Юрай Громкович

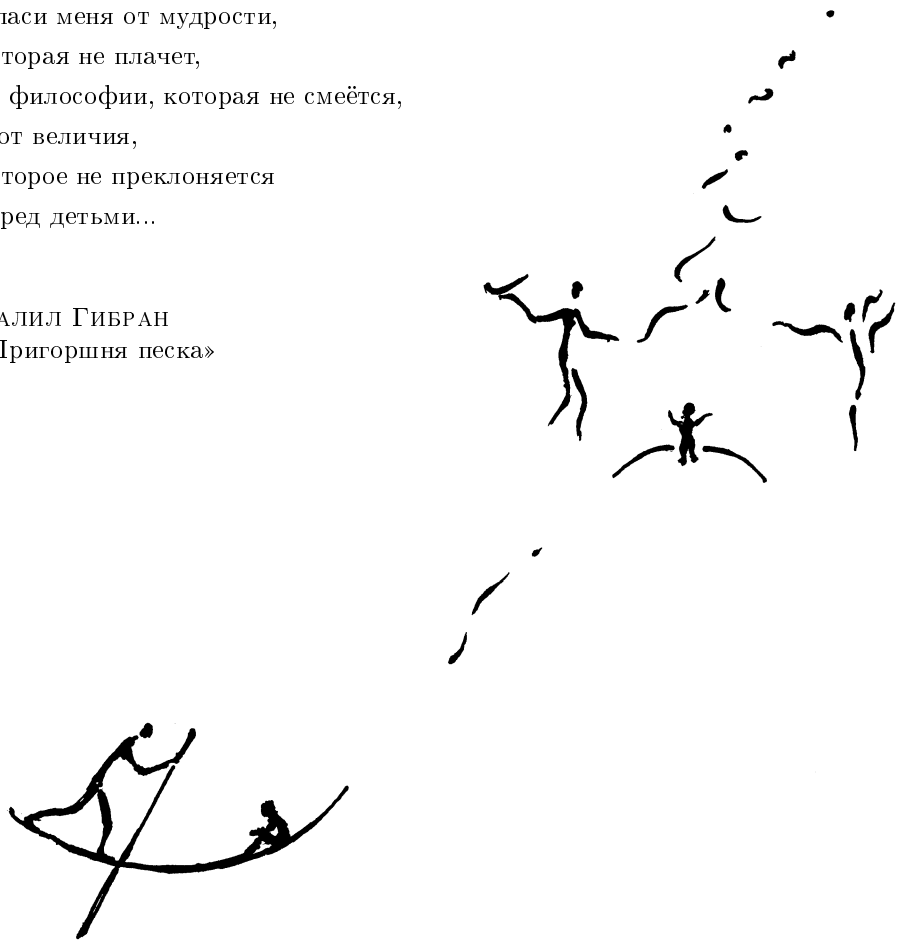
Теоретическая информатика

Введение в теорию автоматов,
теорию вычислимости, теорию сложности,
теорию алгоритмов, рандомизацию,
теорию связи и криптографию

Моим родителям

Спаси меня от мудрости,
которая не плачет,
от философии, которая не смеётся,
и от величия,
которое не преклоняется
перед детьми...

Халил Гибран
«Пригоршня песка»



Предисловие

Этот учебник является введением в теоретическую информатику. При этом основное внимание в нём уделено разработке алгоритмических концепций. Учебник является существенной переработкой предыдущего – написанного на немецком языке «Algorithmische Konzepte der Informatik»; он был создан на основе курса лекций, читавшихся автором в Университете Аахена по теоретическим основам информатики. Материал для курса лекций и соответствующих глав учебника был выбран таким образом, чтобы по-возможности соблюсти баланс между классическими основами информатики (сюда относятся теория автоматов, теория вычислимости и NP-полноты) – и современными её темами (такими как аппроксимационные и рандомизированные алгоритмы[1], а также криптография).

В отличие от технических и прикладных областей информатики теоретическая информатика связана с фундаментальными вопросами существования алгоритмических решений, физических пределов вычислений, методологии разработки алгоритмов и т.д. Все эти вопросы в первую очередь связаны с математикой, не имеют непосредственного отношения к прикладным областям – поэтому студенты часто считают теоретическую информатику слишком трудной, неинтересной, не видят достаточных причин для её глубокого изучения. Главная цель этой книги состоит в том, чтобы изменить это отрицательное мнение относительно теоретической информатики. Это – красивая научная дисциплина; благодаря своим захватывающим результатам и тесным связям с другими науками она во многом изменила наше мировоззрение. С другой стороны, нет сомнения в том, что она весьма полезна для практики. Она даёт методологию – т. е. конкретные понятия и методы, которые могут применяться в течение всего процесса разработки и выполнения программы, а также анализа программных систем. Кроме того, без фундаментальных теоретических разработок были бы невозможны многие постоянно используемые приложения, имеющие огромную экономическую важность, – электронная коммерция и мн. др.

Для привлечения интереса читателя к теоретической информатике мы пытались правильно выбрать темы и подробно сформулировать причины для их изучения. Достаточно трудно добиться глубокого понимания методов, имеющих существенные практические приложения, а также овладеть мастерством применения этих методов на практике – но, несмотря на это, мы попробуем предоставить читателю несложный

[1] В русской литературе применяются и другие термины – «приближённые» вместо «аппроксимационные» и «вероятностные» вместо «рандомизированные». *(Прим. перев.)*

путь их освоения, а также показать, что некоторые вопросы, непосредственно связанные с математической строгостью, могут быть легко доступны даже для новичков. Простота и ясность – вот основные методологические особенности этой книги. Все идеи, понятия, методы анализа и доказательства сначала объясняются неформально – для формирования у читателя верного представления о них. После этого они обычно строго определяются и детально доказываются. Следуя этой стратегии, мы выбрали для рассмотрения те основные понятия теоретической информатики, где можно использовать наиболее ясные и простые примеры, – а не самые лучшие, но технически сложные результаты. Представляя основные понятия теории в порядке их исторического появления, мы стремимся показать развитие способа мышления, свойственного теоретической информатике, – от её истоков до настоящего времени.

Я хотел бы выразить мою глубокую благодарность Дирку Бонгарцу, Хансу-Йоахиму Бёкенхауэру, Александру Феррайну и Ивонне Мо – за тщательное прочтение всей рукописи, многочисленные комментарии и предложения. Особая благодарность – Ивонне Мо за её помощь в течение всего процесса подготовки рукописи на английском языке. Я также обязан Фолькеру Клаусу, Галине Йирасковой, Багдату Эль Абдуни Хаяри, Георгу Шнитгеру, Каролю Тауберу, Эриху Фалкеме и Мануэлю Уолу за различные комментарии и поддержку в процессе работы над книгой.

Я также хотел бы сердечно поблагодарить Ингрид Замечникову за её иллюстрации – единственную безукоризненную часть этой книги. Особой благодарности требует успешное сотрудничество с Альфредом Хофманном и Ингеборгом Майером из издательства «Springer». А последняя (но при этом никак не менее глубокая!) благодарность – Тане за её коллекцию цитат.

От редактора перевода

Читателю предлагается книга известного автора, Юра́я Громковича. На русском языке выходит его первая книга – хотя в издательствах «Springer» и «Teubner» на немецком и английском языках вышли несколько книг этого автора, причём многие из них изданы и на других языках (французском, японском...). Несколько упрощая ситуацию, можно сказать, что по книгам Ю. Громковича «учится пол-мира» – а мы, «как обычно», здесь отстаём от других стран.

Но очень важно, что отстаём мы не только в самом *издании* этих книг. Имеется ещё одно отставание – оно заключается в способе *подачи материала*. На русском языке практически отсутствует учебная и научная литература, в которой приводились бы не только сами опеределения и доказательства утверждений, но и некоторые другие факты и рассуждения, нужные для того, чтобы читателю было легче освоить новую для него теорию.[2] Сам автор для описания этого стиля подачи материала применяет фразы вроде

> «мы работаем на двух уровнях... первый уровень использует только интуитивное понимание, а на втором применяются формальные доказательства...»

– но, по-видимому, нужно говорить не только и не столько об интуиции, о неформальной подаче материала на её основе – сколько о *помощи читателю* в освоении материала книги. Для этой цели автор объясняет свой выбор конкретных вариантов терминов и используемых моделей, указывает ограничения на их использование. А читателю в процессе освоения материала книги более важно понимать не то, *как* именно можно решить рассматриваемую проблему – а то, *почему* её надо решать именно таким способом.[3] В связи с этим в книге, например, нередко приводятся разные варианты доказательства одного и того же утверждения; это вряд ли необходимо для научных статей – но очень полезно студентам, да и вообще читателям-новичкам. Ещё раз отметим, что практически вся учебная литература на русском языке не содержит подобного «дружественного интерфейса», о котором говорилось в данном абзаце.

Есть и другие отличия данной книги от имеющихся учебников по теоретическим основам информатики и дискретной математике. Среди них отметим прежде всего та-

[2] Одно из немногих удачных исключений – журналы «Квант» и литература, издававшаяся в «Библиотечке Кванта». Однако *учебников* на русском языке, написанных в подобном стиле, практически нет.

[3] «You're far to keen and *where* and *how*, and not so hot on *why*» (Andrew Lloyd Webber and Tim Rice, «Jesus Christ Superstar»).

кое: в качестве базы для теоретической информатики используется теория формальных языков.[4] Более того, знания теории формальных языков необходимы при чтении любого раздела книги – однако сама теория формальных языков приводится в книге только в таком объёме, который необходим для чтения следующих глав.

Попробуем ответить и на такой вопрос: чем является эта книга – учебником или научной монографией? По-видимому, скорее – учебником (мы уже называли её так); но её можно считать и научной монографией – например, потому, что существенная часть её материала в монографиях на русском языке ранее не публиковалась.

Название книги – «Теоретическая информатика». Можно даже сказать, что книга включает *всю* математическую теорию, необходимую программистам-практикам.[5] Начинающий программист может не осознавать необходимость изучения всей имеющейся в книге теории – однако, по-видимому, без такого изучения *хорошим* программистом не стать.

Отметим ещё, что именно отсутствие научных монографий, изданных на русском языке и относящихся к тематике нескольких глав данной книги, привело к некоторым проблемам перевода; в первую очередь эти проблемы связаны с неустоявшейся русской терминологией, в частности – с аббревиатурами. В подобных случаях приведены соответствующие сноски (заканчивающиеся пометкой «Прим. перев.») – и мы будем благодарны читателям за размышления о том, какие именно термины, по их мнению, желательно применять в каждом конкретном случае.

Июнь 2006 *Борис Мельников*

[4] Есть ли какой-нибудь другой учебник, изданный на русском языке, в котором применён подобный подход?

[5] Или почти всю. Не очень просто ответить на вопрос, какие именно темы можно было бы добавить к рассматривающимся в этой книге. Возможно, некоторые алгебраические структуры, LR-анализ, рекуррентные последовательности... Но, по-видимому, для первоначального знакомства с теоретической информатикой можно обойтись и без них. А подробно про свой выбор конкретных тем автор пишет во введении.

Оглавление

Если вы хотите
построить корабль –
то не собирайте людей рубить деревья,
не давайте им заданий
и не поручайте никакой работы...
просто заразите их всех своей любовью к морю –
и тогда они сами построят корабль.

А. де Сент-Экзюпери

1

Введение

1.1 Что такое теоретическая информатика?

«Информатика» (или «компьютерные науки») – это наиболее общее название той области науки, основам которой посвящена данная книга. Каждый изучающий эту дисциплину теоретически – или осваивающий её на практике – должен всё время думать о том, как именно следует определять понятие «информатика», об отношении информатики к науке вообще, к образованию – да и к нашей повседневной жизни. И важно понимать, что изучение этой научной дисциплины, всё более и более глубокое «погружение» в неё – всё это приведёт к более глубокому пониманию роли и места этой дисциплины среди других наук. Поэтому необходимо время от времени пересматривать свои взгляды на восприятие информатики как науки – и особенно это важно для студентов. И автор готов к тому, что точка зрения читателя на определение понятия «информатика» может не полностью совпадать с той, которая предлагается в данном введении. «В спорах рождается истина» – поэтому пытайтесь заочно спорить с автором, и ваше понимание информатики от этого только улучшится.

Итак, прежде всего попытаемся ответить на вопрос

что такое информатика?

Поскольку для научной дисциплины сложно дать точное и полное определение, обычно говорят, что

> *информатика – это наука о представлении, хранении и передаче информации, а также её алгоритмической обработке.*

Это определение объявляет главными объектами исследования информацию и алгоритмы её обработки – однако оно не отражает сущность и методологию информатики.

Мы можем поставить и другой вопрос относительно природы информатики как науки:

> *какой именно научной дисциплиной является информатика? некоторой мета-наукой (вроде математики и философии)? частью естествознания? инженерно-технической дисциплиной?*

Ответ на этот вопрос необходим не только для объяснения предмета изучения информатики – он нужен и для определения её методологии и вклада в другие науки. Ответ

таков: информатика не может быть полностью включена ни в одну из названных областей, поскольку она сама включает различные аспекты каждой из них – и математики, и естествознания, и инженерно-технических наук. Почему же это так?

Подобно философии и математике, информатика исследует общие категории – такие, как

> *детерминизм, недетерминизм, случайность, информацию, истину, ложь, сложность, язык, доказательство, знание, связь, приближение, алгоритм, моделирование, и т.д.*

– и вносит свой вклад в понимание природы этих категорий. Более того, информатика придала многим из них новый смысл.

Естествознание, в отличие от философии и математики, изучает конкретные природные объекты и процессы, определяет границу между возможным и невозможным, исследует количественные правила для природных процессов. Оно моделирует, анализирует, а также подтверждает достоверность моделей посредством различных экспериментов. Аналогичные методы используются и в информатике. Объектами здесь являются информация и алгоритмы (программы, компьютеры), а исследуемыми процессами – конкретные вычисления.

Попробуем записать всё это с другой точки зрения – с точки зрения исторического развития компьютерных наук. Исторически первым в информатике был поставлен следующий важный вопрос:

> *существуют ли чётко поставленные задачи, которые не могут быть автоматически решены компьютером – современным или будущим, вне зависимости от вычислительной мощности последнего?*

Вопрос с философскими корнями! Вероятно, именно попытки ответить на него и привели к созданию информатики как независимой научной дисциплины. Ответ на этот вопрос положителен: теперь мы можем описать большое количество задач, которые хотелось бы решать алгоритмически – но мы можем доказать, что это невозможно. Доказательство этого факта для каждой из задач основано на теории алгоритмической неразрешимости (т. е. на *доказательстве отсутствия* алгоритма решения задачи) – а вовсе не на том факте, что к настоящему моменту для данной задачи *ещё не описаны* такие алгоритмы.

После разделения задач на алгоритмически разрешимые и алгоритмически неразрешимые для каждой из разрешимых задач был поставлен следующий вопрос:

> *насколько сложна эта задача?*

При этом сложность здесь понимается не как трудность написания алгоритма для решения данной задачи, и не как размер соответствующей компьютерной программы. Скорее, сложностью здесь можно назвать объём работы, необходимой и достаточной для алгоритмического решения поставленной задачи с конкретными входными данными.

В научно-популярной литературе нередко приводятся такие интересные оценки уровня сложности конкретной задачи: например, говорят о том, что некоторая задача требует для своего решения энергии, превосходящей всю энергию Вселенной; либо – требует больше времени, чем прошло с момента Большого Взрыва... Следовательно, простое существование алгоритма (компьютерной программы) для решения

конкретной задачи – вовсе не признак того, что эта проблема является разрешимой с практической точки зрения.

Именно попытки классификации задач согласно возможности их практического решения на компьютерах – сейчас или в обозримом будущем – и привели к самым красивым (с математической точки зрения) открытиям в различных областях теоретической информатики. В качестве примера рассмотрим т. н. рандомизированные алгоритмы. Большинство известных нам алгоритмов (и соответствующих компьютерных программ) являются детерминированными – т. е. программа и её входные данные полностью (однозначно) определяют все дальнейшие шаги решения задачи.[1] А рандомизированные алгоритмы (программы) могут в каждый момент времени их выполнения иметь несколько возможных вариантов последующих действий – и заранее неизвестно, какой именно из них будет выбран при выполнении программы. Работа рандомизированного алгоритма может быть упрощённо рассмотрена так, как будто алгоритм время от времени «бросает монету» – для того, чтобы определить своё последующее действие, т. е. последующую стратегию поиска правильного ответа. Следовательно, рандомизированная программа может иметь много различных вариантов вычислений – т. е. вариантов своего поведения – для одних и тех же конкретных входых данных. Поэтому, в отличие от детерминированных программ, даже самые лучшие, самые удачные рандомизированные программы в принципе могут давать ошибочные результаты. Следовательно, одно из требований создания качественных рандомизированных программ может быть сформулировано следующим образом: необходимо уменьшить относительное количество (иными словами – просто вероятность) подобных ложных вычислений.

Таким образом, рандомизированные программы кажутся на первый взгляд ненадёжными – в противоположность детерминированным. Почему же они необходимы? Существуют важные задачи, решение которых известными детерминированными алгоритмами требует значительно большего времени работы компьютера, чем это возможно в действительности. Многие из таких задач на первый взгляд представляются неразрешимыми. Но может случиться «чудо» – и этим чудом является рандомизированный алгоритм, который решает проблему в течение нескольких минут, с очень малой вероятностью ошибки, скажем, с вероятностью 10^{-12}.

Но, может быть, стоит назвать подобные программы ненадёжными и «запретить» их? Отрицательный ответ на поставленный вопрос также связан с подсчётом вероятностей: детерминированная программа, которая требует целого дня работы компьютера, является менее надёжной, чем рандомизированная, выполняющаяся за несколько минут – потому что вероятность ошибки аппаратных средств ЭВМ за 24 часа вычислений намного выше вероятности «математической» ошибки быстрой рандомизированной программы.

Конкретный пример практического использования рандомизированной программы – проверка того, является ли заданное число простым. Повсеместно используется т. н. алгоритм шифрования открытым ключом – и для работы этого алгоритма должны быть сгенерированы очень большие простые числа, состоящие примерно из 500 цифр. Первые детерминированные алгоритмы для определения простоты числа были основаны на проверке делимости данного входного значения n. Однако для подобных значений n количество простых чисел, меньших, чем $\sqrt{n}$, превышает число протонов во Вселенной. Следовательно, такие детерминированные алгоритмы фактически яв-

[1] На самом деле, существует разница между понятиями «детерминированность» и «однозначность» – но мы не будем подробно рассматривать этот вопрос. *(Прим. перев.)*

ляются бесполезными. Не так давно был описан новый детерминированный алгоритм для проверки простоты заданного числа, требующий времени $O(m^{12})$, где m – количество двоичных цифр, необходимых для записи входного значения n. Но при этом для проверки простоты числа, состоящего из 500 десятичных цифр, компьютер должен выполнить больше чем 10^{32} команд – это и есть пример программы, которая не может быть выполнена на самых быстрых современных компьютерах за время, прошедшее с момента Большого Взрыва. Но при этом существуют рандомизированные алгоритмы, которые проверяют простоту столь больших чисел в течение минут или даже секунд на простейшем персональном компьютере.

Ещё один очень интересный пример – протокол связи для сравнения содержания двух баз данных, размещённых на двух удалённых компьютерах. Математически доказано, что каждый детерминированный протокол связи, который проверяет такую эквивалентность (иными словами – каждый детерминированный алгоритм сравнения баз данных), требует такого числа побитовых операций, сколько битов имеется в сравниваемых базах данных. Понятно, что для базы данных, содержащей 10^{16} битов информации, провести подобное сравнение практически невозможно. А рандомизированный протокол связи может проверить эту эквивалентность, используя части двух баз данных размером 2000 битов каждая – при этом вероятность ошибки подобного теста меньше 10^{-12}.

Почему такое возможно? Трудно ответить на этот вопрос без некоторых элементарных знаний основ компьютерных наук. Поиск объяснения подобных возможностей рандомизированных алгоритмов – это очень красивая теория, затрагивающая самые глубокие основы математики, философии и естествознания.

Природа – наш лучший учитель, и случайность играет в ней значительно бóльшую роль, чем может показаться на первый взгляд. Специалисты в области компьютерных наук могут привести множество примеров подобных систем – где требуемые характеристики и поведение достижимы только через концепцию рандомизации. В подобных случаях каждая надёжная детерминированная система составлена из миллиардов подсистем, и эти подсистемы должны правильно взаимодействовать. Понятно, что такая комплексная система, сильно зависящая от своих многочисленных подкомпонентов, не являются жизнеспособной: ошибку, в случае её возникновения, крайне сложно обнаружить. И, само собой разумеется, затраты по сопровождению подобной системы можно назвать астрономическими. С другой стороны, можно разаботать небольшие рандомизированные системы с требуемым поведением. Вследствие своего малого размера они недороги, а работа их компонентов легко поддаётся проверке. И что особенно важно – вероятность неправильного поведения системы настолько мала, что ею можно просто пренебречь.

Несмотря на отмеченные научные аспекты, для многих учёных информатика – типичная проблемно-ориентированная практическая инженерно-техническая дисциплина. Она не только включает такие технические вопросы, как

> *организацию процессов разработки, формулировку стратегических целей и имеющихся ограничений, моделирование, составление спецификаций, проверку качества, тестирование, интеграцию в уже существующие системы, повторное использование, поддержку и сопровождение,*

но и охватывает такие вопросы управления, как

организацию команды и руководство ею, оценку производительности, планирование, качественное управление, оценку временны́х планов, сдачу продукта, договорные обязательства, маркетинг.

Специалист в области компьютерных наук должен также быть настоящим прагматиком: при создании комплексых программных и аппаратных систем нужно часто принимать решения, основанные на собственном опыте, поскольку обычно нет возможности моделировать и анализировать сложную окружающую обстановку.

При рассмотрении нашего определения информатики может создаться впечатление, что её изучение слишком сложно: ведь необходимы и математические знания, и понимание естественнонаучных процессов, и владение некоторыми инженерными навыками. Эти требования действительно можно было бы назвать слишком сильными – но ведь они тоже формируют преимущества такого подхода к образованию!

Поясним последнюю мысль. Главным недостатком современной науки можно считать её сверхспециализацию – которая ведёт к независимому развитию её весьма небольших разделов. К настоящему времени каждая область науки разработала свой собственный язык, часто непонятный даже исследователям, работающим в смежных областях. Подобная ситуация зашла так далеко, что стандартные утверждения в одной отрасли науки часто воспринимаются как весьма поверхностные и недопустимые в другой – что сильно замедляет процесс междисциплинарных исследований...

А ведь информатика «междисциплинарна» по своей сути! Она находит применение, предлагает решение проблем во многих научных областях, а также в нашей повседневной жизни – т. е. всюду, где в принципе возможно использование компьютеров. Информатика обладает весьма широким спектром методов – начиная от точных (формальных) математических методов и заканчивая методами совершенно неформальными, иными словами – «методами правого полушария мозга», появляющимися у учёного на основе предыдущего опыта его инженерных разработок. Итак, возможность одновременного (т. е. в рамках одной научной дисциплины – информатики) изучения разных «языков» разных областей науки, с использованием разных путей исследования, разных видов человеческого мышления – это и есть самый дорогой подарок, который информатика дарит человеку, изучающему её.

