежит классу P/poly , если

$$c(f_n) = \text{poly}(n).$$

Теорема 42. $P \subset \text{P/poly}$.

Доказательство. Если МТ работает за полиномиальное время, то и память, которую она использует, ограничена полиномом. Поэтому весь процесс вычисления на входном слове x длины n можно представить таблицей вычисления размера $T \times S$, где $T = \text{poly}(n)$, $S = \text{poly}(n)$.

$t = 0$		$\Gamma_{0,1}$			
$t = 1$					
...					
$t = j$		Γ'_{left}	Γ'	Γ'_{right}	
$t = j + 1$			Γ		
...					
$t = T$...


 S клеток

Строка с номером j таблицы задает состояние МТ после j тактов работы. Символы $\Gamma_{j,k}$, записанные в таблице, принадлежат алфавиту $\Sigma \times \{\emptyset \cup Q\}$. Символ $\Gamma_{j,k}$ определяет пару (символ, записанный в k -й ячейке после j тактов работы; состояние управляющего устройства после j тактов работы, если головка находится над k -й ячейкой, в противном случае второй элемент пары — \emptyset). Для простоты также считаем, что если вычисление заканчивается при некотором входе за $T' < T$ тактов, то строки с номерами, большими T' , повторяют строку с номером T' .

Построить схему, вычисляющую значения предиката на словах длины n , можно следующим образом. Состояние каждой клетки таблицы можно закодировать конечным (не зависящим от n) числом булевых переменных. Имеются локальные правила согласования, т. е. состояние каждой клетки Γ в строке ниже нулевой однозначно определяется состояниями клеток в предыдущей строке, лежащих непосредственно над данной (Γ'), левее данной (Γ'_{left}) и правее данной (Γ'_{right}). Каждая переменная, кодирующая состояние клетки Γ , есть функция от переменных, кодирующих состояния клеток $\Gamma'_{left}, \Gamma', \Gamma'_{right}$. Все эти функции могут быть вычислены схемами конечного размера. Объединяя эти схемы, получим схему, вычисляющую все переменные, кодирующие состояния клеток таблицы; размер этой схемы будет $O(ST) = O(n^{O(1)})$.

Осталось заметить, что переменные, кодирующие часть клеток нулевой строки, определяются входным словом, а переменные, кодирующие остальные клетки нулевой строки, являются константами. Чтобы узнать результат вычисления, нужно определить символ, записанный в нулевой ячейке ленты в конце вычисления.

Без ограничения общности можно считать, что состояния клеток таблицы кодируются так, что одна из кодирующих переменных равна 1 только в том случае, когда в ячейке записана 1. Тогда значение этой переменной для кода $\Gamma_{T,0}$ и будет результатом вычисления. \square

Класс P/poly шире класса P . Любой функции от натурального аргумента $\varphi(n)$ со значениями в $\{0, 1\}$ можно сопоставить предикат f_φ по правилу $f_\varphi(x) = \varphi(|x|)$, где $|x|$ обозначает длину слова x . Ограничение

ние такого предиката на слова длины n тождественно равно 0 или 1 (в зависимости от n). Схемная сложность таких функций ограничена константой. Поэтому все такие предикаты по определению принадлежат P/poly , хотя среди них есть и неразрешимые предикаты.

Справедливо следующее усиление теоремы.

Теорема 43. f принадлежит P тогда и только тогда, когда

1. $f \in \text{P/poly}$;
2. существует МТ, которая для входа n за время $\text{poly}(n)$ строит схему вычисления f_n .

Доказательство. \implies Данное в доказательстве теоремы 42 описание нетрудно превратить в МТ, которая строит схему вычисления f_n за полиномиальное по n время (схема f_n имеет простую структуру: каждая переменная связана с предыдущими одними и теми же правилами согласования).

\impliedby Столь же просто. Вычисляем размер входного слова. Затем строим по этому размеру схему $S_{|x|}$ вычисления $f_{|x|}$, используя указанную в условии 2) машину. После этого вычисляем $S_{|x|}(x)$ на машине, которая по описанию схемы и значениям входных переменных вычисляет значение схемы за полиномиальное от длины входа время. \square

Упражнение 6.4.1. Пусть c_n есть максимум сложности $c(f)$

по всем булевым функциям f от n переменных. Докажите, что $1,99^n < c_n < 2,01^n$ при достаточно больших n .

Упражнение 6.4.2. Покажите, что любую функцию можно вычислить схемой глубины не более 3 из элементов NOT и из элементов AND и OR с произвольным числом входов.

Упражнение 6.4.3. Докажите, что если из схемы глубины $O(\log n)$, вычисляющей $f: \{0, 1\}^n \rightarrow \{0, 1\}^m$, выбросить все несущественные присваивания, то полученная схема имеет полиномиальный по $n + m$ размер.

Упражнение 6.4.4. Постройте схему, которая сравнивает два n -битовых числа и имеет размер $O(n)$, а глубину $O(\log n)$.

Упражнение 6.4.5. 1. Постройте схему сложения двух n -битовых чисел размера $O(n)$.

2. Тот же вопрос, если дополнительно потребовать, чтобы глубина схемы была $O(\log n)$.

Упражнение 6.4.6. Функция $MAJ \{0, 1\}^n \rightarrow \{0, 1\}$ равна 1 на двоичных словах, в которых число единиц больше числа нулей, и 0 — на остальных словах. Постройте схему, вычисляющую эту функцию, размер схемы должен быть линеен по n , глубина — $O(\log n \log \log n)$.

Упражнение 6.4.7. Постройте схему размера $\text{poly}(n)$ и глубины $O(\log^2 n)$, которая проверяет, связаны ли путём две вершины в графе. Граф на m вершинах, которые помечены числами от 1 до m , задаётся $n = m(m - 1)/2$ булевыми переменными. Переменная x_{ij} , где $i < j$, определяет, есть ли в графе ребро, соединяющее вершины i и j .

Упражнение 6.4.8. Пусть схема глубины 3 из элементов NOT и из элементов AND и OR с произвольным числом входов вычисляет сложение n битов по модулю 2 (функция $PARTITY$). Покажите, что размер схемы не меньше c^n для некоторого $c > 1$.

Упражнение 6.4.9. Пусть $f_1, f_2, \dots, f_n, \dots$ — последовательность булевых функций от $1, 2, \dots, n, \dots$ аргументов. Покажите, что следующие два свойства равносильны:

1. существует последовательность вычисляющих эти функции формул, размер которых не превосходит полинома от n ;
2. существует последовательность вычисляющих эти функции схем глубины $O(\log n)$ из элементов NOT , AND и OR (с двумя входами).

Упражнение 6.4.10. Докажите, что существует разрешимый предикат, который принадлежит P/poly , но не принадлежит P .

6.5 Коммуникационная сложность

С появлением и развитием телекоммуникационных сетей и распределенных вычислений стали возникать новые типы ресурсных ограничений — коммуникационные.

О них не могло быть и речи, пока компьютеры были вне сети — хватало рассмотренных в разделе 6.1.2 временных и пространственных ресурсных ограничений и соответствующих мер сложности задач. Объединение компьютеров в локальные сети также редко приводило к тому, что «бутылочным горлышком» при решении какой-либо задачи являлись именно пропускная способность сети или высокая стоимость трафика.