1.2 Замечательная теория

Итак, наша книга посвящена элементарному введению в основные принципы теоретической информатики. Это – очень красивая научная дисциплина. Благодаря своим интересным результатам и тесным связям с другими науками она внесла большой вклад в наше мировоззрение.

Однако для студентов теоретическая информатика – далеко не самый любимый вузовский предмет; и этот факт могли бы подтвердить соответствующие статистические исследования. Многие студенты даже рассматривают теоретическую информатику просто как некоторое препятствие, которое необходимо преодолеть для получения высшего образования. Существует несколько причин появления такой широко распространённой точки зрения. Одна из этих причин такова: среди всех областей информатики именно теоретическая информатика включает наибольшую математическую составляющую – следовательно, лекции по теоретическим основам информатики относятся к самым трудным курсам.

Поэтому многие студенты начинают изучение этих курсов, имея неправильное первое впечатление от информатики, и, более того, многие преподаватели-лекторы проводят эти курсы недостаточно привлекательным для студентов способом. Чрезмерная, слишком точная детализация математических доказательств, сиюминутное увлечение техническими подробностями представления данных – плюс недостаточное побуждение студентов к изучению данной науки, недостаточное понимание ими важности этого изучения, недостаточные знания о возможнотях использования компьютера – всё это может разрушить образ любой области науки, даже столь красивой, как теоретическая информатика.

В нашем предыдущем описании информатики – как науки многосторонней, многогранной, многопрофильной – мы уже обращали внимание читателя на важность понимания её основных теоретических принципов. Перечислим причины изучения теоретических основ информатики.

1. *Философская глубина.*
   Теоретическая информатика исследует знание и разрабатывает новые идеи, концепции и методы – которые влияют на основы науки вообще. Она даёт частичные или полные ответы на такие философские вопросы, как:
   - Существуют ли проблемы, которые автоматически (алгоритмически) разрешить нельзя? Если они существуют – то где именно находится граница между алгоритмической разрешимостью и алгоритмической неразрешимостью?
   - Существуют ли недетерминированные и рандомизированные процессы, которые могут вычислять то, чего не могут детерминированные? Являются ли недетерминированные и рандомизированные процессы и алгоритмы более эффективными, чем детерминированные?
   - Как определить понятие сложности (трудности) для некоторой задачи?
   - Где границы алгоритмической разрешимости – с точки зрения практического использования соответствующих алгоритмов?
   - Что является математическим доказательством? Является ли алгоритмический *поиск* некоторого математического доказательства более сложным, чем алгоритмическая *проверка* правильности некоторого *данного* доказательства?
   - Как определить случайный объект?

   Важно отметить, что многие из этих вопросов без формального определения понятий алгоритма и вычисления вообще не могут быть чётко сформулированы. Таким образом, теоретическая информатика обогатила язык науки этими новыми терминами – тем самым способствуя развитию как этого языка, так и науки вообще. Многие известные основные категории науки (такие как детерминизм, шанс, недетерминизм) в теоретической информатике обрели новый смысл, что повлияло на наше общее представление об окружающем мире.

2. *Практические приложения и красивые результаты.*
   Теоретическая информатика очень важна и для практики. С одной стороны, она обеспечивает методологический подход, определяющий первоначальную стратегию решения алгоритмических проблем. А с другой стороны, теретическая информатика описывает специальные понятия и методы, которые могут быть применены в течение всего процесса разработки программного проекта и его выполнения. Более того, без некоторых понятий, концепций и методов теоретической информатики

было бы невозможно создать многие программные приложения, широко используемые на практике.

Как уже было отмечено выше, помимо понятия рандомизированных алгоритмов есть много других «чудес», рождённых теоретической информатикой. Существуют различные сложные оптимизационные задачи, для которых совсем небольшое ослабление их ограничений настолько уменьшает сложность, что позволяет перейти от практически невыполнимого алгоритма к возможности получения решения за несколько минут. И очень важно отметить, что часто подобные ослабления ограничений являются настолько несущественными, что на них просто не стоит обращать внимания.

Рассмотрим такие примеры. Как вы полагаете – можно ли заставить кого-либо поверить в то, что вы знаете некоторый секретный код (пароль), если при этом вы не видели ни одного бита этого кода? Могут ли два человека определить, кто из них старше, не называя друг другу свой возраст? Можно ли с почти полной уверенностью сказать, верно ли некоторое математическое доказательство объёмом в нескольких тысяч страниц, не читая его подробно, а только посмотрев на несколько беспорядочно выбранных его отрывков? Всё это возможно! Благодаря теоретической информатике то, что первоначально казалось невозможным, становится возможным, а любые исследования в этой науке полны неожиданностей.

3. *Продолжительность жизни знания.*
   Благодаря интенсивному развитию новых технологий – причём как технологий создания программного обеспечения, так и «железа» (аппаратных средств ЭВМ) – мир прикладной информатики непрерывно изменяется и развивается. Половина существующей информации о программном обеспечении и аппаратных средствах полностью устаревает за 5 лет. Поэтому если в студенческих курсах непропорционально много времени посвящено изучению аппаратных средств, операционных систем и зависящим от конкретных трансляторов тонкостям разработки программ, то подобное образование не обеспечивают соответствующих перспектив для будущей работы студента. А понятия и методы теоретической информатики имеет гораздо более длинную среднюю продолжительность жизни – скажем, несколько десятилетий – и поэтому образование, связанное именно с её первоочередным изучением, будет верой и правдой служить своему владельцу в течение длительного времени.

4. *Междисциплинарная направленность.*
   Как уже было отмечено, теоретическая информатика междисциплинарна по своей сути – и поэтому может принять (и принимает) участие во многих захватывающих направлениях научных исследований: в проекте расшифровки генома, в медицинской диагностике, в проблемах оптимизации для различных областей технических наук и экономики, в автоматическом распознавании речи, в исследованиях космоса – причём, понятно, мы привели здесь только очень незначительную часть возможных областей её применения.

   Итак, теоретическая информатика вносит большой вклад в другие области науки – и сама обогащается за счёт этого. Например, изучение вычислений на уровне элементарных частиц, поведение которых подчинено правилам квантовой механики, направлено на эффективную реализацию вычислений в микромире – а в макромире подобные вычисления к успеху не приводят. Уже существует теоретическая модель

квантового компьютера, но его практическая реализация – «вызов, брошенный информатикой физике». При этом в настоящее время никто не может оценить всех последствий возможного в недалёком будущем успешного создания такого компьютера. Кроме того, совершенно независимо от возможного успеха данного проекта, правила микромира являются настолько удивительными, настолько противоречат тому, что называется интуицией (т. е. противоречат нашим обычным взглядам, нашему опыту макромира), что учёные-информатики ожидают ещё очень многих «чудес», связанных с использованием квантовой теории. Например, уже сегодня очевидно, что в микромире существует надёжная и безопасная связь, поскольку любая несанкционированная попытка чтения некоторого сообщения может быть обнаружена и отражена, а информация о ней передана отправителю этого сообщения.
Ещё одна интересная область практического применения теоретической информатики – это алгоритмы и соответствующие вычисления, связанные с молекулами ДНК. Биологи называют молекулы ДНК информационными курьерами – поэтому не удивительно, что эти молекулы могут применяться для хранения и передачи информации. Сегодня мы уже знаем, что молекулы ДНК могут моделировать работу компьютера; причём это – не только теоретический факт: в лабораториях было выполнено несколько проектов по моделированию вычислений компьютера специальными химическими опытами с молекулами ДНК. Нельзя исключить и скорое появление т. н. биокомпьютеров, в которых несколько молекул ДНК способны выполнять работу всей электронно-вычислительной машины.

5. *Способ мышления.*
Учёные-математики считают, что их наука развивает, обогащает и даже формирует способ мышления – и тем самым способствует всестороннему развитию человека. Но раз так высоко оценён вклад математики, то нельзя не увидеть и вклада информатики – причём вклада в те же самые аспекты человеческого мышления.
Теоретическая информатика поощряет создание и анализ математических моделей реальных систем, а на их основе – поиск концепций и методов решения конкретных проблем. Точное понимание того, какие именно особенности реальной системы должны быть полностью описаны в некоторой соответствующей ей модели, а какие именно её характеристики могут быть рассмотрены приближённо или даже проигнорированы – это главное условие успеха, причём как в науке, так и при разработке конкретного программного обеспечения. Поэтому теоретическая информатика уделяет особое внимание разработке математических концепций и моделей, строго соответствущих реальным проблемам. Таким образом, изучение теоретической информатики помогает понять, как объединять теоретические знания с практическим опытом, а на основе этого объединения – развивать своё мышление так, чтобы с его помощью «атаковать мировые научные проблемы».

1.3 Студентам

Эта книга написана для вас. Цель книги – не только ввести некоторые основные понятия информатики, но и вдохновить читателя на дальнейшую работу. Оставляем вам ответ на вопрос, насколько это удалось.

В предыдущих разделах этой главы мы попытались убедить читателя в том, что теоретическая информатика – прекрасная наука, наполненная радостями и волнениями. В качестве доказательства того, что есть люди, действительно наслаждающиеся работой в области теоретической информатики, автор хотел бы процитировать одного из своих друзей-коллег. Приведём некоторые мысли, высказанные им на лекциях и научных семинарах.

> «И если даже после такого воздействия эта глупая вероятность ошибки всё ещё не станет ниже 0.5 – то мы её заменим на другую характеристику! Она будет глубоко раскаиваться в том, что имела наглость испортить мою модель.»

> «Сегодня мы поставили ϵ на колени, выпустили из него пар. Когда мы сначала этот ϵ аппроксимировали, он нервно дёргался – но лишь немного. Но затем тяжёлые удары полуопределённой матрицы сделали его безразличным ко всему окружающему. И это было замечательно!»

> «Вы так говорите? Но никакая λ не сможет осквернить наше μ своими ударами – без того, чтобы не расплатиться за это. Мы дадим этой лямбде в ухо, а она так и не поймёт, почему с ней такое случилось.»

Вы считаете такие эмоции слишком экспрессивными для строгой науки? А автор так не считает! Большая радость – участвовать в таких лекциях и научных семинарах, где, слушая подобные образные выражения, участники сразу понимают, какие именно трудности им надо преодолеть. Страстное увлечение – вот главная движущая сила как хорошего образования, так и научного исследования. Поэтому считайте, что как только у вас возник интерес к теме научного исследования, вы уже сделали половину дела... А у вас уже появился интерес к теоретической информатике? Если нет – то, может быть, стоит посвятить ваши учебные и научные занятия другим дисциплинам, лежащим за пределами компьютерных наук?

В этой книге рассматриваются некоторые основные аспекты теоретической информатики. Почему изучение этой науки считается трудным? Нелегко освоить методы, имеющие широкое практическое применение – и этот факт не должен удивлять. Если человек хочет пробегать стометровку быстрее 10 секунд, или прыгать за 8-метровую отметку, то он должен потратить многие годы на упорные тренировки. Чтобы достигнуть чего-нибудь значительного, надо приложить значительные усилия. И приобретение знаний – не исключение. Возможно, вы даже столкнётесь с большим сопротивлением в течение этой гонки за знаниями, потому что – в отличие от спортивных состязаний – ваша заинтересованность может поколебаться, т. к. вы можете временно потерять из виду саму цель этой гонки.[2] А чтобы не потерять эту цель, нужно терпение, более того, нужна готовность к постоянному повторению любой темы – для того, чтобы добиться более глубокого понимания взаимосвязи между темами в контексте целой теории, в контексте всей теоретической информатики как науки.

В этой книге мы стремимся к тому, чтобы облегчить освоение некоторых фундаментальных областей теоретической информатики. Для этого будем использовать следующие три концепции.

1. *Простота и ясность*

[2] А возможно, вы даже не видите этой цели в начале процесса обучения!

Мы будем объяснять простые понятия простыми же терминами, избегать использования ненужных математических абстракций. Иными словами, мы постараемся излагать теорию ясно и конкретно – настолько, насколько это возможно. При этом мы будем строить наше введение в теоретическую информатику на основе лишь элементарных математических знаний, практически не выходящих за пределы средней школы. А перед изложением сложных формальных доказательств будем стараться объяснять их основные идеи простым и ясным способом.

Пункты (некоторые леммы, теоремы, доказательства, упражнения), помеченные знаком «*», являются более сложными, более техническими – новичкам эти пункты стоит пропустить. Техническое обсуждение работы машины Тьюринга в том разделе, где речь идёт о вычислимости, также является факультативным – поскольку понимание вычислимости может быть получено не только с помощью машины Тьюринга, но и с помощью любого языка программирования.

В описании полученных результатов приоритет будет отдаваться ясности изложения. Если некоторый интересный, но относительно неполный результат легко объясняется с помощью простых аргументов – то мы будем выбирать именно этот путь, а не объяснять некоторый более сильный результат, требующий, однако, аргументов более запутанных.

Всюду в этой книге мы будем стараться действовать систематически; это означает, что, переходя от более простого к более сложному, мы будем делать небольшие шаги, избегая скачков в изложении наших мыслей.

2. *«Лучше меньше, да лучше»*

Авторы многих учебников неправильно предполагают, что их главная цель – просто предоставить читателю очередную порцию информации. При этом они идут неверным путём, пытаясь просто дать читателю максимальный объём знаний за минимальное время и при минимальном объёме текста. Эта поспешность обычно приводит к тому, что в таком учебнике излагается большое количество мелких результатов – и, вследствие этого, игнорируется общий контекст всего курса.

А философия нашей книги иная. Мы будем стараться следить за мышлением студента, влиять на способы, на пути мышления. Нас интересуют историческое развитие системы понятий информатики, а также принятые в ней способы мышления – а представленные в книге основные определения, результаты, доказательства и методы являются только средством для этого изложения. Следовательно, мы не очень обеспокоены количеством информации нашей книги, предпочитая жертвовать частью её материала – скажем, в пределах от 10 до 20% объёма. Вместо этих процентов мы будем уделять больше времени рассмотрению необходимости изучения некоторого понятия, связи между теорией и практикой – а особенно внутреннему контексту представляемой теории.

Мы будем особенно внимательно следить за введением новых терминов: понятия и определения не будут появляться внезапно, как это делается во многих других лекционных курсах, использующих формальный язык математики. Ведь формально определённые термины всегда являются некоторым приближением, некоторой абстракцией интуитивных идей – а формализация этих идей позволяет нам делать точные утверждения и заключения о некоторых объектах и событиях. Поэтому мы будем стараться объяснять наш выбор конкретных вариантов формализации терминов и используемых моделей, будем указывать ограничения на их использование. Итак, мы убеждёны, что очень важно научиться работать на уровне создания

новых терминов (основных определений) и моделей для них – поскольку именно на этом уровне и происходит бóльшая часть движения вперёд в освоении науки.

3. *«Повторение – мать учения»*
   Эта книга написана таким образом, что заставляет читателя возвращаться к определённым ранее понятиям и методам. Для этого каждая глава начинается с раздела «Цели и задачи», в котором, кроме собственно целей и задач главы, рассматривается отношение материала главы к содержанию предыдущих. Ядро каждой главы посвящено: изложению некоторых неформальных идей; их формализации – с помощью специальных теоретических понятий; а также изучению этих идей в границах введённых понятий. При каждом существенном продвижении в изложении материала, содержащем введение большого числа новых понятий и методов, объясняется важность такого продвижения – применительно к рассматриваемым нами целям. А заканчивается каждая глава кратким заключением, включающим как неформальное повторение полученных результатов, так и некоторые возможные перспективы дальнейшего научного развития всего представленного в ней материала; при этом обсуждаются современные границы между уже достигнутыми знаниями и результатами, получение которых возможно в будущем.
   И, как обычно в учебниках, процесс изучения поддерживается соответствующими упражнениями. Но мы не «привязываем» эти упражнения к отдельным разделам и подразделам – они распределены по главам таким образом, чтобы читатель смог выполнять их сразу после освоения соответствующего материала. Упражнения служат не только для того, чтобы освоить на практике применение рассмотренных в книге понятий и методов, но и для того, чтобы углубить теоретическое понимание всего курса.

Итак, наша цель состоит не только в том, чтобы ввести вас в захватывающий мир информатики, но и в том, чтобы предложить вам простой «пропуск» в этот мир. Однако простота вовсе не означает, что книга недостаточно строга – это означает, что все вопросы, кратко упомянутые выше в данном введении, будут представлены настолько чётко и ясно, что читатель сможет освоить материал книги за очень короткое время. Необходимые требования для успешного использования этого «пропуска» минимальны – небольшой программистский опыт (эквивалентный работе в течение первого семестра обучения), а также базовые математические знания. Такие стандартные курсы, как «Архитектура ЭВМ», «Алгоритмы и структуры данных», вовсе не являются необходимыми – хотя и могут быть полезными для лучшего понимания материала книги.

1.4 Структура книги

Книга разделена на 9 глав, включая данное введение (первую главу).

Вторую главу можно назвать трамплином. Здесь введены *формальный язык* информатики и способы представления объектов, целей и задач, нужных для её исследований. Любой вычислительный процесс компьютера может рассматриваться как преобразование одного текста в другой – поскольку мы всегда можем считать, что входные и выходные данные суть тексты. Во второй главе предлагаются основные принципы работы с текстами, которые используются для разработки формальных описаний алгоритмических проблем. Кроме того, во второй главе мы отвечаем на вопросы, как

можно измерить количество информации в некотором тексте и когда некоторый текст можно считать случайным.

В третьей главе в качестве простейшей вычислительной модели рассматриваются *конечные автоматы*. Важно отметить, что целью главы вовсе не является введение в теорию автоматов; цель – подготовить читателя к сложному определению формальной модели алгоритмов (программ). Мы используем конечные автоматы для простого введения следующих ключевых понятий информатики: состояние и конфигурация вычислительной модели; вычисление; шаг вычисления; детерминизм; недетерминизм; описательная сложность; моделирование. Это облегчает понимание данных понятий в общей структуре модели машины Тьюринга, рассматриваемой далее.

Четвёртая глава посвящена *машине Тьюринга*, которая является формальной моделью интуитивного понятия *алгоритм*. Поскольку работа машины Тьюринга отражает работу реального компьютера, выполняющего программу, написанную в машинных кодах, – то мы постараемся ограничиться в этой главе минимальным количеством понятий, а именно только теми из них, которые действительно необходимы для понимания этого формализма.

Пятая глава – введение в *теорию вычислимости*. В ней мы ставим такой вопрос:

какие проблемы являются алгоритмически разрешимыми – а какие нет?

Мы описываем специальные методы, которые могут положительно ответить на этот вопрос – для некоторой конкретной задачи. При этом мы работаем на двух уровнях. Аргументация первого уровня использует только интуитивное понимание понятия программы – а на втором уровне применяются формальные доказательства, базирующиеся на модели машины Тьюринга.

Шестая глава – введение в *теорию сложности*. Главные вопросы этой главы – следующие:

как измерять сложность алгоритмических проблем?

действительно ли существуют задачи сколь угодно большой сложности?

Сложность некоторой вычислительной проблемы можно рассматривать как «количество» работы компьютера, необходимой и достаточной для решения данной задачи с помощью некоторого алгоритма. В этой главе мы сначала представляем фундаментальные результаты о мерах и классах сложности. Некоторые доказательства являются слишком технически сложными для поставленной нами задачи введения в теоретическую информатику – поэтому мы их опускаем. Мы показываем, что существуют задачи столь большой вычислительной сложности, что для их алгоритмического решения было бы недостаточно энергии целой Вселенной.

Наиболее важное из понятий этой главы – NP-сложность, которое может рассматриваться как метод классификации задач на группы «практически разрешимых» и «практически неразрешимых». Это понятие основано на изучении отношения между детерминированными и недетерминированными вычислениями – одного из основных предметов исследования в теоретической информатике. Любое известное детерминированное моделирование недетерминированных алгоритмов требует экспоненциального роста временно́й сложности – и трудно поверить, что существует эффективное «моделирование недетерминизма детерминизмом». Мы приводим здесь важный аргумент, помогающий поверить в *не*существование такого моделирования; этот аргумент касается философских принципов основ математики. Мы показываем, что в некоторой

специальной структуре сложность решения проблемы детерминированным алгоритмом соответствует сложности *создания* доказательства математической теоремы – в то время как сложность решения проблемы недетерминированным способом соответствует сложности *проверки правильности* заданного математического доказательства. Следовательно, вопрос о том, является ли недетерминизм более мощным средством, чем детерминизм, эквивалентен вопросу о том, легче ли проверить данное доказательство, чем найти его.

В седьмой главе представлены несколько «жемчужин» создания *алгоритмов.*[3] Эта глава – продолжение предыдущей; основной её вопрос таков:

> *что мы можем сделать с трудными задачами, для решения которых лучшие алгоритмы требуют годы работы компьютера?*

Мы приводим здесь понятия псевдополиномиальных алгоритмов, алгоритмов локального поиска, аппроксимационных алгоритмов, а также эвристического алгоритма имитационной нормализации. Мы объясняем методологию этих понятий, которая основывается на том факте, что небольшое ослабление ограничений может привести к огромному качественному скачку – от практически нереализуемой вычислительной сложности первоначальной проблемы до нескольких минут работы компьютера. Например, если ослабить требование получить *оптимальное* решение некоторой оптимизационной задачи до требования получить *допустимое* решение, качество которого не слишком отличается от качества оптимального – то можно существенно уменьшить вычислительную сложность рассматриваемой оптимизационной задачи.

В случае *рандомизированных* алгоритмов мы ослабляем требование гарантированного получения правильного решения – на требование получения решения, являющегося правильным с некоторой, достаточно большой вероятностью. Понятие рандомизации относится к основным понятиям создания алгоритмов для труднорешаемых задач – и с этой точки зрения соответствующий материал мог бы являться частью предыдущей главы. Однако, поскольку эти вопросы представляют собой огромную важность для многих основных областей информатики – как теоретических, так и практических – мы посвящаем им целую главу, восьмую. В ней мы не ограничиваемся простым описанием эффективных рандомизированных алгоритмов (например, алгоритма рандомизированной проверки простоты заданного числа); мы также указываем некоторые из причин, приводящих рандомизированные алгоритмы к успеху. Мы рассматриваем метод избытка свидетельств и т. н. дактилоскопический метод – для иллюстрации основных принципов построения таких алгоритмов.

В заключительной, девятой главе, рассматриваются *проблемы связи.* Благодаря крупным технологическим достижениям последних лет мы получили возможность передавать очень большие объёмы информации. Вследствие этого создание алгоритмов для решения проблем связи стало бурно развивающейся частью информатики. В начале главы приводится краткий обзор безопасной связи. Мы начинаем с нескольких примеров криптографических систем, затем вводим понятие открытого ключа; оно используется для доказательства невозможности подделки подписи в подобных системах. Далее мы вводим системы доказательств с нулевым разглашением, которые играют очень важную роль в криптографии. А заканчивается глава разработкой коммуни-

[3] Здесь мы имеем в виду вопросы, не связанные ни с курсом «Алгоритмы и структуры данных», ни с классическими лекциями по алгоритмам. Эти вопросы связаны только с разработкой конкретных алгоритмов для рассматриваемых нами труднорешаемых задач.