Однако с появлением глобальных сетей стали возникать ситуации, когда, например, несколько мощных научных центров, обладающих огромными вычислительными ресурсами, пытаются объединить усилия для решения некоторой вычислительной задачи, либо когда филиалам транснациональной корпорации требуется выполнить, возможно, алгоритмически несложные действия, но над распределенными базами данных, при этом основной стоимостью становится стоимость коммуникаций. Как напрямую, путем оплаты услуг провайдера, так и опосредовано — когда передача больших объемов данных занимает много времени и тормозит вычисления.

Таким образом, появилось новое ресурсное ограничение — *стоимость коммуникации*, и, соответственно, появилась новая мера сложности алгоритмических задач — *коммуникационная сложность*.

При исследовании коммуникационной сложности задачи полагают, что входные данные некоторым образом распределены между $n > 1$ участниками, каждый из которых обладает неограниченными вычислительными ресурсами, и необходимо установить нижние и верхние оценки для трафика (объема со-

Рис. 6.18: Пример коммуникационного протокола для « $x_1x_2 = ? y_1y_2$ »

общений), необходимого для решения задачи. Существуют различные модели коммуникации, обуславливающие различные меры коммуникационной сложности: модели с произвольным числом участников, модели с произвольным распределением данных и т. п. Наиболее стандартной и классической является модель с двумя участниками и симметричным распределением между ними входных данных¹⁷.

Их задача — совместно вычислить некоторую булеву функцию $f(x_1, \dots, x_n)$. Участники по очереди обмениваются сообщениями фиксированной длины (как вариант — однобитовыми сообщениями), пока одним из участников не будет получен ответ, т. е. будет вычислена f .

Разумеется, участники могут применять различные алгоритмы вычисления и разные алгоритмы обмена сообщениями, которые называются **протоколами**. Каждый протокол обмена однобитными сообщениями можно представить как дерево, где каждая дуга помечена коммуникационным битом, пересылаемым участником, а узлы-листья помечены вычисленным ответом (см. рис. 6.18). Длиной протокола считается длина самого длинного пути в таком дереве.

Минимальное число бит, которыми нужно обменяться участникам для вычисления f , и называется **коммуникационной сложностью**. Более формально:

Определение 6.5.1. Коммуникационная сложность вычисления функции f — это минимум по длинам всех протоколов вычисления f .

Коммуникационная сложность оценивается в терминах $O(f(n))$, где n — длина входа (например, число аргументов вычисляемой булевой функции). Сразу заметим, что для любой задачи такого рода суще-

¹⁷Как и во многих задачах, связанных с коммуникациями (например, в криптографических постановках), этих двух участников зовут Алиса и Боб.

ствует тривиальное решение, заключающееся в пересылке одним участком другому всех своих данных — $O(n)$. Это является верхней оценкой коммуникационной сложности для задач с двумя участниками, поэтому интерес составляет получение более низких оценок — полилогарифмических или константных.

Приведем примеры нескольких задач с установленными для них оценками коммуникационной сложности.

Задача 33. «ЧЕТНОСТЬ». Алиса и Боб имеют по битовой строке длины n . Необходимо вычислить четность числа битов строки-конкatenации.

Легко видеть, что коммуникационная сложность задачи 33 равна $O(1)$. Действительно, Алисе достаточно вычислить четность своей строки (один бит) и передать ее Бобу.

Задача 34. «СРАВНЕНИЕ». Алиса и Боб имеют битовые строки длины n : $X = (x_1, \dots, x_n)$ и $Y = (y_1, \dots, y_n)$ соответственно. Нужно сравнить эти строки (проверить на эквивалентность).

Оказывается (доказано), что нижние оценки для задачи 34 совпадают с тривиальными верхними оценками, и, таким образом, коммуникационная сложность точного решения задачи 34 есть $\Theta(n)$.

Задача 35. «СУММА БИТ». Алиса и Боб имеют по битовой строке длины n . Одинаково ли число единиц в битовых строках?

Простой алгоритм для задачи 35: «Алиса высыпает Бобу сумму единиц в своей строке», имеет коммуникационную сложность $O(\log n)$.

С другой стороны, доказано (здесь мы не будем касаться методик получения нижних оценок), что этот алгоритм также является оптимальным.

Приведем задачу, в которой, правда, вычисляется не булевая функция, а число, но оптимальный алгоритм в которой менее тривиален.

Задача 36. «МЕДИАНА».

Алиса и Боб имеют по одному подмножеству множества $(1, 2, \dots, n)$. Необходимо найти элемент-медиану¹⁸ в объединении этих подмножеств.

Упражнение 6.5.1. Постройте алгоритм для задачи 36 с коммуникационной сложностью $O(\log^2 n)$.

Упражнение 6.5.2. Постройте алгоритм для задачи 36 с коммуникационной сложностью $O(\log n)$.

Кроме коммуникационной сложности точного решения задачи и детерминированных алгоритмов рассматривают (аналогично с разделом 4) коммуникационную сложность вероятностных алгоритмов. т. е. участники могут применять вероятностные алгоритмы, каждый из которых использует либо общий для всех источник случайных чисел (*public coins*, менее распространенная модель), либо собственный датчик-источник случайных чисел (*private coins*, более распространенная модель, далее будем рассматривать только ее).

Допустима некоторая вероятность P_{err} ошибки, а количество переданных бит измеряется так же, как и в детерминированном случае.

Так же, как и в случае с временной сложностью, применение вероятностных алгоритмов может дать существенный выигрыш и в коммуникационной сложности.

Итак, рассмотрим задачу 34 (СРАВНЕНИЕ) и попробуем получить алгоритм, более коммуникационно-эффективный, чем тривиальный. Выберем случайное простое p из интервала $[1 \dots m]$, m — целое. Обозначим

$$\begin{aligned} p(X) &= X \bmod p, \\ p(Y) &= Y \bmod p. \end{aligned}$$

¹⁸Элемент, занимающий $\lceil m/2 \rceil$ место в упорядоченном массиве из m элементов.

Если строки X и Y совпадают, то, очевидно, $p(X) = p(Y)$. Ошибка, т. е. X и Y не совпадают при $p(X) = p(Y)$, произойдет, если $|X - Y|$ делится на p без остатка. Оценим ее вероятность.

Обозначим через $\pi(N)$ число простых чисел, не превосходящих N . Известна следующая оценка: $\pi(N) \sim \frac{N}{\ln N}$. Так как длина битовых строк равна n , то $|X - Y| \leq 2^n$, и справедлива

Лемма 47. Число различных простых делителей любого числа, меньшего 2^n , не превосходит n .

Пусть $A(n)$ — число простых, делящих $X - Y$. Тогда вероятность ошибки равна отношению числа простых, делящих $X - Y$, к числу всех простых из интервала $[1 \dots m]$, то есть

$$P_{err} = \frac{A(n)}{\pi(m)} \leq \frac{n}{\pi(m)} \sim \frac{n \ln m}{m},$$

причем $m > n$.

Для того чтобы вероятность ошибки была мала, m должно быть достаточно велико. Пусть $m = n^c$, $c = const$, тогда

$$P_{err} = \frac{nc \ln n}{n^c} = \frac{c \ln n}{n^{c-1}}.$$

Видно, что при больших n , P_{err} очень мала. Например, если $c = 2$, для $n = 10^5$ будет $m = 10^{10}$, а

$$P_{err} \leq \frac{n \ln m}{m} = \frac{2 \ln n}{n} \approx 2.3 \cdot 10^{-4}.$$

При этом для сравнения двух строк достаточно переслать выбранный простой делитель ($\leq m$) и остаток от деления ($< m$), т. е. потратить не более $2 \log m = O(\log n)$ бит.

6.6 Диаграмма классов сложности

В этом разделе постараемся кратко резюмировать информацию по классам сложности задач, встречавшихся в разделе 6.

Мы свели их в единую диаграмму на рис. 6.19, где стрелками изобразили отношения вложенности, а жирными линиями — эквивалентность.

Разумеется, мы не ставили целью отобразить все классы сложности — это вряд ли возможно на одной диаграмме, т.к. сейчас насчитывается не менее четырехсот различных классов сложности (см. [Aar07], «Зоопарк Классов Сложности», где упоминается о примерно пятистах классах).

Упражнение 6.6.1. Покажите, что если $\mathcal{NP} \subseteq \text{co}\mathcal{RP}$, то $\mathcal{NP} \subseteq \text{co}\mathcal{ZPP}$.

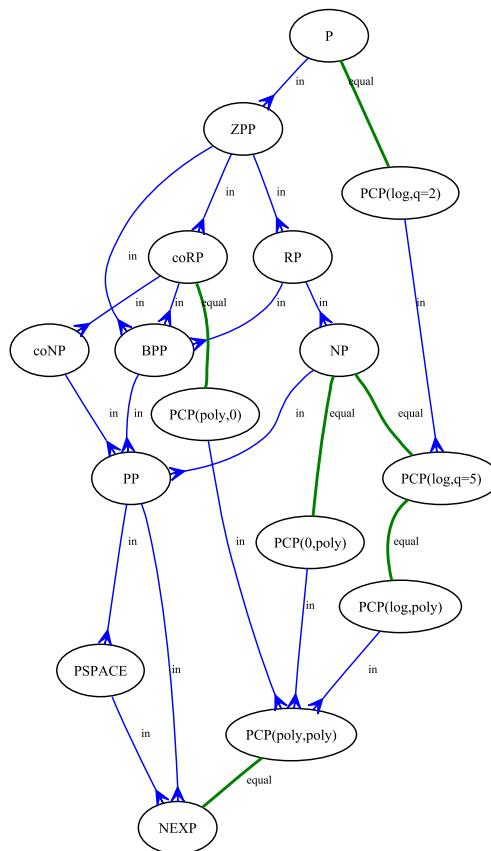


Рис. 6.19: Иерархия некоторых классов сложности

Заключение

Книга основана на двух курсах лекций, читавшихся авторами в течение нескольких лет для студентов 4-го и 6-го курсов Московского физико-технического института: «Сложность комбинаторных алгоритмов» и «Эффективные алгоритмы». Основное внимание удалено рассмотрению вычислительно трудных задач и современных подходов к их решению.

Книга состоит из двух частей: первая посвящена методам разработки и анализа алгоритмов решения конкретных задач, вторая — теории сложности.

В первой части описаны современные подходы к решению вычислительно трудных задач — разработка эффективных приближенных алгоритмов с оценками точности, вероятностных алгоритмов, алгоритмов, эффективных при анализе в среднем.

Во второй части представлены классические понятия сложности вычислений, классы сложности, теория \mathcal{NP} -полноты, \mathcal{PCP} -теорема и ее следствия для доказательства неаппроксимируемости ряда задач, понятия сводимостей, сохраняющих аппроксимации.

Книга предназначена для студентов и аспирантов, специализирующихся по прикладной математике.

Глава 7

Приложения

7.1 Введение в Python



Краткое введение в язык Python для понимания представлений на нем алгоритмов.

Вообще, трудно найти другой такой язык, одновременно пригодный для обучения, с одной стороны, и достаточно мощный для реализации реальных приложений (включая научные) — с другой. Дело в том, что автор Python'a (Guido van Rossum) долгое время участвовал в проекте создания языка обучения программированию ABC, который должен был заменить BASIC, и вынес из проекта несколько ценных идей, как минимизировать синтаксические накладные расходы, которые необходимы в реальном языке программирования (сложные декларации, открытия/закрытия блоков), избежав при этом каббалистического минимализма — понятной только adeptам магии спецсимволов типа «%\$^») Perl'о-подобных языках.

В Python'е была применена идея синтаксического выделения блоков с помощью отступов (в пределах блока отступ должен быть одинаков), кроме того, операторы, начинающие вложенный блок, должны для повышения читаемости отделяться двоеточием:

```
if len(EP)>0:
    for i in xrange(len(EP)):
        if G.degree(EP[i]) > 0:
            v = EP[i]
            break
else:
    v = G.nodes()[0] # добавляем первую
    EP.append(v)      # попавшуюся вершину в EP
    i = 0
```

Далее, все переменные в Python'е являются однотипными ссылками на объекты. Таким образом, устраняется необходимость декларации переменных, переменная определяется в момент первого присваивания, и тип хранимого в ней объекта может быть переопределен последующими присваиваниями.

```
def euler_circuit(G):
    EP = [] # Эйлеров цикл — массив вершин .
```

Кстати, можно присваивать одно и то же значение нескольким переменным одновременно:

```
>>> x = y = z = 0 # Обнулим переменные x, y, z
>>> x
0
```

Также интерпретатор берет на себя все управление памятью (резервирование, сборка мусора и т. п.).

Непосредственно в язык встроены не только примитивные типы данных, такие, как числа и строки, но и более сложные структуры. Например, так выглядят кортежи (*tuples*):

```
>>> a = (3, 2, 1)
>>> print a[0]
3
>>> print a[2]
1
>>>
```

К сожалению, индексы в *Python*, так же, как и в *C*, начинаются с нуля, что несколько снижает наглядность алгоритмов, ведь людям привычно начинать отсчет с единицы, а не нуля.

Эффективно реализованные словари или хэши (*hash*):

```
>>> passwords={'stas': 'mysecret555', 'olga': 'mobydick'}
>>> print passwords['stas']
mysecret555
>>> print passwords['olga']
mobydick
>>> passwords['oleg'] = 'hellraiser'
>>> print passwords['oleg']
hellraiser
```

Списки или последовательности (*list*) (включая операции «среза» и др.):

```
>>> a = [1, 2, 4, 8, 16, 32]
```

```
>>> print a[3]
8
>>> print a[3:]
[8, 16, 32]
>>> print a[:3]
[1, 2, 4]
>>> print a[1:3]
[2, 4]
>>> a.append(64)
>>> print a
[1, 2, 4, 8, 16, 32, 64]
>>> a.insert(0, -22)
>>> print a
[-22, 1, 2, 4, 8, 16, 32, 64]
```

Вообще, Python — современный, очень популярный, красивый, мощный и удобный интерпретируемый, расширяемый и встраиваемый объектно-ориентированный язык (со множественным наследованием), включающий даже элементы функционального программирования. т. е. это живой язык, активно использующийся для программирования как «обычных» пользовательских приложений, с переносимым оконным интерфейсом (см. например *wxPython*) или веб-приложений (см. например, *Zope*, *Django*, *TurboGears*), так и в научной сфере для прототипирования и разработки новых алгоритмов, для анализа или визуализации данных (см., например, *Scientific Python*, <http://www.scipy.org>).