кационной сети – этот проект иллюстрирует некоторые задачи, возможные в данной области.

Мистика – это основа
всей истинной науки,
а человек, который больше не способен
«остолбенеть в благоговении»,
возвращает нас к науке ведьмовства,
из которой произросла вся духовная работа.
Наша весёлая, плодотворная жизнь докажет миру,
что видение священной реальности
прочно покоится на реальности материальной.
А. Эйнштейн

2

Алфавиты, слова, языки, алгоритмические проблемы

2.1 Цели и задачи главы

Внимательно рассматривая любой вычислительный процесс, выполняемый компьютером, мы приходим к выводу, что компьютер работает с *текстами*; а тексты являются последовательностями символов некоторого заданного алфавита. Программы тоже являются текстами – над алфавитом, состоящим из символов клавиатуры компьютера. Все данные представлены в компьютере как последовательности нулей и единиц, любые входы и выходы также являются текстами – или, по крайней мере, могут интерпретироваться как тексты над подходящим алфавитом. С этой точки зрения каждая программа преобразовывает входные тексты в тексты выходные. Итак, тексты в вычислительных процессах – всюду и везде.

В данной главе рассматриваются основные принципы работы с текстами. Сформулируем её основные цели.

- Первая заключается в том, чтобы ввести подходящий формализм для работы с текстами – представлениями данных. Основные вводимыми здесь понятиями являются *алфавит, слово и язык.* Эти термины необходимы в качестве базиса для формального определения фундаментальных понятий информатики – алгоритма (программы), вычисления и т.д. В разделе 2.2 мы научимся работать с этими основными понятиями, использовать их для представления данных, а также выполнять некоторые основные действия над текстами.
- Вторая цель главы – показать, как использовать введённые понятия, применять их для получения формальных представлений алгоритмических проблем.[1] Мы уделим особое внимание двум их классам – проблемам принадлежности (разрешимости)[2] и оптимизационным проблемам.

[1] В литературе на английском языке слово «problem» употребляется чаще слова «task» (и *не* является его полным синонимом) – в отличие от русской литературы, где слово «проблема» употребляется реже слова «задача». В настоящем переводе мы, следуя оригиналу, чаще используем слово «проблема». *(Прим. перев.)*

[2] Эти варианты перевода понятия «decision problem» представляются несколько более удачными, чем другие, также использующиеся в литературе на русском языке – «проблема

- Третья и последняя цель этой главы – рассмотрение некоторых вопросов, связанных со сжатием текстов. Для этого мы вводим понятие сложности по Колмогорову. Данная характеристика может быть использована не только для определения самого короткого представления текстов (данных) – но и для измерения количества информации в тексте, а также для разумного определения понятия «случайный текст».[3] Ещё один очень важный момент: сложность по Колмогорову является очень мощным инструментом исследования компьютерных вычислений; мы неоднократно будем использовать это понятие и в последующих главах.

2.2 Алфавиты, слова и языки

Для обработки информации мы представляем данные (объекты обработки) как последовательности символов. Как и в случае естественных языков, мы начинаем с того, что фиксируем для представления данных специальный набор символов – алфавит. Ниже символом $\mathbb{N} = \{1, 2, \ldots\}$ мы будем обозначать множество натуральных чисел.[4] $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$.

Определение 2.1. Алфавитом *называется любое непустое конечное множество. Каждый элемент алфавита называется* **символом**.

Итак, смысл слова «алфавит» – тот же самый, что и для естественных языков. Алфавит применяется для создания т. н. системы письма – т. е. для письменного представления языка. Важно отметить, что для представления изучаемых объектов мы всегда сможем выбирать алфавит, содержащий конечное число символов.

Перечислим некоторые часто используемые «стандартные» алфавиты.

- $\Sigma_{\text{bool}} = \{0, 1\}$ – логический (Булевый) алфавит, используемый в теоретической информатике.
- $\Sigma_{\text{lat}} = \{a, b, c, \ldots, z\}$ – латинский алфавит.
- $\Sigma_{\text{keyboard}} = \Sigma_{\text{lat}} \cup \{A, B, \ldots, Z, \sqcup, >, <, (,), \ldots, !\}$ – алфавит всех символов, которые можно набрать на клавиатуре компьютера; при этом символ $\sqcup$ означает пробел.
- $\Sigma_m = \{0, 1, 2, \ldots, m-1\}$ для каждого $m \geq 1$ – алфавит для записи чисел в m-ичной системе счисления.
- $\Sigma_{\text{logic}} = \{0, 1, x, (,), \wedge, \vee, \neg\}$ – алфавит, который используется для записи формул алгебры логики.

Для дальнейшей работы определим слово как некоторую последовательность символов. В отличие от естественных языков, где слово – элемент языка, в терминологии, принятой в компьютерных науках, термин «слово» означает произвольный текст.

(задача) принятия», «проблема распознавания» и др. Во всех вариантах перевода имеется в виду, что мы должны *алгоритмически* определить, принадлежит ли некоторое заданное слово рассматриваемому языку – т. е. принимает ли это слово рассматриваемый алгоритм-распознаватель. *(Прим. перев.)*

[3] Заметим, что мы только что кратко описали один из случаев вклада информатики в науку вообще – причём вклада на философском, весьма фундаментальном уровне. Ниже мы дадим подробное и согласующееся с нашей интуицией объяснение того, когда именно некоторый объект, некоторое событие могут быть объявлены случайными.

[4] В оригинале было $\mathbb{N} = \{0, 1, 2, \ldots\}$ (включая 0) – мы же будем использовать вариант, принятый в литературе на русском языке. *(Прим. перев.)*

Определение 2.2. *Пусть Σ – некоторый алфавит.* **Слово** *над Σ – любая конечная последовательность символов алфавита Σ.* **Пустое слово $\boldsymbol{\lambda}$** *– единственное слово, состоящее из нулевого количества символов.*[5]
Длина слова $\boldsymbol{w}$ *над Σ, обозначаемая как $|\boldsymbol{w}|$, есть число символов в этом слове (т. е. длина слова, рассматриваемого как последовательность символов).*
Множество всех слов над алфавитом Σ обозначается записью $\boldsymbol{\Sigma^*}$. $\boldsymbol{\Sigma^+} = \Sigma^* - \{\lambda\}$ означает множество всех слов за исключением пустого.

Последовательность $0, 1, 0, 0, 1, 1$ – слово над алфавитами Σ_{bool} и Σ_{keyboard}. Длина этого слова – $|0, 1, 0, 0, 1, 1| = 6$. Пустое слово λ – слово над любым алфавитом, причём $|\lambda| = 0$.
Соглашение. В дальнейшем изложении при записи слов мы будем опускать запятые, т. е. будем писать

$$x_1 x_2 \dots x_n \quad \text{вместо} \quad x_1, x_2, \dots, x_n.$$

Таким образом, мы будем писать 010011 для представления слова $0, 1, 0, 0, 1, 1$.

Важно отметить, что символ пробела $\sqcup$ над алфавитом Σ_{keyboard} отличен от λ, поскольку $\sqcup$ – элемент алфавита Σ_{keyboard}, следовательно $|\sqcup| = 1$. Используя символ $\sqcup$, можно рассматривать книгу (или, например, компьютерную программу) как слово над алфавитом Σ_{keyboard}.

$$\begin{aligned}(\Sigma_{\text{bool}})^* &= \{\lambda, 0, 1, 00, 01, 10, 11, 000, 001, 010, 100, 011, \dots\} \\ &= \{\lambda\} \cup \{x_1 x_2 \dots x_i \mid i \in \mathbb{N}_0,\ x_j \in \Sigma_{\text{bool}} \text{ для } j = 1, \dots, i\}.\end{aligned}$$

Из этого примера видно, что существует возможность перечисления всех слов над заданым алфавитом. При этом слова записываются в порядке возрастания их длин – т. е. одно слово за другим для каждой рассматриваемой длины.

Упражнение 2.3. Пусть Σ – некоторый алфавит. Для каждого $i \in \mathbb{N}_0$ посчитайте, сколько слов длины i существует над алфавитом Σ .

Упражнение 2.4. Пусть $\Sigma = \{0, 1, \#\}$, и пусть k и n – некоторые натуральные числа, причём такие, что $k \leq n$.

- Посчитайте число слов длины n, которые содержат ровно k вхождений символа 0.
- Посчитайте число слов длины n, которые содержат не более чем k вхождений символа 0.

Слова могут использоваться для представления различных объектов – таких как числа, формулы, графы, программы. Слово

$$x = x_1 x_2 \dots x_n \in (\Sigma_{\text{bool}})^*, x_i \in \Sigma_{\text{bool}} \text{ для } i = 1, \dots, n$$

может рассматриваться как двоичное представление неотрицательного целого числа

$$\mathit{Number}(x) = \sum_{i=1}^{n} 2^{n-i} \cdot x_i.$$

[5] В некоторых книгах вместо λ используются другие варианты обозначения пустого слова: Λ, ϵ, e.

Наоборот, для любого неотрицательного целого числа m запись

$$Bin(m) \in \Sigma^*_{\text{bool}}$$

обозначает наиболее короткое двоичное представление числа m.[6] Следовательно,

$$Number(Bin(m)) = m.$$

Упражнение 2.5. Двоичное представление каждого натурального числа m начинается с 1. Чему равна длина слова $Bin(m)$ для заданного m?

Упражнение 2.6. Пусть $x \in (\Sigma_m)^*$ для некоторого натурального $m \geq 2$. При этом пусть x – представление в m-ичной системе счисления неотрицательного целого числа $Number_m(x)$. Как посчитать $Number_m(x)$ по заданному x?

Последовательность целых чисел $a_1, a_2, \ldots, a_m$, где $m \in \mathbb{N}_0$ и $a_i \in \mathbb{N}_0$ для каждого $i = 1, \ldots, m$, может быть представлена как

$$Bin(a_1)\#Bin(a_2)\#\cdots\#Bin(a_m) \in \{0, 1, \#\}^*.$$

В дальнейшем мы обычно будем использовать именно такое представление множеств и последовательностей целых чисел.

Пусть $G = (V, E)$ – ориентированный граф, V – его множество вершин, а $E \subseteq \{(u, v) \mid u, v \in V,\ u \neq v\}$ – множество дуг. Пусть $|V| = n$ – число элементов множества V. Мы можем представить G с помощью т. н. матрицы смежности M_G. Это – логическая матрица (т. е. матрица, все элементы которой равны 0 или 1) $M_G = [a_{ij}]$, имеющая размерность $n \times n$, такая, что

$$a_{ij} = 1 \iff (v_i, v_j) \in E.$$

Иными словами, условие $a_{ij} = 1$ означает, что G включает дугу (v_i, v_j) из v_i в v_j – а условие $a_{ij} = 0$ означает, что в G такой дуги не существует. Логическая матрица M может быть представлена словом над алфавитом $\{0, 1, \#\}$ следующим образом. Будем писать строки матрицы M одну за другой, а символ # использовать для обозначения конца каждой строки.

В качестве примера рассмотрим граф, приведённый на рис. 2.1.
Соответствующая ему матрица смежности такова:

$$\begin{pmatrix} 0\,0\,1\,1 \\ 0\,0\,1\,1 \\ 0\,1\,0\,1 \\ 0\,0\,0\,0 \end{pmatrix}.$$

Предложенное представление этой матрицы – слово

$$0011\#0011\#0101\#0000\#.$$

Очевидно, что описанное нами представление является однозначным. Последнее означает, что по заданному слову, являющемуся представлением некоторого графа, мы можем построить соответствующий (исходный) граф – причём единственным образом.

[6] Требование, чтобы $Bin(m)$ было наиболее коротким двоичным представлением числа $Number(Bin(m)) = m$, означает всего лишь, что первый символ слова $Bin(m)$ должен быть равен 1.

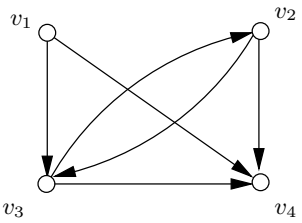


Рис. 2.1.

Упражнение 2.7. Описанное выше представление графа – как слово над алфавитом $\{0, 1, \#\}$ – имеет длину $n(n+1)$ для любого графа, состоящего из n вершин. Попробуйте разработать другой, более короткий способ однозначного представления графа.

Упражнение 2.8. Разработайте представление графа над алфавитом Σ_{bool}.

Входными данными для многих алгоритмических проблем являются взвешенные неориентированные графы $G = (V, E, h)$, где h – функция, действующая из E в $\mathbb{N}$. Это означает, что каждой дуге $e \in E$ приписано некоторое число, смысл которого – вес, или стоимость. Такие графы тоже могут быть представлены с помощью матрицы смежности. Аналогично, равенство $a_{ij} = 0$ означает, что ребро[7,8] $\{v_i, v_j\}$ в графе G отсутствует. Если $\{v_i, v_j\} \in E$, то элемент матрицы

$$a_{ij} = h(\{v_i, v_j\}),$$

является весом ребра $\{v_i, v_j\}$.

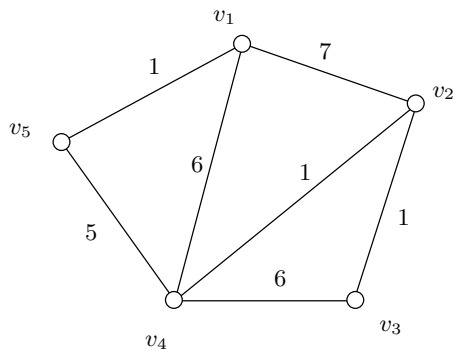


Рис. 2.2.

В этом случае мы можем использовать двоичное представление весов $a_{ij} = h(\{v_i, v_j\})$ и разделять их символом #. Для отметки конца строки в матрице смежности используется специальный символ #. Результирующая матрица смежности графа,

[7] В *хороших* книгах на русском языке для неориентированных графов вместо слова «дуга» применяется слово «ребро» – в отличие от английской литературы, где оба этих понятия обозначаются одним и тем же словом «edge». *(Прим. перев.)*

[8] Ребро между вершинами u и v будем обозначать записью $\{u, v\}$. А для дуги орграфа из u в v будем использовать обозначение (u, v).

изображённого на рис. 2.2, такова:

$$\begin{pmatrix} 0\,7\,0\,6\,1 \\ 7\,0\,1\,1\,0 \\ 0\,1\,0\,6\,0 \\ 6\,1\,6\,0\,5 \\ 1\,0\,0\,5\,0 \end{pmatrix}$$

Её представление в виде слова над алфавитом $\{0, 1, \#\}$ следующее:
0#111#0#110#1##111#0#1#1#0##0#1#0#110#0##110#1#110#
0#101##1#0#0#101#0.

Очевидно, что для матрицы смежности $M_G = [a_{ij}]$ любого неориентированного графа G выполнено равенство $a_{ij} = a_{ji}$ для каждой возможной пары i, j. Этот факт подразумевает избыточность в предложенном выше представлении графа G – поскольку информация о каждом ребре записана дважды. Поэтому достаточно рассмотреть только те элементы матрицы M_G, которые расположены выше главной диагонали. Итоговое представление G над алфавитом $\{0, 1, \#\}$ таково:

$$111\#0\#110\#1\#\#1\#1\#0\#\#110\#0\#\#101.$$

Последний рассматриваемый здесь пример – представление Булевой формулы с помощью операторов отрицания ($\neg$), дизъюнкции ($\vee$) и конъюнкции ($\wedge$). Далее в формулах мы будем обозначать Булевы переменные $x_1, x_2, x_3, \ldots$. Число возможных переменных бесконечно, поэтому мы не можем использовать символы $x_1, x_2, x_3, \ldots$ как символы алфавита. Вместо этого мы будем использовать алфавит

$$\Sigma_{\text{logic}} = \{0, 1, x, (,), \wedge, \vee, \neg\}$$

для кодировки Булевой переменной x_i как слова

$$xBin(i)$$

для всех $i \in \mathbb{N}$. Все остальные символы в формуле переписываются в её представлении один к одному. Таким образом, формула

$$(x_1 \vee x_7) \wedge \neg(x_{12}) \wedge (x_4 \vee x_8 \vee \neg(x_2))$$

имеет следующее представление:

$$(x1 \vee x111) \wedge \neg(x1100) \wedge (x100 \vee x1000 \vee \neg(x10)).$$

Полезная операция над словами – простая конкатенация двух слов.

Определение 2.9. *Пусть Σ – некоторый алфавит.* **Конкатенация** *относительно Σ – это отображение $K : \Sigma^* \times \Sigma^* \to \Sigma^*$ заданное в виде*

$$K(x, y) = x \cdot y = xy$$

для всех $x, y \in \Sigma^*$.

Пусть $x = 0aa1bb$ и $y = 111b$ для $\Sigma = \{0, 1, a, b\}$. Тогда $K(x, y) = x \cdot y = 0aa1bb111b$.

Замечание 2.10. Конкатенация K для Σ – ассоциативная операция над Σ^*, поскольку

$$K(u, K(v, w)) = u \cdot (v \cdot w) = uvw = (u \cdot v) \cdot w = K(K(u, v), w)$$

для всех $u, v, w \in \Sigma^*$. Кроме того, для каждого $x \in \Sigma^*$ выполнено следующее:

$$x \cdot \lambda = \lambda \cdot x = x.$$

Следовательно, (Σ^*, K, λ) – моноид с единицей λ.

Очевидно, конкатенация коммутативна только для алфавитов, состоящих из одной буквы.

Замечание 2.11. Для всех $x, y \in \Sigma^*$,

$$|xy| = |x \cdot y| = |x| + |y|.$$

В дальнейшем мы будем отдавать предпочтение записи xy – вместо $K(x, y)$ и $x \cdot y$.

Определение 2.12. *Пусть Σ – некоторый алфавит. Для каждого $x \in \Sigma^*$ и каждого натурального i определим i-ю итерацию x^i слова x как*

$$x^i = xx^{i-1},$$

где $x^0 = \lambda$.

Так, например, $K(aabba, aaaaa) = aabbaaaaaa = a^2b^2a^6 = a^2b^2(aa)^3$. Поэтому введённое обозначение позволяет нам находить более короткое представление некоторых слов.

Теперь определим подслова слова x как связанные части этого слова.

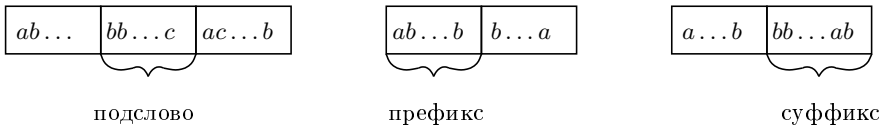


Рис. 2.3.

Определение 2.13. *Пусть $v, w \in \Sigma^*$ для некоторого алфавита Σ. Тогда:*

- v – **подслово** *слова* $w \Leftrightarrow \exists x, y \in \Sigma^* : w = xvy$.
- v – **суффикс** *слова* $w \Leftrightarrow \exists x \in \Sigma^* : w = xv$.
- v – **префикс** *слова* $w \Leftrightarrow \exists y \in \Sigma^* : w = vy$.
- $v \neq \lambda$ – **собственное** *подслово* [*собственный суффикс, собственный префикс*] *слова w тогда и только тогда, когда $v \neq w$, а v – подслово* [*суффикс, префикс*] *слова w* (*рис.2.3*).

Поскольку $(abc)^3 = abcabcabc$, слово abc – собственный префикс слова $(abc)^3$. Слово bc – собственный суффикс слова $(abc)^3$.

Упражнение 2.14. Пусть Σ – некоторый алфавит, и пусть $x \in \Sigma^*$, где $|x| = n$ для некоторого $n \in \mathbb{N}$. Чему равно максимально возможное число различных подслов слова x? Посчитайте все различные подслова слова *abbcbbab*.

Определение 2.15. *Пусть $x \in \Sigma^*$, и пусть $a \in \Sigma$ для некоторого алфавита Σ. Определим $|x|_a$ как число вхождений символа a в x.*
Для каждого множества A запись $\boldsymbol{|A|}$ обозначает мощность множества A. $\boldsymbol{\mathcal{P}(A)} = \{S \mid S \subseteq A\}$ – множество всех подмножеств множества A.

Следовательно, $|(abbab)|_a = 2$ и $|(11bb0)|_0 = 1$. Для каждого $x \in \Sigma^*$

$$|x| = \sum_{a \in \Sigma} |x|_a.$$

В отдельных главах книги нам понадобится отношение порядка[9] на множестве всех слов над заданным алфавитом. Удобнее всего рассматривать т. н. канонический порядок, определяемый следующим образом.

Определение 2.16. *Пусть $\Sigma = \{s_1, s_2, \ldots, s_m\}$, где $m \geq 1$, – некоторый алфавит, и пусть $s_1 < s_2 < \cdots < s_m$ – некоторый линейный порядок, заданный на элементах Σ. Определим* **канонический порядок**, *заданный на элементах Σ^* (т. е. для всех пар $u, v \in \Sigma^*$), следующим образом:*

$$\begin{aligned} u < v \Leftrightarrow \quad & (|u| < |v|) \\ & \vee\, (|u| = |v| \wedge u = x \cdot s_i \cdot u' \wedge v = x \cdot s_j \cdot v' \\ & \quad \text{для некоторых } x, u', v' \in \Sigma^* \text{ и } i < j). \end{aligned}$$

Теперь определим язык как некоторое множество слов над заданным алфавитом.

Определение 2.17. Язык *над алфавитом Σ – некоторое подмножество Σ^*. Дополнение $\boldsymbol{L^C}$ языка L относительно Σ – это язык $\Sigma^* - L$.*
$\boldsymbol{L_\emptyset} = \emptyset$ – **пустой язык**.
$\boldsymbol{L_\lambda} = \{\lambda\}$ – *язык, содержащий единственное слово – пустое слово λ.*
Пусть L_1 и L_2 – языки над алфавитом Σ. Тогда

$$\boldsymbol{L_1 \cdot L_2} = \boldsymbol{L_1 L_2} = \{vw \mid v \in L_1,\ w \in L_2\}$$

– конкатенация языков L_1 и L_2. Пусть L – некоторый язык над алфавитом Σ. Определим

$$\begin{aligned} & \boldsymbol{L^0} := L_\lambda, \\ & \boldsymbol{L^{i+1}} = L^i \cdot L \text{ для всех } i \in \mathbb{N}_0, \\ & \boldsymbol{L^*} = \bigcup_{i \in \mathbb{N}_0} L^i, \\ & \boldsymbol{L^+} = \bigcup_{i \in \mathbb{N}} L^i = L \cdot L^*. \end{aligned}$$

L^* *называется* **звездой Клини** *языка L.*

[9] Общее определение отношения линейного порядка, заданного на элементах некоторого множества A, таково. Объект $a \in A$ предшествует объекту $b \in A$, если и только если $f(a) < f(b)$.