Здесь мы рассмотрим только базовые свойства синтаксиса, чтобы можно было устраниТЬ непонимание текстов, отдельных, не больше страничек, функций и процедур на Python. Желающим узнать больше рекомендуем книгу [[Суз06](#)] или интернет-курс дистанционного обучения [[Суз07](#)], а дальше — добро по-

жаловать на сайт <http://www.python.org>.

Теперь перечислим основные операторы.

Оператор **if**:

```
if x < 0:  
    x = 0  
elif x == 0:  
    print 'Это число - нуль'  
elif x == 1:  
    print 'Это число - 1'  
else:  
    print Это ' число больше единицы '
```

Оператор **for** имеет вид

```
for некаяпеременная_ in некийдиапазон_:
```

Блок кода после заголовка выполняется до тех пор, пока *некая_переменная* принадлежит *некоему_диапазону* (причем этот диапазон может быть списком, числовой последовательностью, массивом каких-либо значений), а если нужно просто организовать цикл «от *a* до *b*», то для этого можно воспользоваться функцией **range(a, b+1)**, которая вернет соответствующий список:

```
for i in range(0, len(EP)):  
    if G.degree(EP[i]) > 0:  
        v = EP[i]  
        break
```

Отметим, что для выхода из цикла можно использовать оператор **break**.

Есть и цикл **while**, который выполняется пока истинно указанное выражение:

```
while N > 0:  
    if (N % 2) == 0:  
        X = X * X % m  
        N = N / 2  
    else:  
        y = y * X % m  
        N = N - 1  
    print N, X, y
```

Декларация любой функции или процедуры (заметим, что декларации могут быть вложенными):

```
def gcd(a, b):  
    print a, b  
    if a == 0:  
        return b  
    return gcd(b % a, a)
```

Для возврата значений (тип которых, кстати, может быть тоже любым) используется **return**. Оператор **print** печатает содержимое переданных ему аргументов, пытаясь придать «читаемый» вид кортежам, спискам и словарям.

В заключение заметим, что мы приводим алгоритмы не на «чистом» Python'e, а в автоматически «облагороженном» виде, где форматированием выделены операторы и идентификаторы, логические операторы представлены математическими символами (« \neg », « \wedge », « \vee »), оператор присваивания показан в виде « \leftarrow » и т. п.

Также мы приводим в тексте только ключевую часть алгоритма, различные вспомогательные функции (инициализация данных, печать и вывод иллюстраций и т. п.) остаются за скобками. Все это сделано,

чтобы максимально избавить текст алгоритма от технических подробностей, но все же это настоящие, работающие алгоритмы, и трассировка их вывода получена при выполнении программы.

7.2 Глоссарий

По результатам чтения лекций, были выявлены основные обозначения, вызывающие у студентов затруднения. Они приведены в виде справочного материала.

Определение 7.2.1. *Алфавит* — конечный набор символов.

Определение 7.2.2. *Слово* — конечная последовательность символов из некоторого алфавита Σ .

Пустое слово обозначается \emptyset . Набор слов длины n над алфавитом Σ обозначается Σ^n , набор всех слов (включая пустое) — Σ^* .

Определение 7.2.3. *Язык* L — произвольное подмножество $L \subseteq \Sigma^*$, т. е. произвольное множество слов над алфавитом Σ .

Надо отличать пустой язык (язык не содержащий ни одного слова), который также обозначается \emptyset , от языка $\{\emptyset\}$, содержащего единственное пустое слово.

Определение 7.2.4. *Дополнение языка* L — язык \bar{L} , состоящий из всех возможных слов над алфавитом Σ , не входящих в язык L — $\bar{L} \equiv \Sigma^* \setminus L$.

Множество вещественных чисел обозначается R , целых — Z , рациональных — Q , натуральных — N . Неотрицательные их подмножества обозначаются соответственно R_+, Z_+, Q_+ .

Пусть $f : N \rightarrow R$, $g : N \rightarrow R$.

Определение 7.2.5. $f=O(g)$, если $\exists c \in R, c > 0$, и

$$\exists n_0 \in N, \forall n > n_0 : |f(n)| \leq c|g(n)|.$$

Определение 7.2.6. $f=o(g)$, если $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.

Определение 7.2.7. $f = \Omega(g)$, если $g = O(f)$.

Определение 7.2.8. $f = \Theta(g)$, если $f = O(g)$ и $g = O(f)$.

Определение 7.2.9. *Кортеж* (k -кортеж) — упорядоченный набор из k элементов (множеств, подмножеств, элементов множеств). Обозначается угловыми скобками, например, кортеж из элементов a , b , и множества C — $\langle a, b, C \rangle$.

Определение 7.2.10. $\lfloor x \rfloor$ — наибольшее целое, не превосходящее x .

Определение 7.2.11. $\lceil x \rceil$ — наименьшее целое, не меньшее x .

\mathbb{Z} — множество целых чисел.

\mathbb{Q} — множество рациональных чисел.

Предметный указатель

- 2-Satisfiability
 - Задача, 266
- 2-Выполнимость
 - Задача, 266
- 2SAT, 266
 - Задача, 266
- 3-Satisfiability
 - Задача, 265–267, 304, 307
- 3-Выполнимость
 - Задача, 265–267, 304, 307
- 3SAT, 265–267, 304, 307
 - Задача, 265–267, 304, 307
- ЗВыполнимость-Максимизация
 - Задача, 304, 305, 309
- APX, 13, 44, 309, 311–314, 318, 319
- APX PB, 318
- BPP, 250, 271, 282–285, 287, 288, 294
- coDTIME, 257
- coNP, 257, 258, 261, 269–271, 278, 291
- coNSPACE, 257
- coNTIME, 257
- Cook
 - Сводимость, 261
- coRP, 271, 275–280, 292, 299, 334
- coZPP, 334
- DNF counting, 167, 168
 - Задача, 167, 168
- DSPACE, 248, 251, 257
- DTIME, 75, 248, 250, 257
 - coDTIME, 257
- Euler cycle

- Алгоритм, 94
- EXPTIME, 250–252
- FPRAS, 163, 166, 168, 169
- FPTAS, 110
- GCD
 - Алгоритм, 21
- halting problem, 245, 253
 - Задача, 245, 253
- K-covering, 76
 - Задача, 76
- Knapsack, 89, 90, 104, 105, 107, 109, 111, 145
 - Алгоритм, 105, 107, 109
 - Жадный, 89, 92, 115
 - Алгоритм Немхаузера – Ульмана, 107, 109, 145, 155, 156
 - Задача, 89, 90, 104, 105, 107, 109, 111, 145
- Knapsack-SAT
 - Алгоритм, 104
- Literal, 163
- Machine
 - Turing, 273
- MAX SNP, 315, 318, 319
- MAX-3SAT, 304, 305, 309
 - Задача, 304, 305, 309
- MAX-CUT, 194, 195, 197, 200
 - Алгоритм
 - Вероятностный, 197
 - Задача, 194, 195, 197, 200
 - MAX-CUT(VP), 196, 200
 - Задача, 196, 200
 - MAX-CUT(ЦП), 195, 196, 200
 - Задача, 195, 196, 200
- MAX-SAT, 186, 188, 204, 205, 207, 304
 - Алгоритм
 - Вероятностный, 190, 196, 207
 - Дерандомизация, 208
 - Задача, 186, 188, 204, 205, 207, 304
- Maximal
 - Independent Set, 210
- Maximal independent set, 210
- MAXSNP, 317
- Mergesort
 - Алгоритм, 45, 47
- Minimum Spanning Tree, 36, 38, 65, 72, 251
 - Алгоритм

- Прима, 36, 38, 65, 72
Задача, 36, 251
- NEXP, 298
- NP, 6, 7, 10–14, 34, 40, 41, 43, 44, 63, 70, 72, 73, 75, 95, 100, 104, 118, 119, 123, 131, 139, 173, 185, 250, 252, 253, 257–266, 269–271, 277, 278, 290, 291, 293, 295–297, 299–302, 305–307, 309–311, 314, 318, 334, 336
- coNP, 257, 258, 261, 269–271, 278, 291
- NPC, 262, 266, 267, 269, 270, 302, 305, 307, 309
- NPO, 311, 312, 318
- NSPACE, 257
- coNSPACE, 257
- NTIME, 257, 298
- coNTIME, 257
- P, 10–13, 43, 131, 250–254, 258, 261–263, 266, 269, 270, 293, 301, 314, 318
- Packing, 122, 123, 128
Алгоритм, 125, 126, 128, 129
Задача, 122, 123, 128
- PCP, 7, 12, 75, 250, 293, 295–302, 305, 306, 308, 309, 314, 336
Система, 300
- PP, 271, 288–291, 294
- PSPACE, 250–252, 258, 290
- PTAS, 312–314, 317, 318, 320
- Quicksort
Алгоритм, 47, 49, 50, 274
- Randomized
Turing Machine, 273, 276, 289, 299
- RP, 250, 271, 275–281, 288, 292, 294
- coRP, 271, 275–280, 292, 299, 334
- Satisfiability, 131, 263, 265, 266
Задача, 131, 263, 265, 266
- Semidefinite programming, 193
- Set
Maximal Independent, 210
- Set Cover, 73, 76
Задача, 73, 76
- Shortest Path
Задача, 27, 61, 251
- Shortest Path Problem, 27, 61, 251
- Traveling Salesman Problem, 24, 34–36, 39, 43, 61, 93, 94, 98, 254, 255, 258, 259, 261
- TSP

- Алгоритм
 - Переборный, [25](#), [26](#), [66](#)
 - Задача, [24](#), [34–36](#), [39](#), [43](#), [61](#), [93](#), [94](#), [98](#), [254](#), [255](#), [258](#), [259](#), [261](#)
 - Метрическая
 - Алгоритм, [95](#), [97](#), [99](#)
- Turing
 - machine, [273](#)
 - Randomized Machine, [273](#), [276](#), [289](#), [299](#)
- Vector programming, [193](#), [195](#)
- Vertex Cover, [87](#)
- Vertex Covering, [78](#), [81](#), [85](#), [86](#), [267](#)
 - Задача, [78](#), [81](#), [85](#), [86](#), [267](#)
- ZPP, [250](#), [291–294](#)
- coZPP, [334](#)
- Алгоритм
 - Euler cycle, [94](#)
 - GCD, [21](#)
 - Knapsack-SAT, [104](#)
 - MAX-CUT
 - Вероятностный, [197](#)
 - MAX-SAT
 - Вероятностный, [190](#), [196](#), [207](#)
- Дерандомизация, [208](#)
- Mergesort, [45](#), [47](#)
- Minimum Spanning Tree
 - Прима, [36](#), [38](#), [65](#), [72](#)
- Quicksort, [47](#), [49](#), [50](#), [274](#)
- TSP
 - Метрическая, [95](#), [97](#), [99](#)
 - Переборный, [25](#), [26](#), [66](#)
- Дейкстры, [27](#), [29–31](#), [36](#), [65](#), [250](#)
- Динамическое программирование
 - Knapsack, [105](#), [107](#), [109](#)
 - Packing, [125](#), [126](#), [128](#), [129](#)
- Жадный
 - Knapsack, [89](#), [92](#), [115](#)
 - Рюкзак, [89](#), [92](#), [115](#)
- Коммивояжер
 - Метрический, [95](#), [97](#), [99](#)
 - Переборный, [25](#), [26](#), [66](#)
- Минимальное оствовное дерево, [36](#), [38](#), [65](#), [72](#)
- НОД, [21](#)
- Немхаузера – Ульмана, [107](#), [109](#), [145](#), [155](#), [156](#)
- Переполнение памяти умножением, [65](#), [66](#), [253](#), [254](#)
- Полиномиальный в среднем, [121](#), [126](#)

Приближенный, 110, 302, 311
Рюкзак, 105, 107, 109
Сортировка
 Быстрая, 47, 49, 50, 274
 Слиянием, 45, 47
Сумма размеров, 104
Упаковка, 125, 126, 128, 129
Флойда – Уоршолла, 33
Эйлеров цикл, 94
Векторное программирование, 193, 195
Вершинное покрытие, 87
 Задача, 78, 81, 85, 86, 267
Выполнимость
 Задача, 131, 263, 265, 266
Выполнимость-Максимизация
 Задача, 186, 204, 205, 304
Выполнимость-Максимизация
 Задача, 188, 207
ДНФ
 посчет выполняющих наборов, 167, 168
Дейкстры
 Алгоритм, 27, 29–31, 36, 65, 250
Доминирующее подмножество
 Рюкзак, 145, 147

Задача
 2-Satisfiability, 266
 2-Выполнимость, 266
 2SAT, 266
 3-Satisfiability, 265–267, 304, 307
 3-Выполнимость, 265–267, 304, 307
 3SAT, 265–267, 304, 307
 3Выполнимость-Максимизация, 304, 305, 309
 DNF counting, 167, 168
 halting problem, 245, 253
 K-covering, 76
 Knapsack, 89, 90, 104, 105, 107, 109, 111, 145
 MAX-3SAT, 304, 305, 309
 MAX-CUT, 194, 195, 197, 200
 MAX-CUT(VP), 196, 200
 MAX-CUT(ЦП), 195, 196, 200
 MAX-SAT, 186, 188, 204, 205, 207, 304
 Minimum Spanning Tree, 36, 251
 Packing, 122, 123, 128
 Satisfiability, 131, 263, 265, 266
 Set Cover, 73, 76
 Shortest Path, 27, 61, 251
 TSP, 24, 34–36, 39, 43, 61, 93, 94, 98, 254, 255,
 258, 259, 261