Следующие множества – примеры языков над алфавитом $\Sigma = \{a, b\}$:

$$\begin{aligned}
L_1 &= \emptyset, \\
L_2 &= \{\lambda\}, \\
L_3 &= \{\lambda, ab, abab\}, \\
L_4 &= \Sigma^* = \{\lambda, a, b, aa, \ldots\}, \\
L_5 &= \Sigma^+ = \{a, b, aa, \ldots\}, \\
L_6 &= \{a\}^* = \{\lambda, a, aa, aaa, \ldots\} = \{a^i \mid i \in \mathbb{N}_0\}, \\
L_7 &= \{a^p \mid p - \text{простое число}\}, \\
L_8 &= \{a^i b^{2 \cdot i} a^i \mid i \in \mathbb{N}_0\}, \\
L_9 &= \Sigma, \\
L_{10} &= \Sigma^3 = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}.
\end{aligned}$$

Множество всех грамматически правильных английских текстов – некоторый язык над алфавитом Σ_{keyboard}; множество всех синтаксически правильных программ на языке Ява – тоже язык над алфавитом Σ_{keyboard}.

Отметим, что

$$\Sigma^i = \{x \in \Sigma^* \mid |x| = i\},$$

кроме того,

$$L_\emptyset L = L_\emptyset = \emptyset$$

и

$$L_\lambda \cdot L = L.$$

Упражнение 2.18. Пусть $L_1 = \{\lambda, ab, b^3a^4\}$, $L_2 = \{ab, b, ab^2, b^4\}$. Какие слова принадлежат языку L_1L_2?

Наша следующая цель – практическая работа с языками. Поскольку языки суть множества, для них можно использовать стандартные операции объединения ($\cup$) и пересечения множеств ($\cap$). К этому набору операций добавим конкатенацию и звезду Клини. Первый вопрос, который мы рассмотрим – выполняются ли законы дистрибутивности относительно объединения и конкатенации, а также относительно пересечения и конкатенации. Следующая лемма даёт положительный ответ для объединения и конкатенации. Для доказательства равенства двух множеств A и B мы будем использовать стандартные методы теории множеств – где обычно отдельно доказываются условия $A \subseteq B$ и $B \subseteq A$, а из этого выводится равенство $A = B$. Чтобы показать выполнение условия $A \subseteq B$, достаточно для каждого элемента $x \in A$ доказать, что x принадлежит B ($x \in B$).

Лемма 2.19. *Пусть L_1, L_2 и L_3 – языки над алфавитом Σ. Тогда*

$$L_1L_2 \cup L_1L_3 = L_1(L_2 \cup L_3).$$

Доказательство. Во-первых, покажем выполнение условия $L_1L_2 \cup L_1L_3 \subseteq L_1(L_2 \cup L_3)$. Комментарии в фигурных скобках поясняют шаги доказательства.

Условие $L_1L_2 \subseteq L_1(L_2 \cup L_3)$ выполняется, поскольку

$$\begin{aligned}
L_1L_2 &= \{xy \mid x \in L_1 \wedge y \in L_2\} \quad \{\text{определение конкатенации}\} \\
&\subseteq \{xy \mid x \in L_1 \wedge y \in L_2 \cup L_3\} \quad \{\text{поскольку } L_2 \subseteq L_2 \cup L_3\} \\
&= L_1 \cdot (L_2 \cup L_3) \quad \{\text{определение конкатенации}\}.
\end{aligned}$$

Тем же способом мы показываем, что $L_1L_3 \subseteq L_1(L_2 \cup L_3)$. Поэтому $L_1L_2 \cup L_1L_3 \subseteq L_1(L_2 \cup L_3)$.

Теперь докажем включение $L_1(L_2 \cup L_3) \subseteq L_1L_2 \cup L_1L_3$.

Пусть $x \in L_1(L_2 \cup L_3)$. Тогда

$$
\begin{aligned}
& x \in \{yz \mid y \in L_1 \wedge z \in L_2 \cup L_3\} \\
& \quad \{\text{определение конкатенации}\} \\
\Rightarrow\ & \exists y \in L_1 \wedge \exists z \in L_2 \cup L_3, \text{ такое, что } x = yz \\
\Rightarrow\ & \exists y \in L_1 \wedge (\exists z \in L_2 \vee \exists z \in L_3), \text{ такое, что } x = yz \\
& \quad \{\text{определение } \cup\} \\
\Leftrightarrow\ & (\exists y \in L_1 \wedge \exists z \in L_2 : x = yz) \vee (\exists y \in L_1 \wedge \exists z \in L_3 : x = yz) \\
& \quad \{\text{дистрибутивный закон для } \wedge, \vee\} \\
\Leftrightarrow\ & (x \in \underbrace{\{yz \mid y \in L_1 \wedge z \in L_2\}}_{L_1L_2}) \vee (x \in \underbrace{\{yz \mid y \in L_1 \wedge z \in L_3\}}_{L_1L_3}) \\
& \quad \{\text{определение конкатенации}\} \\
\Leftrightarrow\ & x \in L_1L_2 \cup L_1L_3 \quad \{\text{определение } \cup\}.
\end{aligned}
$$

□

Теперь рассмотрим вопрос о выполнении дистрибутивного закона для конкатенации и пересечения. Ответ на этот вопрос отрицательный – и на первый взгляд этот факт может показаться неожиданным. Вместо дистрибутивного закона выполняется только соответствующее включение – и мы покажем это в следующей лемме.

Лемма 2.20. *Пусть L_1, L_2 и L_3 – языки над алфавитом Σ. Тогда*

$$L_1(L_2 \cap L_3) \subseteq L_1L_2 \cap L_1L_3.$$

Доказательство. Пусть $x \in L_1(L_2 \cap L_3)$. Это равносильно следующему:

$$
\begin{aligned}
& x \in \{yz \mid y \in L_1 \wedge z \in L_2 \cap L_3\} \\
& \quad \{\text{определение конкатенации}\} \\
\Leftrightarrow\ & \exists y, z \in \Sigma^*, y \in L_1 \wedge (z \in L_2 \wedge z \in L_3) \text{ такие, что } x = yz \\
& \quad \{\text{определение } \cap\} \\
\Leftrightarrow\ & \exists y, z \in \Sigma^*, (y \in L_1 \wedge z \in L_2) \wedge (y \in L_1 \wedge z \in L_3) : x = yz \\
\Rightarrow\ & \exists y, z \in \Sigma^*, (yz \in L_1L_2) \wedge (yz \in L_1L_3) : x = yz \\
& \quad \{\text{определение конкатенации}\} \\
\Leftrightarrow\ & x \in L_1L_2 \cap L_1L_3 \quad \{\text{определение } \cap\}.
\end{aligned}
$$

□

Для того, чтобы показать, что условие $L_1(L_2 \cap L_3) \supseteq L_1L_2 \cap L_1L_3$ выполняется не всегда, достаточно указать три конкретных языка U_1, U_2 и U_3, таких что

$$U_1(U_2 \cap U_3) \subset U_1U_2 \cap U_1U_3.$$

При поиске подходящих языков U_1, U_2 и U_3 можно взять за основу тот факт, что в доказательстве леммы 2.20 имеется только одно значение, которое не может быть преобразовано (заменено эквивалентным). Если некоторое слово x принадлежит как L_1L_2, так и L_1L_3, то можно вообще не прийти к заключению, что $x = yz$ – при $x \in L_1$ и $z \in L_2 \cap L_3$. Это происходит, например, когда слово x может быть записано в виде

$$x = y_1z_1 = y_2z_2 \text{ для } y_1 \neq y_2 \text{ (т. е. } z_1 \neq z_2),$$

где $y_1, y_2 \in L_1$, $z_1 \in L_2$ и $z_2 \in L_3$. В этом случае $x \in L_1L_2 \cap L_1L_3$, но вовсе не очевидно, что слово x должно принадлежать языку $L_1(L_2 \cap L_3)$.

Лемма 2.21. *Существуют U_1, U_2, $U_3 \in (\Sigma_{\text{bool}})^*$, такие что*

$$U_1(U_2 \cap U_3) \subsetneq U_1U_2 \cap U_1U_3.$$

Доказательство. Выберем $U_2 = \{0\}$ и $U_3 = \{10\}$. Отсюда мы получаем $U_2 \cap U_3 = \emptyset$, и поэтому

$$U_1(U_2 \cap U_3) = \emptyset$$

для любого языка U_1. Теперь достаточно найти некоторый язык U_1, такой, чтобы язык $U_1U_2 \cap U_1U_3$ был непустым. Положим $U_1 = \{\lambda, 1\}$. Тогда

$$U_1U_2 = \{0, 10\}, U_1U_3 = \{10, 110\},$$

и следовательно

$$U_1U_2 \cap U_1U_3 = \{10\} \neq \emptyset.$$

□

Упражнение 2.22. Пусть L_1, L_2 и L_3 – языки над алфавитом $\{0\}$. Всегда ли выполняется равенство

$$L_1(L_2 \cap L_3) = L_1L_2 \cap L_1L_3 \ ?$$

Упражнение 2.23. Пусть $L_1 \subseteq \Sigma_1^*$, $L_2, L_3 \subset \Sigma_2^*$ для некоторых алфавитов Σ_1 и Σ_2, таких что $\Sigma_1 \cap \Sigma_2 = \emptyset$. Всегда ли выполняется равенство

$$L_1(L_2 \cap L_3) = L_1L_2 \cap L_1L_3 \ ?$$

Упражнение 2.24. Существуют ли языки L_1, L_2 и L_3, такие что язык $L_1(L_2 \cap L_3)$ конечен, а язык $L_1L_2 \cap L_1L_3$ бесконечен?

Далее будем работать со звездой Клини.

Лемма 2.25. *Выполняется следующее равенство:*

$$\{a\}^*\{b\}^* = \{a^ib^j \mid i, j \in \mathbb{N}_0\}.$$

Доказательство. Во-первых, покажем, что $\{a\}^*\{b\}^* \subseteq \{a^ib^j \mid i, j \in \mathbb{N}_0\}$.

Пусть $x \in \{a\}^*\{b\}^*$. Тогда:

$$
\begin{aligned}
& x = yz, \text{ где } y \in \{a\}^* \wedge z \in \{b\}^* \\
& \quad \{\text{определение конкатенации}\} \\
\Rightarrow\ & x = yz, \text{ где } (\exists k \in \mathbb{N}_0 : y \in \{a\}^k) \wedge (\exists m \in \mathbb{N}_0 : z \in \{b\}^m) \\
& \quad \{\text{определение звезды Клини}\} \\
\Leftrightarrow\ & x = yz, \text{ где } (\exists k \in \mathbb{N}_0 : y = a^k) \wedge (\exists m \in \mathbb{N}_0 : z = b^m) \\
\Leftrightarrow\ & \exists k, m \in \mathbb{N}_0, \text{ такие, что } x = a^k b^m \\
\Rightarrow\ & x \in \{a^i b^j \mid i, j \in \mathbb{N}_0\}.
\end{aligned}
$$

Далее покажем, что $\{a^i b^j \mid i, j \in \mathbb{N}_0\} \subseteq \{a\}^*\{b\}^*$.
Пусть $x \in \{a^i b^j \mid i, j \in \mathbb{N}_0\}$. Тогда:

$$
\begin{aligned}
& x = a^r b^l \text{ для некоторых целых } r, l \in \mathbb{N}_0 \\
\Rightarrow\ & x \in \{a\}^*\{b\}^*, \text{ поскольку } a^r \in \{a\}^*, b^l \in \{b\}^*.
\end{aligned}
$$

□

Упражнение 2.26. Докажите или опровергните равенство

$$(\{a\}^*\{b\}^*)^* = \{a, b\}^*.$$

Определение 2.27. *Пусть Σ_1 и Σ_2 – два произвольных алфавита.* **Гомоморфизм**[10] *из Σ_1 в Σ_2 – произвольная функция $h : \Sigma_1^* \to \Sigma_2^*$, удовлетворяющая следующим условиям:*

- $h(\lambda) = \lambda$;
- $h(uv) = h(u) \cdot h(v)$ *для всех* $u, v \in \Sigma_1^*$.

Очевидно, что для задания гомоморфизма достаточно определить значение $h(a)$ для каждого символа $a \in \Sigma_1$.

Упражнение 2.28. Пусть h – некоторый гомоморфизм из Σ_1 в Σ_2. Докажите по индукции, что для всех слов $x = x_1 x_2 \ldots x_m$, $x_i \in \Sigma_1$ для $i = 1, \ldots, m$, выполнено равенство

$$h(x) = h(x_1)h(x_2)\ldots h(x_m).$$

Рассмотрим отображение h, заданное следующим образом:

$$h(\#) = 10, h(0) = 00 \text{ и } h(1) = 11.$$

Очевидно, что h определяет гомоморфизм из $\{0, 1, \#\}$ в Σ_{bool}. Например,

$$
\begin{aligned}
h(011\#101\#) &= h(0)h(1)h(1)h(\#)h(1)h(0)h(1)h(\#) \\
&= 0011111011001110.
\end{aligned}
$$

Можно использовать h для перевода представления некоторых объектов над алфавитом $\{0, 1, \#\}$ в новое представление тех же самых объектов над алфавитом Σ_{bool}.

[10] По терминологии книги *[А.Саломаа. «Жемчужины теории формальных языков», М., Мир, 1986]* – «морфизм». Мы опускаем алгебраические и лингвистические аспекты различия этих названий. *(Прим. перев.)*

Упражнение 2.29. Определите гомоморфизм из $\{0,1,\#\}$ в Σ_{bool}, который отображает бесконечно много слов над алфавитом $\{0,1,\#\}$ в одно слово из $(\Sigma_{\text{bool}})^*$.

Упражнение 2.30. Определите инъективный гомоморфизм из Σ_{logic} в Σ_{bool}, обеспечивающий однозначное представление Булевых формул над алфавитом Σ_{bool}.

Упражнение 2.31. Пусть Σ_1 и Σ_2 – некоторые алфавиты, а h – некоторый гомоморфизм из Σ_1 в Σ_2. Для каждого языка $L \subseteq \Sigma_1^*$ определим

$$\mathbf{h(L)} = \{h(w) \mid w \in L\}.$$

Пусть $L_1, L_2 \subseteq \Sigma_1^*$. Докажите или опровергните следующее равенство:

$$h(L_1)h(L_2) = h(L_1L_2).$$

2.3 Алгоритмические проблемы

До введения в главе 4 формального определения понятия «алгоритм» – с использованием модели машины Тьюринга – мы рассматриваем алгоритмы как программы; предполагаем, что читатель знает, что это такое. Для наших целей безразличен выбор конкретного языка программирования. Используя синоним «программа» для слова «алгоритм», мы требуем, чтобы программа вычисляла правильный выход для каждого возможного входа. Таким образом, алгоритм может рассматриваться как программа, заканчивающая работу для любого входа (то есть не имеющая бесконечных вычислений) и решающая данную проблему. Согласно этому предположению, любая программа (алгоритм) A выполняет отображение

$$A : \Sigma_1^* \to \Sigma_2^*$$

для некоторых алфавитов Σ_1 и Σ_2. Это означает, что:

- входы представлены как слова над алфавитом Σ_1;
- выходы представлены как слова над алфавитом Σ_2;
- A однозначно определяет выход по каждому входу.

Для некоторых алгоритма A и входа x обозначим записью $\boldsymbol{A(x)}$ выход алгоритма A для этого входа. Будем говорить, что два алгоритма (программы) A и B **эквивалентны**, если они работают над одним и тем же алфавитом Σ, и при этом $A(x) = B(x)$ для всех $x \in \Sigma^*$.

Определение 2.32. *Для заданных алфавита Σ и языка $L \subseteq \Sigma^*$* **проблема принадлежности $(\boldsymbol{\Sigma, L})$** *заключается в том, что для любого $x \in \Sigma^*$ необходимо ответить на вопрос, какое именно условие из следующих двух выполнено:*

$$x \in L \quad \textit{или} \quad x \notin L.$$

Алгоритм A **решает** *проблему принадлежности (L, Σ), если для всех $x \in \Sigma^*$ выполнено следующее:*

$$A(x) = \begin{cases} 1, \textit{ если } & x \in L. \\ 0, \textit{ если } & x \notin L. \end{cases}$$

Будем при этом также говорить, что A **распознаёт** *язык L.*

Если для некоторого языка L существует алгоритм, который распознаёт L, то мы будем говорить, что язык L является **рекурсивным**[11]. Мы часто будем использовать язык $L \subseteq \Sigma^*$ для определения конкретных свойств слов некоторого подмножества множества Σ^* – или множества объектов, представленных этими словами. При этом все слова языка L удовлетворяет этому свойству, а слова из $L^{\complement} = \Sigma^* - L$ – нет.

Обычно проблема принадлежности (Σ, L) описывается таким образом:

$\underline{(\Sigma, L)}$
Вход: $x \in \Sigma^*$.
Выход: $A(x) \in \Sigma_{\text{bool}} = \{0, 1\}$, где

$$A(x) = \begin{cases} 1, \text{ если } & x \in L \text{ (да)}, \\ 0, \text{ если } & x \notin L \text{ (нет)}. \end{cases}$$

Например, $(\{a, b\}, \{a^n b^n \mid n \in \mathbb{N}_0\})$ – проблема принадлежности, которая может быть также определена следующим образом:

$\underline{(\{a, b\}, \{a^n b^n \mid n \in \mathbb{N}_0\})}$
Вход: $x \in \{a, b\}^*$.
Выход: Да – если $x = a^n b^n$ для некоторого $n \in \mathbb{N}_0$.
Нет – в противном случае.

Одна из известных проблем принадлежности, имеющая большое практическое значение – это **проверка числа на простоту**:

$$(\Sigma_{\text{bool}}, \{x \in (\Sigma_{\text{bool}})^* \mid \mathit{Number}(x) - \text{некоторое простое число}\}).$$

Обычная формулировка этой задачи такова:

$\underline{(\Sigma_{\text{bool}}, \{x \in (\Sigma_{\text{bool}})^* \mid \mathit{Number}(x) - \text{простое число}\})}$
Вход: $x \in (\Sigma_{\text{bool}})^*$.
Выход: Да, если $\mathit{Number}(x)$ – простое число.
Нет – в противном случае.

Пусть $L = \{x \in (\Sigma_{\text{keyboard}})^* \mid x - \text{синтаксически корректная программа на Си++}\}$. Рассмотрим следующую задачу – вспомогательную для любого компилятора этого языка.

Вход: $x \in (\Sigma_{\text{keyboard}})^*$.
Выход: Да, если $x \in L$.
Нет – в противном случае.

Проблема существования Гамильтонова цикла (HC)[12] – это проблема (Σ, HC), где $\Sigma = \{0, 1, \#\}$, а

[11] Рекурсия – одно из фундаментальных понятий информатики. Поэтому в дальнейшем для в точного определения этого понятия будем использовать формальную модель вычисления (алгоритма).

[12] В подобных случаях мы будем употреблять английские аббревиатуры – как более устоявшиеся. Отметим также, что названия самих проблем обычно будут совпадать с названиями соответствующих им языков. *(Прим. перев.)*

$$\text{HC} = \{x \in \Sigma^* \mid x \text{ представляет собой неориентированный граф, который содержит Гамильтонов цикл}^{13}\}.$$

Проблема выполнимости (SAT) – это $(\Sigma_{\text{logic}}, \text{SAT})$, где

$$\text{SAT} = \{x \in (\Sigma_{\text{logic}})^* \mid x \text{ – некоторая выполнимая Булева формула}\}.$$

Важный подкласс проблем принадлежности составляют проблемы эквивалентности. Например, проблема эквивалентности для программ состоит в определении того, эквивалентны ли две заданные программы A и B, написанные на одном и том же языке программирования. Другой пример проблемы эквивалентности – определение, представляют ли две данные Булевы формулы одну и ту же Булеву функцию.

Определение 2.33. *Пусть Σ и Γ – два алфавита. Будем говорить, что алгоритм A* **вычисляет функцию $f : \Sigma^* \to \Gamma^*$**, *если для всех $x \in \Sigma^*$*

$$A(x) = f(x).$$

Итак, проблемы принадлежности можно рассматривать как специальный случай вычисления функций – потому что решение некоторой конкретной проблемы принадлежности эквивалентно вычислению характеристической функции языка.[14] И на первый взгляд может показаться, что вычисление функций – это самое общее представление алгоритмических проблем. Но следующее определение показывает, что это не так.

Определение 2.34. *Пусть Σ и Γ – некоторые алфавиты, а $R \subseteq \Sigma^* \times \Gamma^*$ – некоторое бинарное отношение. Алгоритм A* **вычисляет R** *(или* **решает проблему отношения R***), если для каждого $x \in \Sigma^*$ выполнено следующее:*

$$(x, A(x)) \in R.$$

Согласно определению 2.34 мы видим, что для решения проблемы вычисления отношения R с заданным входом x достаточно найти одно значение y (из некоторого множества потенциальных значений, возможно, бесконечного), такое что $(x, y) \in R$. Следующие примеры показывают, что проблемы, связанные с вычислением отношений – это не только абстрактное обобщение вычисления функций[15]; многие проблемы практического программирования можно рассматривать именно как проблемы вычисления отношений.

Пусть $R_{\text{fac}} \subseteq (\Sigma_{\text{bool}})^* \times (\Sigma_{\text{bool}})^*$, где $(x, y) \in R_{\text{fac}}$ тогда и только тогда, когда:

- либо $Number(y)$ – делитель[16] числа $Number(x)$;

[13] Гамильтонов цикл графа – это цикл (замкнутый путь), который содержит каждую вершину этого графа в точности один раз.

[14] Характеристическая функция f_L языка $L \subseteq \Sigma^*$ – функция, действующая из Σ^* в $\{0, 1\}$, такая что $f_L(x) = 1$ тогда и только тогда, когда $x \in L$.

[15] Напомним, что функции можно рассматривать как отношения, обладающие следующим свойством: для каждого значения x существует в точности одно значение y, такое что $(x, y) \in R$.

[16] Некоторое натуральное a является делителем целого b, если b делится на a нацело, а также $a \notin \{1, b\}$.

- либо $y = 1$ – в том случае, когда $Number(x)$ – простое число.

Строгая формулировка этой проблемы такова.

$\underline{R_{\text{fac}}}$
Вход: $x \in (\Sigma_{\text{bool}})^*$.
Выход: $y \in (\Sigma_{\text{bool}})^*$, причём:
$y = 1$ если $Number(x)$ – простое;
$Number(y)$ – делитель числа $Number(x)$, если x – составное.

Другим примером сложной проблемы является поиск доказательства некоторой теоремы. Пусть $R_{\text{Proof}} \subseteq (\Sigma_{\text{keyboard}})^* \times (\Sigma_{\text{keyboard}})^*$; при этом $(x, y) \in R_{\text{Proof}}$ если:

- либо x является кодом истинного утверждения в рассматриваемой математической теории, а y является представлением доказательства этого утверждения;
- либо $y = \sqcup$, а x не является кодом истинного утверждения в рассматриваемой теории.