- Vertex Covering, 78, 81, 85, 86, 267
 Вершинное покрытие, 78, 81, 85, 86, 267
 Выполнимость, 131, 263, 265, 266
 Выполнимость-Максимизация, 186, 204, 205, 304
 Выполнимость-Максимизация, 188, 207
 К-покрытие, 76
 Коммивояжер, 24, 34–36, 39, 43, 61, 93, 94, 98, 254, 255, 258, 259, 261
 Коммивояжера метрическая, 98
 Кратчайшие пути, 27, 61, 251
 Максимальный разрез, 194, 195, 197, 200
 Максимальный разрез(VP), 196, 200
 Максимальный разрез(ЦП), 195, 196, 200
 Минимальное оствовное дерево, 36, 251
 Остановки, 245, 253
 Покрытие множества, 73, 76
 Рюкзак булев, 89, 90, 104, 105, 107, 109, 111, 145
 Сумма размеров, 104
 Упаковка, 122, 123, 128
 Инцидентность, 123
 К-покрытие
 Задача, 76
- Карп
 Сводимость, 262, 305
 Класс
 APX, 13, 44, 309, 311–314, 318, 319
 APX PB, 318
 BPP, 250, 271, 282–285, 287, 288, 294
 DSPACE, 248, 251, 257
 DTIME, 75, 248, 250, 257
 coDTIME, 257
 EXPTIME, 250–252
 FPRAS, 163, 166, 168, 169
 FPTAS, 110
 MAX SNP, 315, 318, 319
 MAXSNP, 317
 NEXP, 298
 NP, 6, 7, 10–14, 34, 40, 41, 43, 44, 63, 70, 72, 73, 75, 95, 100, 104, 118, 119, 123, 131, 139, 173, 185, 250, 252, 253, 257–266, 269–271, 277, 278, 290, 291, 293, 295–297, 299–302, 305–307, 309–311, 314, 318, 334, 336
 coNP, 257, 258, 261, 269–271, 278, 291
 NPC, 262, 266, 267, 269, 270, 302, 305, 307, 309
 NPO, 311, 312, 318
 NSPACE, 257

- CONSPACE, 257
 - NTIME, 257, 298
 - CONTIME, 257
 - P, 10–13, 43, 131, 250–254, 258, 261–263, 266, 269, 270, 293, 301, 314, 318
 - PCP, 7, 12, 75, 250, 293, 295–302, 305, 306, 308, 309, 314, 336
 - PP, 271, 288–291, 294
 - PSPACE, 250–252, 258, 290
 - PTAS, 312–314, 317, 318, 320
 - RP, 250, 271, 275–281, 288, 292, 294
 - CORP, 271, 275–280, 292, 299, 334
 - ZPP, 250, 291–294
 - coZPP, 334
 - Коммивояжер
 - Алгоритм
 - Переборный, 25, 26, 66
 - Задача, 24, 34–36, 39, 43, 61, 93, 94, 98, 254, 255, 258, 259, 261
 - Метрический
 - Алгоритм, 95, 97, 99
 - Кратчайшие пути
 - Алгоритм Дейкстры, 27, 29–31, 36, 65, 250
 - Алгоритм Флойда – Уоршолла, 33
- Задача, 27, 61, 251
 - Кук
 - Сводимость, 261
 - Литерал, 163
 - Максимальное
 - Независимое множество, 210
 - Максимальный разрез
 - Задача, 194, 195, 197, 200
 - Максимальный разрез(VP)
 - Задача, 196, 200
 - Максимальный разрез(ЦП)
 - Задача, 195, 196, 200
 - Матрица
 - Инцидентности, 123
 - Машина
 - Тьюринга, 273
 - Тьюринга Вероятностная, 273, 276, 289, 299
 - Метрическая
 - Задача коммивояжера, 98
 - Минимальное оствовное дерево
 - Алгоритм, 36, 38, 65, 72
 - Задача, 36, 251
 - Множество
 - Максимальное независимое, 210

НОД

Алгоритм, 21

Независимое множество, 210

Остановки

Задача, 245, 253

Парето-набор

Рюкзак, 145, 147

Покрытие множества

Задача, 73, 76

Полиномиальный в среднем, 121, 126

Полуопределенное программирование, 193

Приближенный

Алгоритм, 110, 302, 311

Рюкзак

Алгоритм

Динамическое программирование, 105, 107, 109

Жадный, 89, 92, 115

Алгоритм Немхаузера – Ульмана, 107, 109, 145, 155, 156

Доминирующее подмножество, 145, 147

Парето-набор, 145, 147

Рюкзак булев

Задача, 89, 90, 104, 105, 107, 109, 111, 145

Сводимость

Карп, 262, 305

Кука, 261

Система

PCP, 300

Сортировка

Быстрая, 47, 49, 50, 274

Слиянием, 45, 47

Сумма размеров, 104

Алгоритм, 104

Задача, 104

Тьюринга

Машина, 273

Машина Вероятностная, 273, 276, 289, 299

Упаковка

Алгоритм

Динамическое программирование, 125, 126, 128, 129

Задача, 122, 123, 128

Флойда – Уоршолла

Алгоритм, 33

Эйлеров цикл

Алгоритм, 94

Список иллюстраций

1.1	Работа алгоритма 6 «TSP-перебор»	25
1.2	Работа алгоритма 7 «Дейкстры»	29
1.3	Иллюстрация работы алгоритма 8 «Флойда – Уоршолла»	33
1.4	Иллюстрация работы алгоритма 9 «MST Прима»	38
1.5	Работа «Quicksort» с разными методами выбора оси	51
1.6	Карта-памятка раздела 1.1	53
1.7	RAM — машина с произвольным доступом	56
1.8	Моделирование циклов для RAM	58
2.1	«Плохой» граф для жадного алгоритма вершинного покрытия	80
2.2	Неоптимальность жадного алгоритма для вершинного покрытия	82
2.3	«Ленивый» и «жадный» алгоритмы вершинного покрытия на «плохих» графах	83
2.4	«Ленивый» и «жадный» алгоритмы вершинного покрытия на случайных графах	84
2.5	Карта-памятка разделов 2.1.2 и 2.1.1	87
2.6	Наборы S_g и \tilde{S}_g в «жадном» алгоритме для «рюкзака»	93

2.7	Карта-памятка раздела 2.1.4	101
2.8	Карта-памятка раздела 2.2.2	116
3.1	Работа алгоритма 28 «Упаковка-ДинПрог»	125
3.2	Алгоритм 28 «Упаковка-ДинПрог» на случайных данных	129
3.3	Карта-памятка раздела 3.2.0	130
3.4	«Обнуляющие» наборы для трехскобочной КНФ	133
3.5	Карта-памятка раздела 3.3.0	139
3.6	Граф зависимостей утверждений в разделе 3.5	153
3.7	Алгоритм 25 «Рюкзак Немхаузера–Ульмана» на случайных данных	156
4.1	Множества U, G, H в задаче 18 «DNF#»	167
4.2	Карта-памятка раздела 4.2.0	170
4.3	<i>PRAM — Parallel Random Access Machine</i>	171
4.4	График функции $1 - (1 - x/k)^k$ при различных k	189
4.5	График функции $\frac{2\theta}{\pi(1-\cos\theta)}$	198
4.6	Вектора в вероятностном округлении «MAX-CUT»	199
4.7	Карта-памятка раздела 4.4.2	202
5.1	Дерандомизация на основе минимизации оценок математического ожидания	206
5.2	Карта-памятка раздела 5.1.0	209
6.1	Машина Тьюринга: удвоение строки	233
6.2	Машина Тьюринга: унарное сложение	234
6.3	Машина Тьюринга: распознавание четных строк	235
6.4	Пример МТ: «Количество 0 и 1 равно?»	236

6.5	Выполнение МТ «Количество 0 и 1 равно?»	237
6.6	Трехленточная универсальная МТ для одноленточных МТ	241
6.7	Карта-памятка раздела 6.1.1	247
6.8	Отображение выполнимой КНФ на граф	268
6.9	Классы \mathcal{NP} , $\text{co}\mathcal{NP}$, \mathcal{P} , \mathcal{NPC}	270
6.10	Карта-памятка раздела 6.2.3	272
6.11	Классы сложности \mathcal{RP} и $\text{co}\mathcal{RP}$	279
6.12	Классы сложности: \mathcal{BPP} и его «соседи»	287
6.13	Вероятностные классы сложности: \mathcal{ZPP} , \mathcal{RP} , \mathcal{BPP} , \mathcal{PP}	294
6.14	Карта-памятка раздела 6.3	295
6.15	Карта-памятка раздела 6.3.5	303
6.16	Карта-памятка раздела 6.3.6	310
6.17	Пример схемы: сравнение двух строк	322
6.18	Пример коммуникационного протокола для « $x_1x_2 \stackrel{?}{=} y_1y_2$ »	330
6.19	Иерархия некоторых классов сложности	335

Список алгоритмов

1	Тривиальное вычисление $y = x^n \bmod m$	16
2	Разумное вычисление $y = x^n \bmod m$	17
3	Вычисление факториала $y = n! \bmod m$	19
4	Алгоритм Евклида	20
5	Неэффективный алгоритм для «НОД»	21
6	Переборный алгоритм для «TSP»	23
7	Алгоритм Дейкстры	28
8	Алгоритм Флойда-Уоршолла	32
9	Алгоритм Прима	37
10	Сортировка слиянием	46
11	Процедура «partition» в «Quicksort»	48
12	Быстрая сортировка/«Quicksort»	49
13	Переполнение памяти умножением	60
14	Тривиальное вычисление $a \cdot b$ на RAM	60
15	Жадный алгоритм для задачи о покрытии	74

16	Жадный алгоритм для вершинного покрытия	79
17	«Ленивый» алгоритм для задачи 12 «Min Vertex Covering»	85
18	Жадный алгоритм для задачи 13 «Knapsack»	90
19	«Модифицированный жадный» для «Рюкзака»	91
20	Алгоритм нахождения Эйлерова цикла	95
21	Простой алгоритм для решения метрической задачи коммивояжера	96
22	Алгоритм Кристофицеса для метрической TSP	98
23	Сумма размеров	103
24	«Рюкзак»: динамическое программирование	106
25	«Рюкзак» Немхаузера – Ульмана	108
26	«Рюкзак» с отбором «легких» решений	112
27	PTAS для рюкзака	117
28	«Упаковка»: динамическое программирование	124
29	Динамическое программирование для «SAT»	136
30	Алгоритм MIS-Greedy	174
31	Алгоритм «Parallel MIS»	176
32	Протокол византийского соглашения	182
33	Приближенный вероятностный алгоритм для задачи 20 (MAX-SAT) на основе линейной релаксации	188
34	«SDP-округление MAX-CUT»	200
35	Дерандомизация вероятностного алгоритма 33 «вероятностный MAX-SAT» по методу условных вероятностей	207
36	Полиномиальный алгоритм проверки простоты	218
37	Симулятор работы машины Тьюринга	232

Bibliography

- [07] *List of NP-complete problems*. English. 2007. URL: http://en.wikipedia.org/wiki/List_of_NP-complete_problems.
- [Aar07] Scott Aaronson. *Complexity Zoo, Wiki-review of complexity classes*. English. 2007. URL: http://qwiki.stanford.edu/wiki/Complexity_Zoo.
- [AKS02] M. Agrawal, N. Kayal, and N. Saxena. *PRIMES is in P*. English. 2002. URL: <http://citeseer.ist.psu.edu/article/agrawal02primes.html>.
- [AKS04] M. Agrawal, N. Kayal, and N. Saxena. “PRIMES is in P”. English. In: *Annals of Mathematics* 160 (2004), pp. 781–793.
- [AS92] N. Alon and J.H. Spencer. *The Probabilistic Method*. English. Wiley, 1992.
- [Blu67] Manuel Blum. “A Machine-Independent Theory of the Complexity of Recursive Functions”. English. In: *Journal of the ACM* 14.2 (1967), pp. 322–336.

- [BV03] R. Beier and B. Vöcking. “Random Knapsack in Expected Polynomial Time”. English. In: *Proceedings of the 35th ACM Symposium on Theory of Computing (STOC)*. 2003, pp. 232–241. URL: <http://citeseer.ist.psu.edu/beier03random.html>.
- [Chu36] Alonzo Church. “An Unsolvable Problem of Elementary Number Theory”. English. In: *American Journal of Mathematics* 58.2 (1936), pp. 345–363.
- [CK] P. Crescenzi and V. Kann. *A compendium of NP optimization problems*. English. URL: <http://www.nada.kth.se/viggo/problemList/compendium.html>.
- [CW90] Don Coppersmith and Shmuel Winograd. “Matrix multiplication via arithmetic progressions”. In: *J. Symb. Comput.* 9.3 (1990), pp. 251–280. ISSN: 0747-7171. DOI: [http://dx.doi.org/10.1016/S0747-7171\(08\)80013-2](http://dx.doi.org/10.1016/S0747-7171(08)80013-2).
- [Fei98] Uriel Feige. “A Threshold of $\ln n$ for Approximating Set Cover”. English. In: *Journal of the ACM* 45.4 (1998), pp. 634–652. URL: <http://citeseer.ist.psu.edu/feige95threshold.html>.
- [Fre77] Rusins Freivalds. “Probabilistic Machines Can Use Less Running Time”. English. In: *IFIP Congress*. 1977, pp. 839–842.
- [Gol99] Oded Goldreich. *Complexity Theory: lecture notes*. English. 1999. URL: <http://www.wisdom.weizmann.ac.il/~oded/cc.html>.
- [Gur+98] Venkatesan Guruswami et al. “A Tight Characterization of NP with 3 Query PCPs”. English. In: *IEEE Symposium on Foundations of Computer Science*. 1998, pp. 8–17. URL: <http://citeseer.ist.psu.edu/guruswami98tight.html>.
- [GW94] M. Goemans and D. Williamson. “New 3/4-approximation algorithms for MAX SAT”. English. In: *SIAM Journal on Algebraic and Discrete Methods*. Vol. 7. 1994, pp. 656–666. URL: <http://citeseer.iast.psu.edu/goemans94new.html>.