Далее рассмотрим оптимизационные проблемы, которые можно интерпретировать как специальные случаи (варианты) проблемы отношения. Оптимизационные проблемы представляют большой интерес – как с теоретической, так и с практической точки зрения. Однако для формулировки оптимизационных проблем мы вместо приведённой выше формулировки проблемы отношения будем используовать несколько иной подход; вначале приведём его неформальное описание.

Частный случай[17] x оптимизационной проблемы определяет множество $\mathcal{M}(x)$ допустимых решений для этого входа x. Таким образом, мы можем сказать, что имеется отношение R, для которого

$$(x, y) \in R \text{ если и только если } y - \text{допустимое решение для } x.$$

Но в данном случае мы должны решить не только проблему отношения R. Вход x определяет ещё и стоимость каждого y, принадлежащего множеству $\mathcal{M}(x)$. А выход должен быть одним из допустимых решений с наиболее благоприятной (максимальной или минимальной) стоимостью.

Рассмотрим формальное описание оптимизационной проблемы.

Определение 2.35. Оптимизационная проблема – *это шестёрка* $\mathcal{U} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, cost, goal)$, *где:*

- Σ_I – *некоторый алфавит, называемый* **входным алфавитом**;
- Σ_O – *некоторый алфавит, называемый* **выходным алфавитом**;
- $L \subseteq \Sigma_I^*$ – *язык* **подходящих входов** (**частных случаев**; *в качестве входов мы допускаем только слова, имеющие приемлемую интерпретацию). Некоторое слово* $x \in L$ *называется* **частным случаем проблемы** $\mathcal{U}$;
- $\mathcal{M}$ – *функция, действующая из* L *в* $\mathcal{P}(\Sigma_O^*)$, *где для каждого* $x \in L$ *множество* $\mathcal{M}(x)$ *есть множество* **допустимых решений для** x;
- $cost$ – *функция вида* $cost : \bigcup_{x \in L}(\mathcal{M}(x) \times \{x\}) \to \mathbb{R}^+$, *называемая* **функцией стоимости**;
- $goal \in \{minimum, maximum\}$ – *цель*.

[17] В русской литературе иногда употребляются и другие термины. Подробнее см. ниже. *(Прим. перев.)*

Некоторое допустимое решение $\alpha \in \mathcal{M}(x)$ *называется* **оптимальным** *для* x *– частного случая проблемы* U*, если*

$$cost(\alpha, x) = \mathbf{Opt_{\mathcal{U}}(x)} = goal\{cost(\beta, x) \mid \beta \in \mathcal{M}(x)\}.$$

Будем говорить, что некоторый алгоритм A **решает** *проблему* $\mathcal{U}$*, если для произвольного* $x \in L$*:*

- $A(x) \in \mathcal{M}(x)$ $\big(A(x)$ *– некоторое допустимое решение для* x *– частного случая проблемы* $\mathcal{U}\big)$,
- $cost(A(x), x) = goal\{cost(\beta, x) \mid \beta \in \mathcal{M}(x)\}$.

Если $goal = minimum$*, то* $\mathcal{U}$ *называется* **минимизационной проблемой**; *а если* $goal = maximum$*, то* $\mathcal{U}$ *называется* **максимизационной проблемой**.

Чтобы понять, почему мы определяем оптимизационные проблемы именно таким образом, приведём некоторые комментарии к элементам шестёрки $\mathcal{U}$. Входной алфавит Σ_I имеет тот же самый смысл, что и входной алфавит для проблем принадлежности – т. е. используется для представления входа проблемы $\mathcal{U}$. Аналогично, выходной алфавит Σ_O используется для представления выходов (допустимых решений). Язык $L \subseteq \Sigma_I^*$ – множество корректных представлений частных случаев проблемы; при этом мы предполагаем, что никакое слово языка $L^{\complement} = \Sigma_I^* - L$ входом являться не может. Последний факт очень важен: с точки зрения сложности он означает, что мы сосредотачиваемся именно на само́й оптимизации – а не на решении проблемы принадлежности (Σ_I, L).

Пусть x – некоторый частный случай (вход) проблемы. Он обычно определяет ряд ограничений, а $\mathcal{M}(x)$ – множество объектов (допустимых решений для x), которые удовлетворяют этим ограничениям. Вход x также определяет стоимость $cost(\alpha, x)$ каждого решения $\alpha \in \mathcal{M}(x)$. И задача состоит в том, чтобы найти оптимальное решение во множестве $\mathcal{M}(x)$ допустимых решений входа x. Обычно трудность решения проблемы $\mathcal{U}$ заключается в том, что множество $\mathcal{M}(x)$ имеют столь большое число элементов, что практически невозможно эффективно сгенерировать все допустимые решения $\mathcal{M}(x)$ для выбора лучшего из них.

Для более ясного описания оптимизационной проблемы будем часто опускать рассмотрение отдельных частей этого описания – а именно, алфавитов Σ_I и Σ_O, а также способа кодирования данных над этими алфавитами. Мы просто предполагаем, что обычные данные – вроде целых чисел, графов, формул – представлены так, как описано выше. Это несколько упрощает ситуацию, поскольку мы можем теперь «непосредственно обращаться» к этим объектам – вместо работы с их формальным представлением. Поэтому для описания оптимизационной проблемы обычно достаточно следующих элементов шестёрки:

- множества L частных случаев проблемы;
- ограничений для любого частного случая проблемы $x \in L$ и соответствующих множеств $\mathcal{M}(x)$;
- функции стоимости;
- цели.

Рассмотрим пример – **задачу коммивояжёра** (TSP, ЗКВ)[18].

[18] Для этой проблемы русская аббревиатура тоже является устоявшейся.

TSP

Вход: Взвешенный полный граф (G, c), где $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$ для некоторого $n \in \mathbb{N}$, а также функция $c : E \to \mathbb{N}$.

{Вход представляет собой некоторое слово $x \in \{0, 1, \#\}^*$, которое кодирует (представляет) полный взвешенный граф (G, c).}

Ограничения: Для каждого частного случая проблемы (G, c) множество $\mathcal{M}(G, c)$ включает все Гамильтоновы циклы графа G. Каждый Гамильтонов цикл может быть представлен в виде последовательности вершин $v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1}$,[19] где $(i_1, i_2, \ldots, i_n)$ – некоторая перестановка множества $(1, 2, \ldots, n)$.

{Строгое формальное представление $\mathcal{M}(G, c)$ следующее. Это множество всех слов $y_1 \# y_2 \# \ldots \# y_n \in \{0, 1, \#\}^* = \Sigma_O^*$, таких что $y_i \in \{0, 1\}^+$ для $i = 1, 2, \ldots, n$, причём

$$\{Number(y_1), Number(y_2), \ldots, Number(y_n)\} = \{1, 2, \ldots, n\},$$

а $Number(y_1) = 1$.}

Стоимость: Для Гамильтонова цикла $H = v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1} \in \mathcal{M}(G, c)$ полагаем

$$cost((v_{i_1}, \ldots, v_{i_n}, v_{i_1}), (G, c)) = \sum_{j=1}^{n} c\left(\{v_{i_j}, v_{i_{(j \mod n)+1}}\}\right),$$

т. е. стоимость каждого Гамильтонова цикла равна сумме весов всех его рёбер.

Цель: *minimum*.

Для частного случая проблемы TSP, приведённого на рис. 2.4, мы получаем

$$cost\big((v_1, v_2, v_3, v_4, v_5, v_1), (G, c)\big) = 8 + 1 + 7 + 2 + 1 = 19,$$
$$cost\big((v_1, v_5, v_3, v_2, v_4, v_1), (G, c)\big) = 1 + 1 + 1 + 1 + 1 = 5.$$

Гамильтонов цикл $v_1, v_5, v_3, v_2, v_4, v_1$ – единственное оптимальное решение рассматриваемого частного случая проблемы TSP.

TSP является примером труднорешаемой оптимизационной проблемы. Но во многих приложениях нас интересуют только те частные случаи проблемы, которые имеют некоторые специальные «хорошие» свойства – облегчающие поиск оптимального решения. Будем говорить, что оптимизационная проблема $\mathcal{U}_1 = (\Sigma_I, \Sigma_O, L', \mathcal{M}, cost, goal)$ является **вариантом проблемы**[20] $\mathcal{U}_2 = (\Sigma_I, \Sigma_O, L, \mathcal{M}, cost, goal)$, если $L' \subseteq L$ – т. е. если проблема $\mathcal{U}_1$ может быть получена из $\mathcal{U}_2$ путём установления специальных ограничений на множество подходящих (разрешённых) входов.

[19] Заметим, что этот формализм допускает несколько различных правильных вариантов представления любого конкретного Гамильтонова цикла.

[20] Здесь и всюду далее применяется следующая терминология – по-видимому, она достаточно удачна. «Вариант» проблемы определяется некоторым подмножеством всех возможных входов (аналог – знак $\subseteq$). Иногда применяемые в русской литературе альтернативные термины для данного понятия – фактически т. н. кальки, «подпроблема» и «подзадача», – представляются менее удачными; в литературе на русском языке они чаще употребляются в несколько ином смысле.

А «частный случай» проблемы – это один конкретный возможный вход (аналог – знак $\in$). В оригинале для «варианта проблемы» и «частного случая проблемы» применяются термины «subproblem» и «problem instance» соответственно. *(Прим. перев.)*

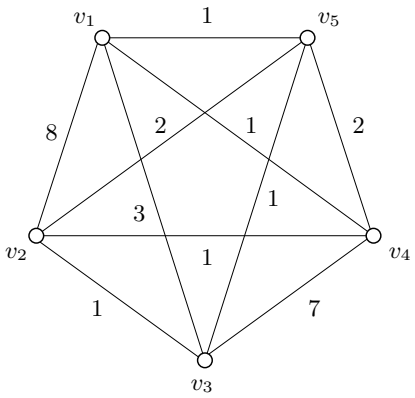


Рис. 2.4.

Например, мы определяем т. н. **метрическую TSP** (**Δ-TSP**) как вариант проблемы TSP. Это означает, что ограничения, стоимость и цель Δ-TSP остаются такими же, как и в TSP вообще – только множество возможных входов специально сокращено до некоторого своего собственного помножества. А именно, для Δ-TSP мы требуем, чтобы каждый возможный вход (G, c) удовлетворял бы т. н. **неравенству треугольника** –

$$c(\{u, v\}) \leq c(\{u, w\}) + c(\{w, v\})$$

для всех троек вершин u, v, w графа G. Неравенство треугольника можно рассматривать как естественное свойство частных случаев проблемы – поскольку оно утверждает, что прямой путь между вершинами u и v не может быть более дорогим, чем любой другой путь между этими же вершинами. Таким образом, если вершины графа представляют собой города, рёбра представляют связи (дороги) между этими городами, а стоимости соответствуют расстояниям, то неравенство треугольника – естественное ограничение реальной модели. Заметим, что приведённый на рис.2.4 частный случай проблемы неравенству треугольника не удовлетворяет.

Упражнение 2.36. Докажите, что при $n > 2$ для любого графа G с n вершинами выполнено равенство $|\mathcal{M}((G, c))| = (n-1)!/2$.

Вершинное покрытие графа $G = (V, E)$ – это некоторое множество U его вершин ($U \subseteq E$), такое что каждое ребро множества E инцидентно[21] по крайней мере одной вершине множества U. Например, множество $\{v_2, v_4, v_5\}$ – вершинное покрытие графа, приведённого на рис. 2.5, поскольку каждое ребро этого графа инцидентно по крайней мере одной из этих трёх вершин. А множество $\{v_1, v_2, v_3\}$ не является вершинным покрытием этого графа, поскольку ребро $\{v_4, v_5\}$ не покрыто ни одной из вершин v_1, v_2, v_3.

Проблема минимального вершинного покрытия (MIN-VCP) – это минимизационная проблема, в которой для заданного графа G необходимо найти вершинное покрытие минимальной мощности.

[21] Ребро инцидентно некоторой вершине, если эта вершина является одним из двух элементов этого ребра – т. е. ребро $\{u, v\}$ инцидентно вершинам u и v.

Упражнение 2.37. Опишите множество всех вершинных покрытий графа, приведённого на рис. 2.5.

Упражнение 2.38. Дайте формальное описание MIN-VCP как шестёрки. Для представления входа и допустимых решений используйте алфавит $\{0, 1, \#\}$.

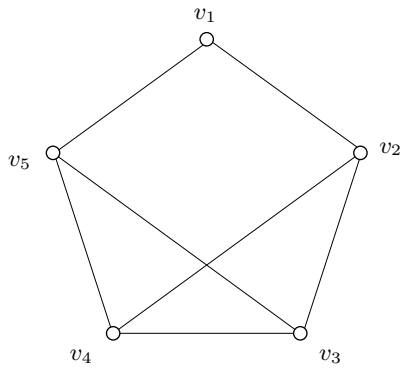


Рис. 2.5.

Клика графа $G = (V, E)$ – это некоторое множество $U \subseteq V$, такое что

$$\{\{u, v\} \mid u, v \in U, u \neq v\} \subseteq E$$

– т. е. любое множество U, вершины которого образуют подграф графа G, являющийся полным графом[22] (размера $|U|$). **Проблема максимальной клики (MAX-CL)** состоит в том, чтобы в данном графе найти клику максимального размера. Формальное представление MAX-CL в качестве оптимизационной проблемы таково.

MAX-CL
Вход: Граф $G = (V, E)$.
Ограничения: $\mathcal{M}(G) = \{S \subseteq V \mid \{\{u, v\} \mid u, v \in S, u \neq v\} \subseteq E\}$, т. е. $\mathcal{M}(G)$ содержит все клики графа G.
Стоимость: Для каждого $S \in \mathcal{M}(G)$ $\quad cost(S, G) = |S|$.
Цель: maximum.

Упражнение 2.39. Граф $T = (V, E')$ называется **остовным деревом** графа $G = (V, E)$, если T – некоторое дерево (связанный граф без циклов), и $E' \subseteq E$. Весом остовного дерева $T = (V, E')$ графа G называется $\sum_{e \in E'} c(e)$, т. е. сумма весов всех рёбер множества E'. Дайте формальное описание проблемы поиска минимального остовного дерева как оптимизационной проблемы.

Пусть $X = \{x_1, x_2, \ldots\}$ – множество Булевых переменных. Множество всех литералов над X – это $Lit_X = \{x, \overline{x} \mid x \in X\}$, где $\overline{x}$ является отрицанием переменной x.

[22] В некоторых книгах на русском языке употребляется фраза «образуют полный подграф графа G» (либо нечто аналогичное). Но, видимо, приведённый нами вариант определения более удачный. *(Прим. перев.)*

Значения 0 и 1 называются Булевыми значениями (константами). **Клауза**[23] – это конечная дизъюнкция над литералами (например, $x_1 \vee \overline{x}_3 \vee x_4 \vee \overline{x}_7$). (Булева) формула F является **конъюнктивной нормальной формой** (**КНФ**[24,25]), если F – некоторая конечная конъюнкция клауз.

Пример формулы над X в КНФ:

$$\Phi = (x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge \overline{x}_2 \wedge (x_2 \vee x_3) \wedge x_3 \wedge (\overline{x}_1 \vee \overline{x}_3).$$

Проблема максимальной выполнимости (MAX-SAT) состоит в поиске таких значений входных переменных данной формулы, представленной в виде КНФ, что число клауз, имеющих значение 1, максимальное. (Клаузы, имеющие значение 1 для данного набора значений переменных, по-другому называются выполненными.)

MAX-SAT

Вход: Формула $\Phi = F_1 \wedge F_2 \wedge \cdots \wedge F_m$ над X в КНФ, где F_i – некоторая клауза для $i = 1, \ldots, m$, $m \in \mathbb{N}$.

Ограничения: Для каждой формулы Φ над множеством из n Булевых переменных $\{x_{i_1}, x_{i_2}, \ldots, x_{i_n}\}$ множество допустимых решений –

$$\mathcal{M}(\Phi) = \{0,1\}^n.$$

{Каждое $\alpha = \alpha_1 \ldots \alpha_n \in \mathcal{M}(\Phi)$, где $\alpha_j \in \{0,1\}$ для $j = 1, \ldots, n$, представляет собой т. н. подстановку – в которой переменная x_{i_j} полагается равной α_j.}

Стоимость: Для каждой формулы Φ и любого $\alpha \in \mathcal{M}(\Phi)$ значение $cost(\alpha, \Phi)$ – число клауз формулы Φ, выполненных для набора значений α.

Цель: maximum.

Продолжим рассмотрение формулы Φ. В таблице 2.1 приведены все 8 возможных наборов значений переменных x_1, x_2, x_3 (8 возможных подстановок), и легко показать, что на наборах 001, 011 и 101 выполненными являются 5 клауз (максимально возможное число) – следовательно, эти подстановки являются оптимальными решениями для формулы Φ.

Пусть заданы система линейных уравнений и некоторая линейная функция, определённая на переменных этой системы. Проблема **целочисленного линейного программирования** (**ILP**) состоит в поиске такого решения системы, которое минимизирует эту линейную функцию. ILP может быть выражена как оптимизационная проблема следующим образом.

ILP

Вход: Матрица размерности $m \times n$:

$$A = [a_{ij}]_{i=1,\ldots m, j=1,\ldots,n},$$

два вектора:

---

[23] Русская терминология здесь допускает очень много вариантов – причём весьма непохожих друг на друга, и, более того, даже в чём-то противоречивых. Например – «дизъюнкт» у С. Яблонского; «конъюнктивный член» в переводе известного учебника Э. Мендельсона, выполненном под редакцией С. Адяна; «предложение»... Мы применяем кальку английского термина – и считаем этот вариант в данном случае более удачным. *(Прим. перев.)*

[24] Иными словами – F задана в КНФ.

[25] В этом случае мы будем применять русскую аббревиатуру. *(Прим. перев.)*

Таблица 2.1.

x_1	x_2	x_3	$x_1 \vee x_2$	$\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3$	$\overline{x}_2$	$x_2 \vee x_3$	x_3	$\overline{x}_1 \vee \overline{x}_3$	число подходящих клауз
0	0	0	0	1	1	0	0	1	3
0	0	1	0	1	1	1	1	1	5
0	1	0	1	1	0	1	0	1	4
0	1	1	1	1	0	1	1	1	5
1	0	0	1	1	1	0	0	1	4
1	0	1	1	1	1	1	1	0	5
1	1	0	1	1	0	1	0	1	4
1	1	1	1	0	0	1	1	0	3

$$b = (b_1, \ldots, b_m)^{\mathsf{T}} \quad \text{и} \quad c = (c_1, \ldots, c_n)$$

$(n, m \in \mathbb{N})$, где для $i = 1, \ldots, m$ и $j = 1, \ldots, n$ значения a_{ij}, b_i, c_j являются целыми числами.

Ограничения: $\mathcal{M}(A, b, c) = \{X = (x_1, \ldots, x_n)^{\mathsf{T}} \in (\mathbb{N}_0)^n \mid AX = b\}$.

$\{\mathcal{M}(A, b, c)$ – множество всех решений (векторов), которые удовлетворяют системе линейных уравнений $AX = b$ с заданными A и b.

Стоимость: Для каждого $X = (x_1, \ldots, x_n) \in \mathcal{M}(A, b, c)$,

$$cost(X, (A, b, c)) = c \cdot X = \sum_{i=1}^{n} c_i x_i.$$

Цель: minimum.

Кроме введённых выше проблем принадлежности и оптимизационных проблем мы будем рассматривать алгоритмические проблемы другой природы. Эти проблемы не требуют никаких входных данных, их единственная задача – генерировать некоторое слово (или бесконечное слово, т. е. бесконечную последовательность символов).

Определение 2.40. *Пусть Σ – некоторый алфавит, и пусть $x \in \Sigma^*$. Будем говорить, что алгоритм* **генерирует** *слово x, если A, имея на входе λ, даёт на выходе x .*

Следующая программа A генерирует слово 100111.

A :
```
begin
  write(100111);
end
```

Для каждого натурального n следующая программа A_n генерирует слово $(01)^n$.

A_n :
```
begin
  for i = 1 to n do
    write(01);
end
```

Программа, которая генерирует некоторое слово x, может рассматриваться как альтернативное представление этого слова. Поэтому можно вместо слов хранить в памяти программы, генерирующие эти слова.

Определение 2.41. *Пусть Σ – некоторый алфавит, и пусть $L \subseteq \Sigma^*$. A – алгоритм,* **перечисляющий L**, *если для каждого натурального n алгоритм A даёт на выходе $x_1, x_2, \ldots, x_n$, где $x_1, x_2, \ldots, x_n$ – первые n слов языка L, взятые в каноническом порядке.*

Пусть $\Sigma = \{0\}$, а $L = \{0^p \mid p - \text{простое число}\}$.

Вход: n.
Выход: $0^2, 0^3, 0^5, 0^7, \ldots, 0^{p_n}$, где p_n – n-е по порядку простое число.

Упражнение 2.42. Докажите, что язык рекурсивен, если и только если существует алгоритм, перечисляющий L.

2.4 Сложность по Колмогорову

В этом разделе мы будем рассматривать слова в качестве «информационных контейнеров», и сосредоточимся на поиске приемлемого способа измерения информационного содержания слов. Будем рассматривать только слова над алфавитом Σ_{bool}.

Неформальное описание способа измерения информации заключается в следующем: будем считать, что слово w содержит *мало* информации, если есть короткое представление этого слова (т. е. если оно сжимаемо), и что слово w содержит *много* информации, если не существует короткого представления w (т. е. представления, которое было бы короче, чем $|w|$). Интуиция подсказывает нам, что слово с малым информационным содержанием является регулярным, и, следовательно, его можно легко описать, а слово с высоким информационным содержанием нерегулярно.[26] Поэтому единственный способ представить это слово состоит в том, чтобы записать его постепенно, бит за битом.

На основании этой идеи мы должны говорить, что слово

$$011011011011011011011011,$$

имеющее короткое представление $(011)^8$, содержит меньше информации, чем слово

$$010110100010110100111011001 0.$$

Процесс построения некоторого представления слова w, которое короче, чем само w, а также короче, чем предыдущий вариант его представления, называется **сжатием** w.[27]

Следующий этап заключается в фиксации какого-либо удобного метода сжатия – и использовании длины окончательного, сжатого представление некоторого слова как меры его информационного содержания. Очевидно, что сжатое представление заданного слова также должно быть словом над алфавитом Σ_{bool} – поскольку использование большего по мощности алфавита для получения более короткого представления данного слова не является настоящим сжатием.

[26] Является случайным распределением нулей и единиц.

[27] Формально сжатие можно рассматривать как инъективное отображение из $(\Sigma_{\text{bool}})^*$ в $(\Sigma_{\text{bool}})^*$.

Упражнение 2.43. Найдите инъективное отображение H, действующее из $(\Sigma_{\text{bool}})^*$ в $\{0,1,2,3,4\}^* = \Sigma_5^*$, такое что для каждого слова $x \in (\Sigma_{\text{bool}})^*$, $|x| \geq 4$, выполнено условие

$$|x| \geq 2 \cdot |H(x)|.$$

Какой «коэффициент сжатия» (в последней формуле он равен 2) может быть достигнут, если вместо Σ_5 использовать алфавит Σ_m для некоторого натурального $m > 5$?