- [GW95] Michel X. Goemans and David P. Williamson. "Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming". English. In: *J. Assoc. Comput. Mach.* 42 (1995), pp. 1115–1145. URL: <http://citesear.ist.psu.edu/goemans95improved.html>.
- [Hås97] Johan Håstad. "Some optimal inapproximability results". English. In: *STOC'97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. El Paso, Texas, United States: ACM Press, 1997, pp. 1–10. ISBN: 0-89791-888-6. DOI: <http://doi.acm.org/10.1145/258533.258536>.
- [Hoa62] C. A. R. Hoare. "Quicksort". English. In: *Computer Journal* 5.1 (1962), pp. 10–15.
- [IM02] Kazuo Iwama and Hiroki Morizumi. "An Explicit Lower Bound of $5n - o(n)$ for Boolean Circuits". English. In: *Proc. of the 27th Int. Symp. on Math. Foundations of Computer Science* 2420 (2002), pp. 353–364. ISSN: 0302-9743. URL: <http://link.springer.de/link/service/series/058/papers/2420/24200353.pdf>.
- [Iwa89] Kazuo Iwama. "CNF satisfiability test by counting and polynomial average time". English. In: *SIAM J. Comput.* 18.2 (1989), pp. 385–391. ISSN: 0097-5397. DOI: <http://dx.doi.org/10.1137/0218026>.
- [Joh73] David S. Johnson. "Approximation algorithms for combinatorial problems". English. In: *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*. Austin, Texas, United States: ACM Press, 1973, pp. 38–49. DOI: <http://doi.acm.org/10.1145/800125.804034>.
- [Joh90] David S. Johnson. "A Catalog of Complexity Classes". English. In: *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*. MIT Press, 1990, pp. 67–161.
- [Kar84] N. Karmarkar. "A New Polynomial-Time Algorithm for Linear Programming". English. In: *Combinatorica* 4.4 (1984), pp. 373–395.

- [Kha79] L. G. Khachiyan. “A Polynomial Algorithm in Linear Programming”. English. In: *Soviet Mathematics Doklady* 20 (1979), pp. 191–194.
- [Kha+99] Sanjeev Khanna et al. “On Syntactic versus Computational Views of Approximability”. English. In: *SIAM J. Comput.* 28.1 (1999), pp. 164–191. ISSN: 0097-5397. DOI: <http://dx.doi.org/10.1137/S0097539795286612>.
- [KLM89] R. M. Karp, M. Luby, and N. Madras. “Monte-Carlo approximation algorithms for enumeration problems”. English. In: *J. Algorithms* 10.3 (1989), pp. 429–448. ISSN: 0196-6774. DOI: [http://dx.doi.org/10.1016/0196-6774\(89\)90038-2](http://dx.doi.org/10.1016/0196-6774(89)90038-2).
- [KZ97] Howard Karloff and Uri Zwick. “A 7/8-approximation algorithm for MAX 3SAT?” English. In: *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, Miami Beach, FL, USA*. IEEE Press, 1997. URL: <http://citeseer.ist.psu.edu/karloff97approximation.html>.
- [Lov99] László Lovász. *Complexity of algorithms*. English. 1999. URL: <http://research.microsoft.com/users/lovasz/notes.htm>.
- [Lub85] M. Luby. “A simple parallel algorithm for the maximal independent set problem”. English. In: *STOC '85: Proceedings of the seventeenth annual ACM symposium on Theory of computing*. Providence, Rhode Island, United States: ACM Press, 1985, pp. 1–10. ISBN: 0-89791-151-2. DOI: <http://doi.acm.org/10.1145/22145.22146>.
- [MR95] R. Motwani and P. Raghavan. *Randomized algorithms*. English. Cambridge Univ. Press, 1995.
- [Pos36] Emil L. Post. “Finite Combinatory Processes—Formulation 1”. English. In: *J. Symb. Log.* 1.3 (1936), pp. 103–105.
- [Pri57] R. C. Prim. “Shortest Connection Networks and Some Generalizations”. In: *Bell System Technical Journal* 36 (1957), pp. 1389–1401.

- [PY91] C. H. Papadimitriou and M. Yannakakis. “Optimization, approximation, and complexity classes”. English. In: *Journal of Computing and System Sciences* 43 (1991), pp. 425–440.
- [Rab83] Michael O. Rabin. “Randomized Byzantine Generals”. English. In: *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*. 1983, pp. 403–409.
- [Rag88] Prabhakar Raghavan. “Probabilistic constructions of deterministic algorithms: approximating packing integer programs”. English. In: *Journal of Computer and System Scince* 37.4 (1988), pp. 130–143.
- [Sla96] Petr Slavík. “A Tight Analysis of the Greedy Algorithm for Set Cover”. English. In: *ACM Symposium on Theory of Computing*. 1996, pp. 435–441. URL: <http://citeseer.ist.psu.edu/slavik95tight.html>.
- [Sma00] Steven Smale. “Mathematical Problems for the Next Century”. English. In: *Mathematics: Frontiers and Perspectives, IMU, AMS, 2000*. Ed. by V. Arnold et al. American Mathematical Society, Providence, 2000. URL: <http://citeseer.ist.psu.edu/smale98mathematical.html>.
- [Tur36] Alan M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. English. In: *Proceedings of the London Mathematical Society* 2.42 (1936), pp. 230–265. URL: <http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf>.
- [АХУ79] Альфред Ахо, Джон Хопкрофт, and Джейфри Ульман. *Построение и анализ вычислительных алгоритмов*. М.: Мир, 1979.
- [ГД82] М. Гэри and Д. С. Джонсон. *Вычислительные машины и труднорешаемые задачи*. russian. М.: Мир, 1982.
- [Кар75] Р. М. Карп. “Сводимость комбинаторных задач”. russian. In: *Киб. сборник, нов. сер.* Ed. by ДАН СССР. 12. 1975, pp. 16–38.

- [КЛР99] Томас Кормен, Чарльз Лейзерсон, and Рональд Ривест. *Алгоритмы: построение и анализ.* russian. М.: МЦНМО, 1999.
- [КР96] Н. Н. Кузюрин and А. А. Разборов. *Оценка состояния и прогнозные исследования эффективных алгоритмов для точного и приближенного решения переборных задач дискретной оптимизации.* russian. Tech. rep. Математический институт им. В.А. Стеклова РАН, Москва, 1996.
- [Кук75] С. А. Кук. “Сложность процедур вывода теорем”. russian. In: *Киб. сборник, нов. сер.* Ed. by M. Мир. 12. 1975, pp. 5–15.
- [КШВ99] А. Китаев, А. Шень, and М. Вялый. *Классические и квантовые вычисления.* russian. Издательство МЦНМО, 1999. URL: <http://www.mccme.ru/free-books/>.
- [ЛН88] Р. Лидл and Г. Нидеррайтер. *Конечные поля.* russian. М. Мир, 1988.
- [Раз85а] А. А. Разборов. “Нижние оценки монотонной сложности логического перманента”. russian. In: *Математические Заметки* 37.4 (1985), pp. 887–900.
- [Раз85б] А. А. Разборов. “Нижние оценки размера схем ограниченной глубины в полном базисе, содержащем функцию логического сложения.” russian. In: *Математические Заметки* 37.6 (1985).
- [Суз06] Р.А. Сузи. *Язык программирования Python.* russian. Интернет-университет информационных технологий — ИНТУИТ.ру, 2006.
- [Суз07] Р.А. Сузи. *Курс ИНТУИТ: Язык программирования Python.* russian. 2007. URL: <http://www.intuit.ru/department/pl/python/>.
- [Схр91] А. Схрейвер. *Теория линейного и целочисленного программирования.* russian. М. Мир, 1991.
- [ХС67] Дж. Хартманис and Р. Стирнз. “О вычислительной сложности алгоритмов”. russian. In: *Киб. сборник, нов. сер.* 4. М., Мир, 1967, pp. 57–85.