Один из вариантов сжатия слов может быть следующим. Начнём с алфавита $\{0,1,(,)\}$ – и будем для любого $w \in (\Sigma_{\text{bool}})^*$ представлять любое слово вида

$$w^a \quad \text{как} \quad (w)Bin(a).$$

Таким образом, например, (011)1000 представляет слово $(011)^8$, а

$$(0)1010(010)1(01)1101 \;-\; \text{слово} \;\; (0)^{10}(010)^1(01)^{13}.$$

А чтобы получить окончательное представление исходного слова в виде слова над алфавитом Σ_{bool}, можно использовать гомоморфизм, действующий из $\{0,1,(,)\}$ в Σ_{bool}, определённый следующим образом:

$$h(0) = 00, \quad h(1) = 11, \quad h(\,(\,) = 10, \quad h(\,)\,) = 01.$$

Таким образом, сжатое представление слова $(011)^8$ становится таким:

$$100011110111000000.$$

Заметим, что этот метод сжатия правилен – поскольку каждое сжиматое представление однозначно определяет заданное слово.

Проблема состоит в том, что можно предложить бесконечно много различных механизмов сжатия – какой из них правилен? Например, можно улучшить метод сжатия, введённый выше – путём сжатия представления показателей степени. При этом слово $(011)^{2^{20}}$ может быть использовано как более короткое представление слова $(011)^{1048576}$. Используя эту же стратегию, мы можем улучшить процедуру сжатия, генерируя различные представления вида

$$(01)1^{2^{2^n}}, (01)^{2^{2^{2^n}}}, \ldots$$

для регулярных слов над алфавитом Σ_{bool}. Это означает, что для любого из методов сжатия M существует некоторый другой метод сжатия, который для бесконечного числа слов над алфавитом Σ_{bool} работает лучше, чем M. Поэтому если мы хотим получить приемлемую, объективную меру информационного содержания слов, то ни одну из конкретных возможных стратегий (например – описанных выше) использовать нельзя.

Но ситуация может быть ещё хуже. Рассмотрим следующий метод сжатия. Для каждого $x \in (\Sigma_{\text{bool}})^*$ неотрицательное целое число $Number(x)$ может быть представлено в виде своего разложения на множители:

$$p_1^{i_1} \cdot p_2^{i_2} \cdot \cdots \cdot p_k^{i_k}$$

– для простых чисел $p_1 < p_2 < \cdots < p_k$, где $i_1, i_2, \ldots, i_k \in \mathbb{N}$. Возможно такое представление числа $p_1^{i_1} \cdot p_2^{i_2} \cdot \cdots \cdot p_k^{i_k}$ в виде слова над алфавитом $\{0,1,(,),\}$:

$$Bin(p_1)(Bin(i_1))Bin(p_2)(Bin(i_2))\dots Bin(p_k)(Bin(i_k)).$$

Применяя введённый выше гомоморфизм h, мы получаем двоичное представление слова x. К сожалению, этот метод сжатия невозможно сравнить с методом, основанным на повторениях подслова – т. е. иногда сжатие повторениями подслова может давать лучшие результаты, чем сжатие разложением на множители, а иногда наоборот.

Упражнение 2.44. Найдите два слова, такие что:

- метод сжатия повторениями подслова даёт существенно более короткое представление слова x, чем метод сжатия разложением на множители;
- метод сжатия разложением на множители даёт существенно более короткое представление слова y, чем метод сжатия повторениями.

Определение меры сложности должно согласовываться со здравым смыслом – т. е. быть обоснованным, а также должно иметь возможность применения в различных областях теоретической информатики. После рассмотрения зависимости между размерами представлений слова с помощью различных методов сжатия становится очевидным, что выбор одного конкретного метода для определения размера информационного содержания слова не даёт возможности сформулировать общие утверждения о количестве информации в словах. Следующее определение Колмогорова показывает возможный выход из этого тупика. Отметим, что введение алгоритма (программы) – очень важный момент, который и позволяет найти способ измерения информационного содержания.

Определение 2.45. *Для любого слова $x \in (\Sigma_{\rm bool})^*$* **сложность по Колмогорову $K(x)$ слова x** *– это двоичная длина наиболее короткой программы на Паскале, которая генерирует x.*[28]

Мы знаем, что любой компилятор Паскаля генерирует машинный код каждой написанной на Паскале синтаксически правильной программы, и что машинный код программы является ничем иным как словом над алфавитом $\Sigma_{\rm bool}$. Следовательно, для каждого слова $x \in (\Sigma_{\rm bool})^*$ мы рассматриваем все машинные коды программ, генерирующих x, – причём отметим, что их бесконечно много – а длина самой короткой из этих программ объявляется сложностью по Колмогорову слова x.

Является ли $K(x)$ хорошим кандидатом для определения размера информационного содержания x? Если мы хотим рассматривать все методы сжатия – то, конечно, является. Для каждого метода сжатия M, вычисляющего соответствующее сжатие $M(x)$ произвольного слова x, можно написать программу, содержащую данный метод M в качестве параметра (или некоторой программной константы), и генерирующую слово $M(x)$ на основе x. Однако нам нужен и более подробный анализ определения $K(x)$ – перед ним мы рассмотрим некоторые основные свойства сложности по Колмогорову.

Первое свойство $K(x)$ гарантирует, что это значение не может быть существенно бóльшим, чем $|x|$. Очевидно, что это свойство очень важно.

Лемма 2.46. *Существует константа d, такая что для каждого $x \in (\Sigma_{\rm bool})^*$ выполнено неравенство*

$$K(x) \leq |x| + d.$$

[28] Более точно – длина самого короткого двоичного кода программы, генерирующей x.

Доказательство. Для любого $x \in (\Sigma_{\mathrm{bool}})^*$ рассмотрим следующую программу[29] A_x:

```
begin
  write(x);
end
```

Элементы этой программы `begin`, `write`, `end` и запятые одинаковы для каждого $x \in (\Sigma_{\mathrm{bool}})^*$, и длины их представления в машинном коде ограничены небольшой константой d, независимой от x. Слово x представлено в программе A_x как x, и поэтому его вклад в длину (двоичного) машинного кода этой программы в точности равен $|x|$. □

Очевидно, что регулярные слова со многими повторениями некоторого подслова имеют небольшую сложность по Колмогорову. Пусть $y_n = 0^n \in \{0,1\}^*$ для любого $n \in \mathbb{N}$. Следующая программа Y_n генерирует y_n.

```
begin
  for i = 1 to n do
    write(0);
end
```

Все соответствующие элементы программ Y_n одинаковы для всех n – кроме самого́ числа n. Длина $Bin(n)$ в точности равна $\lceil \log_2(n+1) \rceil$ – и, таким образом, вклад n в двоичное представление Y_n не превышает $\lceil \log_2 n \rceil + 1$. Следовательно, существует некоторая константа c, такая что

$$K(y_n) \leq \lceil \log_2 n \rceil + c = \lceil \log_2 |y_n| \rceil + c$$

для всех $n \in \mathbb{N}$.

Теперь рассмотрим слова $z_n = 0^{n^2} \in \{0,1\}^*$ для $n|in\mathbb{N}$. Следующая программа Z_n генерирует слово z_n:

```
begin
  m := n;
  m := m × m;
  for i = 1 to m do
    write(0);
end
```

Все программы Z_n аналогичны – опять же за исключением самого́ числа n. Если общая длина двоичного кода всех частей программы Z_n (кроме n) равна d, то

$$K(z_n) \leq \lceil \log_2(n+1) \rceil + d \leq \lceil \log_2(\sqrt{|z_n|} \rceil + d + 1.$$

Обратим внимание на следующий важный момент при подсчёте длины. Программе Z_n не нужно хранить значение n^2 как константу. Вместо этого Z_n хранит исходное

[29] Для простоты мы используем некоторый паскалеподобный язык программирования. В нём мы, например, опускаем объявления переменных.

(меньшее) значение n, а значение n^2 вычисляется в процессе выполнения программы. Поскольку объём памяти, необходимой для выполнения программы Z_n, не включён в длину описания Z_n, то мы за счёт этого экономим приблизительно $\lceil \log_2 n \rceil$ битов представления данных.

Упражнение 2.47. Докажите следующее утверждение. Существует некоторая константа c, такая что для каждого натурального n выполнено неравенство

$$K\left((01)^{2^n}\right) \leq \lceil \log_2(n+1) \rceil + c = \left\lceil \log_2 \log_2 \left(\left| (01)^{2^n} \right| /2 \right) \right\rceil + c.$$

Упражнение 2.48. Найдите бесконечную последовательность слов $y_1, y_2, y_3, \ldots$ над алфавитом Σ_{bool}, удовлетворяющую таким условиям:

- $|y_i| < |y_{i+1}|$ для всех $i \in \mathbb{N}$;
- существует константа c, такая что для всех $i \in \mathbb{N}$ выполнено неравенство

$$K(y_i) \leq \lceil \log_2 \log_2 \log_2 |y_i| \rceil + c.$$

Упражнение 2.49. Докажите, что для любого $m \in \mathbb{N}$ существует некоторое слово w_m, такое что

$$|w_m| - K(w_m) > m.$$

Можно формально определить информационное содержание и для натуральных чисел – просто взяв сложность по Колмогорову их двоичных представлений.

Определение 2.50. Сложность по Колмогорову числа $n \in \mathbb{N}$ *есть*

$$\boldsymbol{K(n)} = K(Bin(n)).$$

Упражнение 2.51. Пусть $n = p\,q$ – некоторое натуральное число. Докажите, что

$$K(n) \leq K(p) + K(q) + c$$

для некоторой константы c, не зависящей от n, p и q.

Следующий фундаментальный результат показывает, что существуют слова, несжимаемые относительно сложности по Колмогорову.

Лемма 2.52. *Для каждого $n \in \mathbb{N}$ существует слово $w_n \in (\Sigma_{\text{bool}})^n$, такое что*

$$K(w_n) \geq |w_n| = n,$$

т. е. для любого $n \in \mathbb{N}$ существует несжимаемое слово длины n.

Доказательство. Доказательство основано на следующей простой комбинаторной идее.[30] В множестве $(\Sigma_{\text{bool}})^n$ существует в точности 2^n слов – обозначим их $x_1, \ldots, x_{2^n}$. Пусть для всех $i = 1, 2, \ldots, 2^n$ записью C-Prog$(x_i) \in \{0,1\}^*$ обозначен машинный код программы Prog(x_i), которая генерирует x_i, а $K(x_i) = |\text{C-Prog}(x_i)|$; т. е. Prog$(x_i)$ – одна из наиболее коротких[31] программ, генерирующих x_i.

[30] Другими словами – на пересчёте. *(Прим. перев.)*

[31] Заметим, что могут существовать несколько самых коротких программ, которые генерируют x_i; тогда мы просто берём любую из них.

Очевидно, что для $i \neq j$ программы C-Prog(x_i) и C-Prog(x_j) должны быть различными – потому что различны x_i и x_j. Это означает, что мы имеем 2^n различных машинных кодов

$$\text{C-Prog}(x_1), \text{C-Prog}(x_2), \ldots, \text{C-Prog}(x_{2^n})$$

наиболее коротких программ для слов $x_1, x_2, \ldots, x_{2^n}$. Достаточно показать, что длина хотя бы одного из этих кодов не меньше n.

На основе нашей комбинаторной идеи мы просто утверждаем, что не существует 2^n различных машинных кодов (слов), каждое из которых короче n. Каждый машинный код – слово над алфавитом $(\Sigma_{\text{bool}})^*$. Число слов длины i над алфавитом Σ_{bool} в точности равно 2^i. Следовательно, число всех непустых слов над Σ_{bool} с длиной, не превышающей $n-1$, равно

$$\sum_{i=1}^{n-1} 2^i = 2^n - 2 < 2^n.$$

Поэтому среди C-Prog$(x_1), \ldots,$ C-Prog(x_{2^n}) есть по крайней мере одно слово с длиной, не меньшей, чем n. Пусть это слово – C-Prog(x_j),

$$|\text{C-Prog}(x_j)| \geq n.$$

Поскольку

$$|\text{C-Prog}(x_j)| = K(x_j),$$

то x_j несжимаемо. □

Упражнение 2.53. Докажите, что для всех $i, n \in \mathbb{N}$, $i < n$, существуют $2^n - 2^{n-i}$ различных слов x множества $(\Sigma_{\text{bool}})^n$, таких что

$$K(x) \geq n - i.$$

Упражнение 2.54. Докажите, что существует бесконечно много натуральных m, таких что

$$K(m) \geq \lceil \log_2 m \rceil - 1.$$

Теперь вернёмся к вопросу о том, является ли сложность по Колмогорову удачной мерой информационного содержания слов: ведь вместо того, чтобы выбрать конкретный метод сжатия, мы взяли формальную модель программ, которая неявно применяет *все* такие возможные методы. Но мы в качестве языка программирования зафиксировали Паскаль – а не является ли это ограничением? Может быть, некоторый другой язык, например Ява или Си++, даст более короткое представление некоторых слов, чем их представление с помощью программы на Паскале? Не уменьшает ли выбор Паскаля надёжность измерения информационного содержания слов?

Ответы на эти вопросы отрицательны. В следующей теореме мы показываем, что выбор конкретного языка программирования на сложность по Колмогорову «оказывает ограниченное влияние» – следовательно, определение сложности по Колмогорову на основе Паскаля является разумной формализацией нашего интуитивного понятия информационного содержания.

Пусть для каждого слова x над алфавитом Σ_{bool} и каждого языка программирования A значение $K_A(x)$ является сложностью по Колмогорову слова x относительно языка программирования A – т. е. длиной самого короткого машинного кода программы на языке A, которая генерирует x.

Теорема 2.55. *Пусть A и B – языки программирования. Существует константа $c_{A,B}$, зависящая только от A и B, такая что*

$$|K_A(x) - K_B(x)| \leq c_{A,B}$$

для всех $x \in (\Sigma_{\text{bool}})^*$.

Доказательство. Мы знаем, что можно построить интерпретатор $U_{A\to B}$ для произвольных языков программирования A и B. $U_{A\to B}$ – это программа на языке B, которая переводит любую программу P_A (написанную на языке программирования A) в эквивалентную программу P_B (написанную на том же языке B). $U_{A\to B}$ выполняет программу P_B для входа x программы P_A. Рассматривая процесс генерации слова, мы можем упростить это утверждение – считая, что $U_{A\to B}$ получает программу P_A на языке программирования A в качестве входного параметра, и что $U_{A\to B}$ выполняет те же вычисления, что и P_A.

Следовательно, $U_{A\to B}$ со входом P_A – программа на языке программирования B, которая генерирует те же самые слова, что и P_A. Пусть $c_{A\to B}$ – двоичная длина программы $U_{A\to B}$ (без своего входа). Пусть также P_x – программа на A, которая генерирует x. Тогда $U_{A\to B}$ со входом P_x также генерирует x. Поскольку $U_{A\to B}$ – программа на B, мы получаем:

$$K_B(x) \leq K_A(x) + c_{A\to B}. \tag{2.1}$$

Если мы возьмём интерпретатор $U_{B\to A}$ языка программирования B (написанный на A), имеющий двоичную длину $c_{B\to A}$, то мы получим

$$K_A(x) \leq K_B(x) + c_{B\to A} \tag{2.2}$$

для каждого $x \in (\Sigma_{\text{bool}})^*$. Выбрав $c_{A,B}$ как максимальное из чисел $c_{A\to B}$ и $c_{B\to A}$, мы на основе неравенств (2.1) и (2.2) получим, что

$$|K_A(x) - K_B(x)| \leq c_{A,B}.$$

□

Итак, мы решили, что сложность по Колмогорову является приемлемой характеристикой меры информационного содержания слова; теперь покажем, что это понятие можно использовать в качестве инструмента для исследования некоторых важных отношений, использующихся в математике и теоретической информатике. Ниже мы представим три приложения сложности по Колмогорову, которые можно понять без каких-либо специальных предварительных знаний. Другие фундаментальные приложения описываются далее – в главах, посвящённых конечным автоматам и теории высчислимости.

Первое приложение – с философской точки зрения – находится на уровне формирования понятий. «Случайность» – это одно из основных научных понятий, и мы рассмотрим его в главе 8, посвящённой рандомизированным алгоритмам. А здесь мы бы хотели обсудить следующий вопрос:

> *какие объекты – или их представления в виде слов – можно считать случайными?*

Классическая теория вероятности не поможет нам ответить на этот вопрос, поскольку она только *назначает* вероятности каждого из возможных случайных событий. Например, если мы рассматриваем случайный выбор слова, принадлежащего языку $(\Sigma_{\mathrm{bool}})^n$, то каждое слово этого языка может быть выбрано с одинаковой вероятностью.

А можно ли связать вероятность выбора слова с его степенью случайности? Если можно – то в приведённом примере слово 0^n должно быть столь же случайным, как и любое нерегулярное слово; а последнее не соответствует нашему интуитивному пониманию прилагательного «случайный». Согласно словарю английского языка Би-Би-Си,

нечто является случайным, если сделано без определённого плана или образца,

т. е. некоторый случайный объект имеет хаотическую структуру, полностью нерегулярен.[32]

Для строгого определения случайного слова и нужны наши предыдущие рассуждения, касающиеся сжатия и информационного содержания. Слово случайно, когда оно не имеет более короткого представления, чем его длина – т. е. не позволяет произвести сжатие. Другими словами,

слово случайно – если любое его описание имеет по крайней мере такой же размер, как и его полное побитовое описание.

Поэтому следующее определение, по-видимому, является наиболее известной формализацией слова «случайный».

Определение 2.56. *Слово* $x \in (\Sigma_{\mathrm{bool}})^*$ *называется* **случайным**, *если*

$$K(x) \geq |x|.$$

Натуральное n *называется* **случайным**, *если*

$$K(n) = K(Bin(n)) \geq \lceil \log_2(n+1) \rceil - 1.$$

Заметим, что мы вычитаем 1 из $\lceil \log_2(n+1) \rceil$, поскольку знаем, что каждое двоичное представление любого натурального числа начинается с цифры 1.

Второе приложение сложности по Колмогорову показывает, что существование программ (алгоритмов), решающих проблему принадлежности $(\Sigma_{\mathrm{bool}}, L)$, даёт некоторую информацию о сложности по Колмогорову слов языка L. Например, если каждые два слова в L имеют разную длину, то сложность по Колмогорову каждого слова $x \in L$ не превосходит $\log_2 |x|$.

Теорема 2.57. *Пусть* L *– некоторый язык над алфавитом* Σ_{bool}. *Пусть для любого возможного натурального* n *слово* z_n *является* n*-м словом этого языка относительно канонического порядка. Кроме того предположим, что существует программа (алгоритм)* A_L, *которая решает проблему принадлежности* $(\Sigma_{\mathrm{bool}}, L)$. *Тогда существует константа* c, *не зависящая от* n, *такая что*

$$K(z_n) \leq \lceil \log_2(n+1) \rceil + c,$$

для каждого натурального n.

[32] Одно из толкований слова «случайный» в «Толковом словаре русского языка» С.Ожегова является аналогичным: «возникший, появившийся непредвиденно». *(Прим. перев.)*

Доказательство. Для любого натурального n рассмотрим программу C_n, генерирующую слово z_n. Такая программа C_n должна содержать A_L как подпрограмму. Программа C_n может быть описана следующим образом:

```
begin
  i = 0;
  x := λ;
  while  i <n do begin
    ⟨Выполнить A_L(x) – т. е. применить к x подпрограмму A_L⟩;
    if A_L(x) = 1 then begin
      i := i + 1;
      z := x;
    end;
    x := следующее слово после x
         относительно канонического порядка;
  end
  write(z);
end
```

В программе C_n слова языка $(\Sigma_{\mathrm{bool}})^*$ генерируются в каноническом порядке. Для каждого x программа C_n проверяет условие $x \in L$ с помощью подпрограммы A_L. При этом C_n подсчитывает число слов, принятых A_L. Очевидно, что выходом программы C_n является n-е слово z_n языка L.

Ещё раз отметим, что все программы C_n идентичны – различие заключается в параметре n. Пусть c – длина двоичного машинного кода программы C_n, без кода числа n. Тогда для каждого $n \in \mathbb{N}$ двоичная длина программы C_n в точности равна

$$c + \lceil \log_2(n+1) \rceil.$$

Поскольку программа C_n генерирует z_n, значение $K(z_n)$ не превышает её двоичную длину. □

Типичное ошибочное мнение о сложности по Колмогорову заключается в том, что двоичная длина программы C_n якобы зависит от размера памяти, необходимой ей в процессе своего выполнения для хранения значений переменных c, i и z. Опровергнем это неверное мнение: для хранения переменных программы C_n, возможно, требуется большее количестве битов (более длинное двоичное представление) – но это никак не влияет на длину двоичного представления программы C_n, поскольку все упомянутые ранее значения не являются частью такого представления. В описании программы C_n мы должны закодировать только названия переменных. Единственное *значение*, которое является частью программы C_n – это n. Для программы C_n значение n является константой, двоичное представление которой является частью машинного кода этой программы. В следующей главе мы разовьём идею доказательства теоремы 2.57 – для доказательства существования языков, которые не могут быть распознаны конечными автоматами.

Упражнение 2.58. Пусть p – некоторый полином, зависящий от одной переменной, а $L \subseteq (\Sigma_{\mathrm{bool}})^*$ – бесконечный рекурсивный язык, удовлетворяющий неравенству $|L \cap (\Sigma_{\mathrm{bool}})^n| \leq p(n)$ для всех натуральных n. Для любого натурального m записью z_m

обозначим m-е слово языка L относительно канонического порядка. Какова верхняя граница значения $K(z_m)$ для заданного m?

Последнее приложение сложности по Колмогорову, рассматриваемое в этом разделе, связано с одним из фундаментальных результатов теории чисел – имеющим огромную важность для разработки рандомизированных алгоритмов. Пусть для любого натурального n запись $Prim\,(n)$ обозначает количество простых чисел, не превосходящих данное n. Следующая фундаментальная теорема утверждает, что простые числа плотно распределены среди целых чисел; это можно увидеть на примере нескольких первых чисел:

$$1, \underline{2}, \underline{3}, 4, \underline{5}, 6, \underline{7}, 8, 9, 10, \underline{11}, 12, \underline{13}, 14, 15, 16, \underline{17}, 18, \underline{19}, 20, 21, 22, \underline{23}, \ldots$$

Теорема 2.59 (о простых числах).

$$\lim_{n\to\infty} \frac{Prim\,(n)}{n/\ln n} = 1.$$

□

Теорема о простых числах – одно из самых замечательных открытий во всей математике. Она говорит нам, что количество простых чисел растёт приблизительно с такой же скоростью, как функция $n/\ln n$.[33] При этом для «малых» значений n точное значение $Prim\,(n)$ может быть посчитано. В таблице 2.2 приведены некоторые из этих значений.

Таблица 2.2.

n	$Prim\,(n)$	$\frac{Prim(n)}{(n/\ln n)}$
10^3	168	~ 1.161
10^6	78 498	~ 1.084
10^9	50 847 478	~ 1.053

Следующее неравенство показывает, насколько близки значения $Prim\,(n)$ и $n/\ln n$ для $n \geq 67$.

$$\ln n - \frac{3}{2} < \frac{n}{Prim\,(n)} < \ln n - \frac{1}{2}.$$

Упражнение 2.60. Пусть для $i \in \mathbb{N}$ p_i обозначает i-е по значению простое число. Используя теорему 2.59, докажите, что

$$\lim_{n\to\infty} \frac{p_n}{n \ln n} = 1.$$

Гаусс, основываясь на эмпирическом исследовании таблиц простых чисел, первым догадался, что $Prim\,(n)\,/\,n$ приблизительно равно $1/\ln n$ – особенно для больших значений n. Но строгое доказательство этой догадки Гаусса не могло быть выполнено с

[33] Однако теорема о простых числах даёт только это среднее значение плотности – и на её основе мы ничего не можем сказать о *распределении* конкретных простых чисел среди всех натуральных. Отметим только, что это распределение весьма нерегулярно.

помощью известных в то время математических методов. Первые идеи, нужные для решения этой проблемы, были начерно изложены Риманом в середине XIX века, а впоследствии были независимо доказаны Адамаром и Пуассоном в 1896 г. – однако эти доказательства очень сложные. Первый интересный момент этих доказательств заключался в том, что требовалось «неестественное» использование комплексных чисел для доказательства теоремы о натуральных – и только спустя ещё полвека Норберт Винер смог изменить доказательство, не используя комплексных чисел. Другой важный момент: насколько быстро ряд $Prim\,(n)\,/(n/\ln n)$ сходится к 1? Этот вопрос связан с широко известной гипотезой Римана – одной из наиболее интересных открытых проблем математики.

Теперь рассмотрим, как можно применить понятие сложности по Колмогорову для получения простого комбинаторного доказательства упрощённой версии теоремы о простых числах. Эта упрощённая версия, однако, всё же вполне достаточна для всех наших приложений, связанных с разработкой рандомизированных алгоритмов, а представленное далее простое доказательство помогает нам понять, почему существует столь много простых чисел. Основная идея доказательства заключается в том, что если бы плотность простых чисел была значительно меньше, чем $n/\ln n$, то можно было бы представить любое натуральное число таким коротким способом, который противоречил бы *инъективности* представления всего множества натуральных чисел.[34] Итак, докажем следующий фундаментальный результат, утверждающий, что имеется бесконечно много простых чисел, обладающих специальным свойством.

Лемма 2.61. *Пусть $S = n_1, n_2, n_3, \ldots$ – возрастающая бесконечная последовательность натуральных чисел, такая что $K(n_i) \geq \lceil \log_2 n_i \rceil / 2$. Пусть для любого $i \in \mathbb{N}$ число q_i является наибольшим простым делителем числа n_i. Тогда множество*

$$Q = \{q_i \mid i \in \mathbb{N}\}$$

бесконечно.

Доказательство. Докажем лемму 2.61 методом от противного. Пусть $Q = \{q_i \mid i \in \mathbb{N}\}$ – конечное множество, а p_m – наибольшее число в этом множестве; по построению, p_m является простым. Поэтому каждое натуральное число n_i ($i \in \mathbb{N}$), принадлежащее последовательности S, может быть однозначно представлено в виде

$$n_i = p_1^{r_{i,1}} \cdot p_2^{r_{i,2}} \cdot \ldots \cdot p_m^{r_{i,m}}$$

– для некоторых $r_{i,1}, r_{i,2}, \ldots, r_{i,m} \in \mathbb{N}_0$. Поскольку $p_1, p_2, \ldots, p_m$ фиксированы, то для представления каждого числа множества S показателями $r_{i,1}, r_{i,2}, \ldots, r_{i,m}$ однозначно определяются значения n_i. Следовательно, можно написать программу A_i, генерирующую для данных $r_{i,1}, \ldots, r_{i,m}$ значение n_i.

Пусть c – двоичная длина программы A_i, в которую не включается память, необходимая для представления её параметров $r_{i,1}, \ldots, r_{i,m}$ – иными словами, двоичная длина общей части всех таких программ. Поскольку верхняя граница двоичной длины программы A_i равна $K(n_i)$, то мы получаем, что для всех $i \in \mathbb{N}$ выполняется неравенство

[34] А мы уже видели, что невозможно обеспечить сжатое представление каждого числа (слова).

$$K(n_i) \leq c + \lceil \log_2(r_{i,1}+1) \rceil + \lceil \log_2(r_{i,2}+1) \rceil + \cdots + \lceil \log_2(r_{i,m}+1) \rceil.$$

Поскольку $r_{i,j} \leq \log_2 n_i$ для всех $j \in \{1,2,\ldots,m\}$, то мы получаем, что для всех $i \in \mathbb{N}$ выполнено следующее:

$$K(n_i) \leq c + m \cdot \lceil \log_2(\log_2 n_i + 1) \rceil.$$

А так как m и c являются константами, не зависящими от i, неравенство

$$\lceil \log_2 n_i \rceil / 2 \leq c + m \cdot \lceil \log_2(\log_2 n_i + 1) \rceil$$

может быть истинным только для конечного количества значений i. Однако это противоречит сделанному нами предположению о том, что

$$K(n_i) \geq \lceil \log_2 n_i \rceil / 2$$

для всех натуральных n. □

Упражнение 2.62. В лемме 2.61 мы предполагали, что существует некоторая возрастающая бесконечная последовательность натуральных чисел $n_1, n_2, n_3, \ldots$, обладающая свойством $K(n_i) \geq \lceil \log_2 n_i \rceil / 2$. Как можно ослабить это предположение, не меняя заключение леммы?

Упражнение 2.63. Пусть t – некоторое натуральное число. Сколько чисел n во множестве $\{2^t, 2^t+1, \ldots, 2^{t+1}-1\}$ не удовлетворяет условию

$$K(n) \geq \lceil \log_2 n \rceil / 2 \ \ ?$$

Лемма 2.61 показывает не только бесконечность множества простых чисел, но и бесконечность множества наибольших простых делителей любой бесконечной последовательности натуральных чисел, имеющих нетривиальные значения сложности по Колмогорову. Мы будем использовать это утверждение для получения следующей нижней границы значения $Prim\,(n)$.

Теорема 2.64.* *Для бесконечно многих $k \in \mathbb{N}$ выполнено неравенство*

$$Prim\,(k) \geq \frac{k}{64 \log_2 k \cdot (\log_2 \log_2 k)^2}.$$

Доказательство. Вспомним, что p_j обозначает j-е по величине простое число. Пусть $n \geq 2$ – произвольное целое, а p_m – наибольший простой делитель этого n. Очевидно, что n может быть сгенерировано из пары $(p_m, n/p_m)$ просто с помощью умножения p_m на n/p_m. Заметим, что это может быть сделано даже с меньшей информацией, с помощью пары меньшей длины, а именно – $(m, n/p_m)$, поскольку программа, просто перебирающая первые m простых чисел, генерирует p_m для заданного m.

Теперь ответим на вопрос, каким образом однозначно представить пару $(m, n/p_m)$ в виде одного слова над алфавитом Σ_{bool}. Нельзя просто объединить $Bin(m)$ и $Bin(n/p_m)$ в одно слово – поскольку из-за нескольких возможных интерпретаций слова $Bin(m)Bin(n/p_m)$ невозможно однозначно определить границу между его подсловами $Bin(m)$ и $Bin(n/p_m)$.

Однако можно предложить следующий способ кодирования $Bin(m)$. Пусть

$$Bin(m) = a_1 a_2 a_3 \dots a_{\lceil \log_2(m+1) \rceil}$$

для некоторых $a_i \in \Sigma_{\text{bool}}$, $i = 1, 2, \dots, \lceil \log_2(m+1) \rceil$. Определим

$$\overline{Bin}(m) = a_1 0 a_2 0 a_3 0 \dots a_{\lceil \log_2(m+1) \rceil - 1} 0 a_{\lceil \log_2(m+1) \rceil} 1.$$

Слово $\overline{Bin}(m) Bin(n/p_m)$ является однозначным представлением пары $(m, n/p_m)$, и, следовательно, однозначным представлением числа n – поскольку окончание кода $\overline{Bin}(m)$ числа m точно указывает на первую цифру 1 в чётной позиции кода $\overline{Bin}(m) Bin(n/p_m)$. Длина этого представления равна

$$2 \cdot \lceil \log_2(m+1) \rceil + \lceil \log_2\big(\frac{n}{p_m} + 1\big) \rceil.$$

Однако эта длина всё ещё слишком велика – и мы уменьшаем её путём кодирования пары $(m, n/p_m)$ в виде

$$\overline{Bin}(\lceil \log_2(m+1) \rceil) Bin(m) Bin(n/p_m).$$

Последнее представление также является однозначным – поскольку окончание слова $\overline{Bin}(\lceil \log_2(m+1) \rceil)$ помечено первым символом 1 в чётной позиции, а число $\overline{Bin}(\lceil \log_2(m+1) \rceil)$ говорит о том, что следующие $\lceil \log_2(m+1) \rceil$ бит принадлежат представлению $Bin(m)$. Длина такого кода пары $(m, n/p_m)$ равна

$$2\lceil \log_2 \lceil \log_2(m+1) \rceil \rceil + \lceil \log_2(m+1) \rceil + \lceil \log_2\big(\frac{n}{p_m} + 1\big) \rceil.$$

Используя такую стратегию, можно выполнить бесконечно много усовершенствований, но для доказательства нашей теоремы достаточно сделать ещё только один шаг. А именно, мы окончательно представим пару $(m, n/p_m)$ как

$$\begin{aligned} &Word(m, n/p_m) = \\ &\overline{Bin}(\lceil \log_2 \lceil \log_2(m+1) \rceil \rceil) Bin(\lceil \log_2(m+1) \rceil) Bin(m) Bin(n/p_m). \end{aligned}$$

Следовательно,

$$\begin{aligned} |Word(m, n/p_m)| = {} & 2 \cdot \lceil \log_2 \lceil \log_2 \lceil \log_2(m+1) \rceil \rceil \rceil \\ & + \lceil \log_2 \lceil \log_2(m+1) \rceil \rceil \\ & + \lceil \log_2(m+1) \rceil + \lceil \log_2\big(\frac{n}{p_m} + 1\big) \rceil. \end{aligned} \tag{2.3}$$

Теперь рассмотрим слово $Word(m, n/p_m)$ в качестве сжатия слова $Bin(n)$. Мы применяем одну и ту же (фиксированную) стратегию сжатия для каждого n – поэтому, используя те же самые аргументы, что и в лемме 2.52 и упражнении 2.53, мы получаем, что при всех $i \in \mathbb{N}$ для более чем половины из чисел n, принадлежащих множеству

$$S_i = \{2^i, 2^i + 1, \dots, 2^{i+1} - 1\},$$

выполняется неравенство

$$|Word(m, n/p_m)| \geq \lceil \log_2(n+1) \rceil - 1.$$

Аналогично, для более чем половины чисел n из того же множества S_i выполняется неравенство

$$K(n) \geq \lceil \log_2(n+1) \rceil - 2.$$

Для каждого натурального i последний факт влечёт существование некоторого $n_i \in S_i$ (т. е. $2^i \leq n_i \leq 2^{i+1} - 1$), такого что

$$|\mathit{Word}(m, n_i/p_m)| \geq \lceil \log_2(n_i+1) \rceil - 1$$

и

$$K(n) \geq \lceil \log_2(n_i+1) \rceil - 2.$$

Другими словами, существует бесконечно много натуральных n, удовлетворяющих неравенствам

$$|\mathit{Word}(m, n/p_m)| \geq \lceil \log_2(n+1) \rceil - 1 \tag{2.4}$$

и

$$K(n) \geq \lceil \log_2(n+1) \rceil - 2. \tag{2.5}$$

Объединение неравенств (2.4) и (2.3) даёт оценку

$$\begin{aligned} \lceil \log_2(n+1) \rceil - 1 \leq 2\lceil \log_2 \lceil \log_2 \lceil \log_2(m+1) \rceil \rceil \rceil \\ + \lceil \log_2 \lceil \log_2(m+1) \rceil \rceil + \lceil \log_2(m+1) \rceil + \lceil \log_2 \big(\frac{n}{p_m} + 1 \big) \rceil. \end{aligned}$$

Поскольку $\log_2(n/p_m) = \log_2 n - \log_2 p_m$, мы получаем, что

$$\begin{aligned} \log_2 p_m &\leq 2 \cdot \lceil \log_2 \log_2 \log_2(m+1) \rceil + \lceil \log_2 \log_2(m+1) \rceil + \lceil \log_2(m+1) \rceil + 2 \\ &\leq 2 \cdot \log_2 \log_2 \log_2 m + \log_2 \log_2 m + \log_2 m + 6. \end{aligned}$$

Следовательно,

$$p_m \leq 64 \cdot m \cdot \log_2 m \cdot (\log_2 \log_2 m)^2. \tag{2.6}$$

Применяя неравенство (2.5), мы получаем, что бесконечная последовательность $n_1, n_2, n_3, \ldots$ условиям леммы 2.61 удовлетворяет. Таким образом, неравенство (2.6) выполняется для бесконечно многих натуральных чисел – значит, и для бесконечно многих простых чисел p_m. Из этого следует, что m-е по величине простое число меньше, чем

$$64 \cdot m \cdot \log_2 m \cdot (\log_2 \log_2 m)^2$$

для бесконечно многих m. Последнее же равносильно тому, что имеется по крайней мере m простых чисел среди первых $64 \cdot m \cdot \log_2 m \cdot (\log_2 \log_2 m)^2$ натуральных, т. е. для бесконечно многих m выполнено неравенство

$$\mathit{Prim}\left(64 \cdot m \cdot \log_2 m \cdot (\log_2 \log_2 m)^2\right) \geq m. \tag{2.7}$$

В (2.7) положим $k = 64 \cdot m \cdot \log_2 m \cdot (\log_2 \log_2 m)^2$. Поскольку $k \geq m$, мы получаем

$$\begin{aligned} \mathit{Prim}\,(k) \geq m &= \frac{k}{64 \cdot \log_2 m \cdot (\log_2 \log_2 m)^2} \\ &\geq \frac{k}{64 \cdot \log_2 k \cdot (\log_2 \log_2 k)^2}. \end{aligned}$$

□

Упражнение 2.65. В качестве сжатого представления числа $n = p_m(n/p_m)$ рассмотрим $\overline{Bin}(\lceil \log_2 m \rceil) \cdot Bin(m) \cdot Bin(n/p_m)$. Если мы будем использовать это представление в доказательстве теоремы 2.59 – то какую нижнюю границу значения $Prim\,(k)$ можно получить?

Упражнение 2.66. Какое утверждение можно получить, если в качестве представления числа $n = p_m \cdot (n/p_m)$ в доказательстве теоремы 2.59 взять

$$\overline{Bin}(\lceil \log_2 \lceil \log_2 \lceil \log_2 (m+1) \rceil \rceil \rceil) Bin(\lceil \log_2 \lceil \log_2 (m+1) \rceil \rceil)$$
$$Bin(\lceil \log_2 (m+1) \rceil) Bin(m) Bin(n/p_m)\,?$$

Упражнение 2.67.* Какую наибольшую нижнюю границу значения $Prim\,(k)$ можно получить в доказательстве теоремы 2.59?

Упражнение 2.68.* Пусть $p_1, p_2, p_3, \ldots$ – возрастающая последовательность всех простых чисел. Примените теорему о простых числах для доказательства существования такой константы c, что для всех натуральных m выполнено неравенство

$$K(p_m) \leq \lceil \log_2 p_m \rceil - \lceil log_2 \log_2 p_m \rceil + c.$$

2.5 Заключение

В этой главе мы ввели основные понятия теоретической информатики – такие как алфавит, слово и язык. Алфавит – это произвольное непустое множество символов, имеющее такой же смысл, как алфавит любого естественного языка. Но термин «слово над алфавитом» в теоретической информатике соответствует произвольному тексту, составленному из символов этого алфавита. Любое множество слов (текстов) над одним и тем же алфавитом называется языком.

Эти основные понятия используются во всех областях обработки данных. Мы начинаем рассмотрение таких областей с двух специальных алгоритмических проблем, в которых входы и выходы представлены некоторыми словами. Первая из них – проблема принадлежности. Она заключается в том, чтобы определить, принадлежит ли некоторое слово заданному языку L. Вторая – т. н. оптимизационная проблема, она более сложна. В ней каждое входное слово, во-первых, специальным образом кодирует некоторое множество ограничений, и, во-вторых, определяет стоимости объектов (допустимых решений), удовлетворяющих этим ограничениям. При этом обычно заданным ограничениям удовлетворяет довольно много допустимых решений – и целью задачи является поиск того из них, которое является оптимальным с точки зрения заданной стоимости.

Сложность по Колмогорову – это обоснованное, согласующееся с нашей интуицией понятие, предназначенное для измерения объёма (количества) информации в словах над алфавитом $\Sigma_{\text{bool}} = \{0, 1\}$. Сложность по Колмогорову слова x можно рассматривать как двоичную длину самой короткой программы на Паскале, генерирующей x. А саму эту программу можно считать специальным сжатием (сжатым представлением) генерируемого слова. Следовательно, сложность по Колмогорову слова x – это длина наиболее короткого представления x над алфавитом $\{0, 1\}$.

А несжимаемые слова очень «нерегулярны» – и поэтому они могут рассматриваться как случайные. Существует бесконечно много случайных слов над алфавитом $\{0, 1\}$

– по крайней мере одно для каждой длины. Сложность по Колмогорову помогает измерять не только объём информации, содержащейся в слове, но и «уровень случайности» слова. Это понятие используется также во многих областях теоретической информатики и математики в качестве мощного – но в то же время достаточно очевидного – инструмента для проверки полученных результатов.

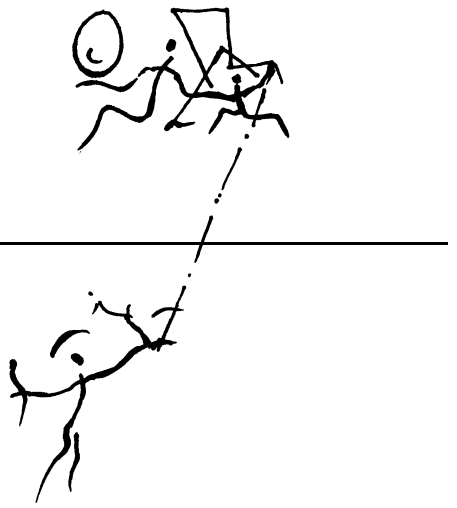
А мы применяем сложность по Колмогорову для доказательства бесконечности множества простых чисел – и, в ещё большей степени, для объяснения их большой плотности во множестве всех натуральных. Если бы эта плотность была мала, то в качестве следствия мы получали бы существование сжатого представления для всех слов – а это противоречило бы доказанному существованию случайных (несжимаемых) слов.

Предмет изучения теории формальных языков – алфавиты, слова, языки и способы их представления. Эта теория – одна из старейших и наиболее классических областей информатики. Поскольку объекты изучения информатики – такие как данные, информация, содержимое памяти, сообщения, программы, теоремы, доказательства, вычисления – представлены словами, теория формальных языков является основой для других фундаментальных областей теоретической информатики – теории вычислимости, теории трансляции и др.

В нашем вводном материале мы упростили представление основных понятий теории формальных языков – до того минимального уровня, который необходим для изложения основных тем данной книги, представленных в последующих главах. Одна из причин такого упрощения заключается в том, что уже существует много замечательных книг по теории формальных языков – и мы не видим смысла писать «101-ю книгу» на эту тему. Для читателей, заинтересовавшихся основами этой теории, мы рекомендуем книгу Хопкрофта и Ульмана [28] – это один из наиболее удачных классических учебников. Заметим, что недавно вышло новое издание этой книги [27], в котором рассматривается несколько новых тем. Автором ещё одного замечательного классического учебника [60], посвящённого формальным языкам, является Саломаа. Главная особенность вышеупомянутых учебников – отличный, но при этом несложный способ изложения материала, введения новых понятий и объяснения их – на интуитивном уровне, но в то же время достаточно строго и подробно. Эта особенность делает данные книги жемчужинами среди имеющихся на данную тему – и поэтому крайне желательными для использовании в образовательном процессе. Кроме упомянутых книг, мы можем также порекомендовать учебники Эрка и Призе [18], Шёнинга [62], а также Вегенера [71].

Идея понятия сложности по Колмогорову как меры количества содержащейся в словах информации была предложена в 1960-е годы – независимо Колмогоровым [39, 40] и Чаитиным [9, 10, 11]. Обстоятельный обзор этой темы приведён в монографии Ли и Витаного [43]. К сожалению, эта книга написана для научных работников и вряд ли подойдёт для начинающих. Весьма интересные – и в то же время простые примеры применения сложности по Колмогорову можно найти в книге Шёнинга [62].

Смысл высшего образования –
говорить просто
о сложных вещах.
Р. Эмерсон



3

Конечные автоматы

3.1 Цели и задачи главы

Конечный автомат – это самая простая вычислительная модель, рассматриваемая в теоретической информатике; его можно считать моделью специальных программ – программ, решающих задачи без использования каких-либо переменных. Конечные автоматы работают в режиме реального времени – в том смысле, что входное слово доступно только для чтения, только один раз и только слева направо, а вывод (результат работы) получается сразу после прочтения последнего входного символа.

Основная причина рассмотрения конечных автоматов в этой книге – вовсе не введение в теорию автоматов, не изучение её главных результатов. Вместо этого данная глава преследует специальную «дидактическую» цель – рассмотрение конечных автоматов даёт очень простой и ясный способ понимания процесса моделирования вычислений вообще. Введение основных понятий теоретической информатики при описании работы конечного автомата – конфигурации, шага вычисления, процесса вычисления вообще, детерминизма, недетерминизма, моделирования работы автомата (с помощью другого автомата) – даёт читателю интуитивное представление об общих, фундаментальных свойствах этих понятий, а также некоторый опыт работы с ними. А наличие у читателя такой интуиции и такого опыта помогает впоследствии – например, упрощает понимание точных определений этих терминов в *общих* моделях алгоритмических вычислений.

Таким образом, в этой главе мы научимся без излишней сложности – но в то же время вполне строго – исследовать и моделировать некоторый ограниченный подкласс алгоритмов. И, кроме первого знакомства с вышеупомянутыми фундаментальными терминами информатики, мы заодно научимся доказывать неразрешимость некоторых конкретных проблем в этом подклассе.

3.2 Различные варианты представления конечных автоматов

Определяя вычислительную модель, мы должны ответить на следующие вопросы.

- Какие элементарные операции допустимы, т. е. какие команды можно использовать для построения программы?

- Насколько велик объём доступной памяти, и какие имеются операции работы с ней (операции доступа)?
- Как в компьютер (в вычислительную модель) поступают входные данные?
- Как определяется выход (вывод, результат)?

Конечный автомат не имеет никакой памяти – кроме той, в которой записана сама программа, а также указателя на выполняющуюся в текущее время строку (команду) программы. Это означает, что программе не позволяется использовать переменные. На первый взгляд это требование может показаться странным: можно спросить, как же программа может что-либо вычислять без использования переменных. Ответ на вопрос – т. е. идея определения конечных автоматов – состоит в том, что содержание указателя – то есть номер фактически выполняемой строки программы – это и есть измененяемая информация, и программа работает с такой псевдопеременной.

Если $\Sigma = \{a_1, a_2, \ldots, a_k\}$ – заданный алфавит входных символов, то конечный автомат может использовать только следующие команды (инструкции, элементарные операции):

$$\begin{aligned} &\texttt{select } \mathit{input} = a_1 \texttt{ goto } i_1 \\ &\qquad\quad \mathit{input} = a_2 \texttt{ goto } i_2 \\ &\qquad\qquad \vdots \\ &\qquad\quad \mathit{input} = a_k \texttt{ goto } i_k \end{aligned}$$

Выполнение каждой из этих команд `select` состоит в том, что программа, моделирующая процесс работы конечного автомата[1], читает следующий входной символ и сравнивает его со всеми $a_1, a_2, \ldots, a_k$. Если входной символ совпадает с некоторым a_j, то программа продолжает работу со строкой i_j (т. е. следующая исполняемая команда – та, которая записана в строке i_j). Выполнение команды автоматически совмещается с удалением прочитанного символа – после чего конечный автомат читает следующий входной символ в строке i_j. Каждая строка программы содержит в точности одну команду вышеупомянутого типа. Строки программы пронумерованы числами $0, 1, 2, 3, \ldots$, и работа (вычисление) моделирующей программы всегда начинается со строки 0. Если Σ состоит только из двух символов (например, 0 и 1), то вместо команды `select` мы можем использовать более простую команду `if` ... `then` ... `else`:

$$\texttt{if } \mathit{input} = 1 \texttt{ then goto } i \texttt{ else goto } j.$$

Такие программы можно рассматривать как специальные проблемы принадлежности[2] – они отвечают на вопрос, принадлежит ли некоторое слово заданному языку. На

[1] Будем иногда называть её моделирующей программой. Её не следует путать с программой самого автомата – а выше в данном разделе слово «программа» уже употреблялось в таком смысле. Последнюю программу будем иногда называть программой-автоматом. *(Прим. перев.)*

[2] По-видимому, правильнее было бы сказать – как алгоритмы для решения специального варианта проблемы принадлежности. *(Прим. перев.)*

каждом шаге вычисления ответ (локальный результат работы) полностью определяется номером строки. Программа состоит из m строк $\{0, 1, 2, \ldots, m-1\}$. Кроме того, задаётся специальное подмножество $F \subset \{0, 1, 2, \ldots, m-1\}$; мы можем интерпретировать F как т. н. допускающее множество строк. Если после прочтения последнего символа входного слова моделирующая программа оказывается в некоторой строке $j \in F$, то мы считаем, что программа-автомат принимает (допускает) заданное слово – т. е. даёт для данного входа ответ «да». Если же $j \in \{0, 1, 2, \ldots, m-1\} - F$, то программа-автомат *не* принимает входное слово – т. е. ответом является «нет». Множество L всех слов, допускаемых программой-автоматом, является языком, допускаемым этим автоматом. Иными словами, конечный автомат решает проблему принадлежности (принятия) – проблему (Σ, L) для распознаваемого языка L.

Для иллюстрации данного выше описания конечного автомата рассмотрим следующую программу (конечный автомат) A над алфавитом Σ_{bool}.

0: `if` *input* $= 1$ `then goto` 1 `else goto` 2

1: `if` *input* $= 1$ `then goto` 0 `else goto` 3

2: `if` *input* $= 0$ `then goto` 0 `else goto` 3

3: `if` *input* $= 0$ `then goto` 1 `else goto` 2

Пусть $F = \{0, 3\}$. Моделирующая программа автомата A работает со входным словом 1011 следующим образом. Она начинает работу в строке 0, и после прочтения первого символа 1 входного слова переходит на строку 1. После этого она читает символ 0 в строке 1 и переходит на строку 3. В строке 3 она читает 1 и переходит на строку 2. Наконец, после прочтения последнего символа 1 входного слова она снова переходит на строку 3. Вычисления закончены – и, поскольку $3 \in F$, мы считаем, что слово 1011 принимается (принадлежит рассматриваемому языку).

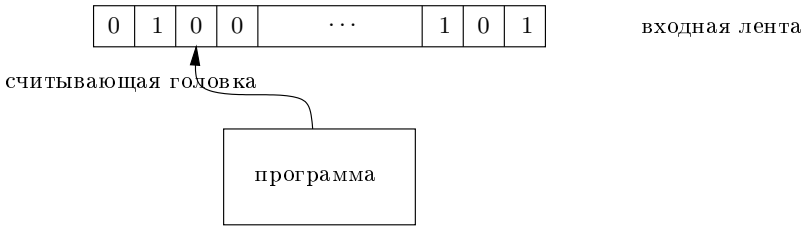


Рис. 3.1.

Рисунок 3.1 представляет собой схему, часто используемую для рассмотрения конечного автомата в качестве специальной вычислительной модели. На нём мы видим три главных компоненты этой модели – **программу**, записанную в памяти, **ленту** (как средство ввода), которая содержит входное слово, и **считывающую головку**,

которая может перемещаться по входной ленте только слева направо.[3] (Входная) лента может рассматриваться как линейная память для ввода; она состоит из последовательности ячеек, каждая из которых является единицей памяти и содержит в точности один символ алфавита. Фактическая лента всегда имеет ровно столько ячеек, сколько символов содержится во входном слове.

Однако класс программ, описанных выше, для задания конечных автоматов применяется нечасто – поскольку эти программы используют команду `goto`, и, следовательно, обычно не являются хорошо структурированными.[4] Поэтому такой способ моделирования конечных автоматов не очень удобен и, в большинстве случаев, является довольно громоздким.

А идея применения «дружественного интерфейса» для определения конечного автомата основана на следующем варианте «визуализации» рассматриваемых нами программ. Каждой программе (конечному автомату) A мы ставим в соответствие помеченный ориентированный граф $G(A)$. Этот граф имеет в точности столько вершин, сколько строк содержит программа A. Каждая вершина графа $G(A)$ соответствует одной строке автомата A, а номер этой строки является пометкой рассматриваемой вершины. И если программа A переходит из строки i в строку j, читая входной символ b, то $G(A)$ содержит ориентированную дугу (i, j), помеченную таким b. Поскольку наши специальные программы (программы без переменных) содержат команды `goto` для каждого символа $a \in \Sigma$ в каждой строке[5], то каждая вершина графа $G(A)$ имеет выходную степень[6], равную $|\Sigma|$.

На рис. 3.2 показан граф $G(A)$, соответствующий вышеприведённой программе A. Вершины, соответствующие строкам 0 и 3 (принадлежащим F), нарисованы как двойные окружности – для того, чтобы отличить их от остальных вершин. А вершина, соответствующая входной строке 0, отмечена специальным указателем, маленькой входной стрелкой.

На основе такого графического представления программы без переменных разработаем стандарт представления конечных автоматов. При этом мы будем использовать и данное графическое представление – поскольку оно даёт чёткий, ясный, и в то же время однозначный способ описания автомата. Однако для изучения свойств конечных автоматов – например, для более понятного изложения некоторых доказательств – более удобно определение, приводимое ниже. Мы немного изменим стандартную терминологию, используемую в теории автоматов. Термин «строка программы», или, что то же самое, «вершина соответствующего графа», будет заменён на «состояние конечного автомата». (Ориентированные) дуги графа, соответствующие командам `goto`, будут описываться так называемой функцией переходов – которая выбирает новое состояние на основе текущего состояния и прочитываемого символа.

Обратим внимание на следующий важный факт. Из приведённого ниже определения конечного автомата следует *общая схема* – которая может быть применена для определения *любой* вычислительной модели. В это схеме определяется структура, кото-

[3] А схема *общих* вычислительных моделей дополнительно к этому содержит память, а также описание возможностей доступа к ней – для чтения, записи, удаления и добавления данных. Кроме того, общая модель может также содержать среду вывода.

[4] Здесь мы применяем терминологию теории программных схем.

[5] А любую строку можно рассматривать в качестве команды выбора, заданной *для всех* символов входного алфавита.

[6] Выходная степень вершины ориентированного графа G – это число дуг, выходящих из неё, т. е. число $|\{(v, u) \mid (v, u) - \text{дуга орграфа G}\}|$.

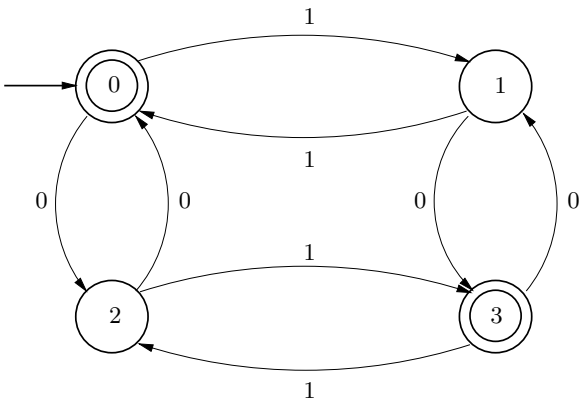


Рис. 3.2.

рая обеспечивает точное описание каждого объекта моделируемого класса алгоритмов. Смысл этой структуры объясняется в следующем порядке. Во-первых, определяется конфигурация – полное описание текущей ситуации (текущего обобщённого состояния[7]) рассматриваемой машины. Затем определяется шаг (вычисления), т. е. переход из одной конфигурации в другую – это делается на основе выполнения текущей команды (элементарной операции) машины (программы). Всё вычисление рассматривается как последовательность таких шагов. Наконец, на основе понятия «вычисление» мы определяем результат вычисления программы для любого заданного входа.

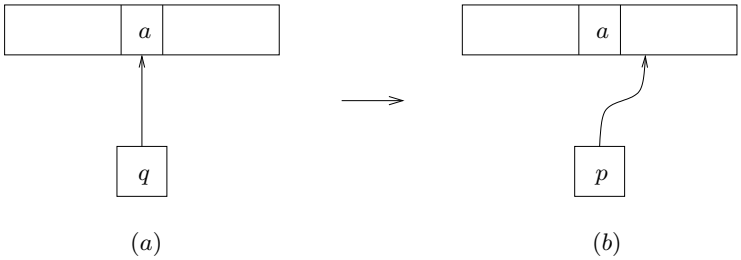


Рис. 3.3.

Определение 3.1. *(Детерминированный)* **конечный автомат (КА)** *– это пятёрка* $M = (Q, \Sigma, \delta, q_0, F)$*, где:*

- Q *– конечное непустое множество* **состояний**
  {в рассмотренном представлении – множество всех строк программы без переменных};

[7] В отличие от конфигураций (обобщённых состояний), «просто» состояния автоматов ниже иногда будут называться «внутренними состояниями». Такая же терминология ниже будет применяться и для состояний машин Тьюринга. По-видимому, такая терминология, не принятая в литературе на русском языке, является достаточно удобной. *(Прим. перев.)*

- Σ – *некоторый алфавит, называемый* **входным алфавитом** *программы* M {*т. е. возможные входы – это все слова над алфавитом* Σ };
- $q_0 \in Q$ – *инициальное (стартовое) состояние* {*в нашем случае – строка* 0 *программы без переменных*};
- $F \subseteq Q$ – **множество всех допускающих состояний**[8];
- δ – *функция, действующая из* $Q \times \Sigma$ *в* Q, *называемая* **функцией переходов** *программы* M {$\delta(q, a) = p$ *означает, что при чтении символа* a *программа* M *переходит из состояния* q *в состояние* p – *см. рис. 3.3.*}.

Конфигурация *программы* M – *произвольный элемент вида* $Q \times \Sigma^*$.
{*Достижение программой* M *конфигурации* $(p, w) \in Q \times \Sigma^*$ *означает, что* M *находится в состоянии* p, *и при этом ей осталось прочесть суффикс* w *некоторого входного слова. Т. е. считывающая головка обозревает первый символ слова* w – *это показано на рис. 3.3.*}
Конфигурация $(q_0, x) \in \{q_0\} \times \Sigma^*$ *называется* **стартовой конфигурацией программы** M **над входом** x.
{*Именно в стартовой конфигурации* (q_0, x) *должно начинаться вычисление (работа) программы* M *над входом* x.}
Любая конфигурация вида $Q \times \{\lambda\}$ *называется* **финальной конфигурацией**.
Шаг (вычисления) *программы* M – *это заданное на множестве конфигураций бинарное отношение вида*

$$|\overline{}_M \ \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*),$$

определённое следующим образом:

$$(q, w) \,|\overline{}_M \, (p, x) \Leftrightarrow w = ax,\ a \in \Sigma \text{ и } \delta(q, a) = p\,.$$

{*Т. е. определённый нами шаг вычисления соответствует применению функции переходов к рассматриваемой конфигурации: программа* M *находится в состоянии* q *и читает со входной ленты символ* a.}
Вычисление C *программы* M – *это конечная последовательность конфигураций* $C = C_0, C_1, \ldots, C_n$, *такая что для всех* i *из диапазона* $0 \leq i \leq n-1$ *выполнено отношение* $C_i \,|\overline{}_M \, C_{i+1}$.
{*Поэтому вычисление можно рассматривать как некоторую последовательность шагов программы* M. *Иногда нам даже будет удобнее описывать некоторое вычисление как* $C_0 \,|\overline{}_M \, C_1 \,|\overline{}_M \, \ldots \,|\overline{}_M \, C_n$ – *а не в виде более простой записи* $C_0, C_1, \ldots, C_n$.}
C *называется* **вычислением программы** M **над входом** $x \in \Sigma^*$, *если* $C_0 = (q_0, x)$ *и* $C_n \in Q \times \{\lambda\}$.

[8] Некоторые авторы используют другое название: «финальные состояния» вместо «допускающие состояния»; однако это название может ввести читателя в заблуждение. Во-первых, вычисление может закончиться в любом состоянии, поэтому прилагательное «финальное» не совсем соответствует действительности. Во-вторых – и это ещё более важно – значения понятий «допускающее состояние» и «финальное состояние» в более общих вычислительных моделях не совпадают (даже, в некоторых случаях, «не пересекаются»); основная из таких вычислительных моделей – машины Тьюринга. Таким образом, при использовании для конечных автоматов термина «финальное состояние» мы сильно изменяем (можно сказать – упрощаем) смысл этого термина.

{Т. е. вычисление над входом x должно начинаться в стартовой конфигурации (q_0, x) программы M над x и может заканчиваться только тогда, когда прочитаны все символы входного слова.}
Если $C_n \in F \times \{\lambda\}$, то будем говорить, что C является **допускающим вычислением** *программы M над x, или, что то же самое, что* $\boldsymbol{M}$ **допускает (принимает)** $\boldsymbol{x}$.
Если же $C_n \in (Q - F) \times \{\lambda\}$, то будем говорить, что C является **отклоняющим вычислением** *программы M над x, или, что то же самое, что* $\boldsymbol{M}$ **отклоняет**[9] x.
{Заметим, что M определяет ровно одно вычисление для любого входа $x \in \Sigma^*$.}
Язык $\boldsymbol{L(M)}$, допускаемый (принимаемый) $\boldsymbol{M}$, *определяется следующим образом:*

$$\begin{aligned} L(M) &= \{w \in \Sigma^* \mid \textit{вычисление программы } M \textit{ над } w \textit{ заканчивается} \\ &\qquad \textit{в некоторой финальной конфигурации } (q, \lambda) \textit{ с } q \in F\} \\ &= \{w \in \Sigma^* \mid M \textit{ принимает } w\}. \end{aligned}$$

Класс регулярных языков

$$\boldsymbol{\mathcal{L}(\mathrm{KA})} = \{L(M) \mid M \textit{ – некоторый } KA\}$$

– это множество всех языков, которые допускаются конечными автоматами. Каждый из языков L класса $\mathcal{L}(\mathrm{KA})$ называется **регулярным**.

Рассмотрим ещё раз программу A – теперь для иллюстрации приведённого определения конечного автомата. Формальное описание соответствующего конечного автомата – это пятёрка $M = (Q, \Sigma, \delta, q_0, F)$, где:

- $Q = \{q_0, q_1, q_2, q_3\}$;
- $\Sigma = \{0, 1\}$;
- $F = \{q_0, q_3\}$
- $\delta(q_0, 0) = q_2$, $\delta(q_0, 1) = q_1$, $\delta(q_1, 0) = q_3$, $\delta(q_1, 1) = q_0$,
- $\delta(q_2, 0) = q_0$, $\delta(q_2, 1) = q_3$, $\delta(q_3, 0) = q_1$, $\delta(q_3, 1) = q_2$.

Неформальное описание функции переходов δ приведено в табл. 3.1.

Таблица 3.1.

Состояние	Вход	
	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Другим неформальным описанием программы A является её графическое представление $G(A)$ – см. рис. 3.2. В дальнейшем в качестве стандартного представления

[9] Другие термины, использующиеся в русской литературе, – «отвергает», «не допускает», «не принимает». *(Прим. перев.)*

соответствующего конечного автомата мы будем использовать немного изменённый вариант; единственная его модификация – обозначение вершин с помощью имён состояний вместо натуральных чисел, нумерующих строки программы.[10]

Например, для рассматриваемой программы A мы получаем граф, приведённый на рис. 3.4.

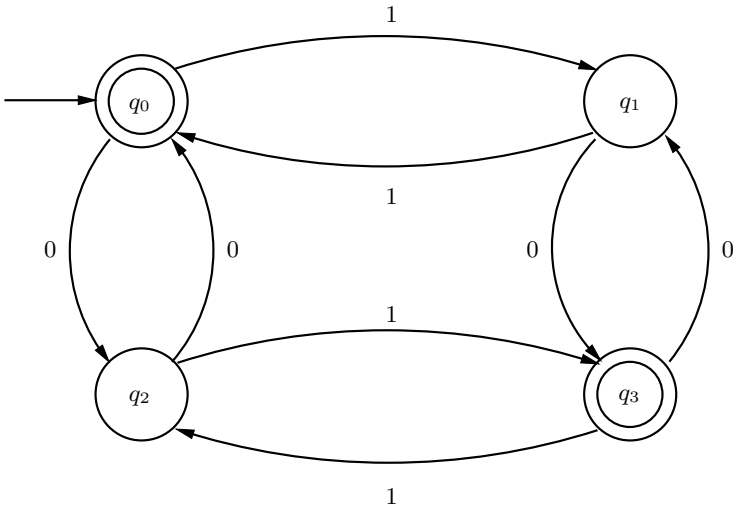


Рис. 3.4.

Вычисление автомата M над входом 1011 таково:

$$(q_0, 1011) \mid\!\!\overline{}_M\ (q_1, 011) \mid\!\!\overline{}_M\ (q_3, 11) \mid\!\!\overline{}_M\ (q_2, 1) \mid\!\!\overline{}_M\ (q_3, \lambda).$$

Поскольку $q_3 \in F$, мы получаем, что $1011 \in L(M)$, т. е. M допускает слово 1011.

В следующем определении вводятся обозначения, полезные для формальной работы с конечными автоматами – особенно для записи коротких, элегантных формулировок некоторых доказательств.

Определение 3.2. *Пусть $M = (Q, \Sigma, \delta, q_0, F)$ – некоторый конечный автомат. Определим бинарное отношение $\mid\!\!\overline{}^*_M$ как рефлексивно-транзитивное замыкание отношения «шаг» $\mid\!\!\overline{}_M$ этого автомата.*
Иными словами – условие $(q, w) \mid\!\!\overline{}^*_M (p, u)$ выполняется в случае, если либо $q = p$ и $w = u$, либо существует $k \in \mathbb{N}$, такое что:

- *$w = a_1 a_2 \ldots a_k u$, где $a_i \in \Sigma$ для $i = 1, 2, \ldots, k$;*
- *$\exists r_1, r_2, \ldots, r_{k-1} \in Q$, такие что*
  $(q, w) \mid\!\!\overline{}_M\ (r_1, a_2 \ldots a_k u) \mid\!\!\overline{}_M\ (r_2, a_3 \ldots a_k u) \mid\!\!\overline{}_M\ \cdots (r_{k-1}, a_k u) \mid\!\!\overline{}_M\ (p, u)$.

Для заданной функции δ определим функцию $\hat{\delta} : Q \times \Sigma^* \to Q$ следующим образом:

[10] Этот способ обозначения часто повышает ясность изложения (см., например, рис. 3.6 ниже) – поскольку выбранные соответствующим образом имена состояний могут давать больше информация, чем любой линейный порядок на множестве состояний.

- $\hat{\delta}(q, \lambda) = q$ *для всех* $q \in Q$,
- $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$ *для всех* $a \in \Sigma$, $w \in \Sigma^*$, $q \in Q$.

Смысл записи

$$(q, w) \,|\overset{*}{\underset{M}{-}}\, (p, u)$$

заключается в следующем. Вычисление программы M, начинающее в конфигурации (q, w), заканчивается в конфигурации (p, u). Равенство

$$\hat{\delta}(q, w) = p$$

означает, что:

- M начинает работу в состоянии q – с целью прочесть всё слово w;
- M завершает работу в состоянии p.

То есть

$$\hat{\delta}(q, w) = p \;\text{ равносильно }\; (q, w) \,|\overset{*}{\underset{M}{-}}\, (p, \lambda).$$

Следовательно, мы получаем такое эквивалентное описание языка $L(M)$:

$$\begin{aligned} L(M) &= \{w \in \Sigma^* \mid (q_0, w) \,|\overset{*}{\underset{M}{-}}\, (p, \lambda), \text{ где } p \in F\} \\ &= \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}. \end{aligned}$$

Теперь *неформально* определим, какой именно язык принимается конечным автоматом M, заданным на рис. 3.4, – т. е. язык $L(M)$. Для этого заметим, что вычисление M над словами с чётным [нечётным] числом символов 1 заканчивается либо в q_0 либо в q_2 [соответственно, либо в q_1 либо в q_3]. Аналогично, если число символов 0 во входном слове x чётно [нечётно], то $\hat{\delta}(q_0, x) \in \{q_0, q_1\}$ [$\hat{\delta}(q_0, x) \in \{q_2, q_3\}$]. Это наблюдение приводит к следующему утверждению.

Лемма 3.3. $L(M) = \{w \in \{0,1\}^* \mid |w|_0 + |w|_1 \equiv 0 \pmod 2\}$.

Доказательство. Сначала заметим, что каждый конечный автомат разделяет множество всех слов Σ^* на $|Q|$ классов

$$\begin{aligned} \mathrm{Kl[p]} &= \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) = p\}, \\ &= \{w \in \Sigma^* \mid (q_0, w) \,|\overset{}{\underset{M}{-}}\, (p, x)\}, \end{aligned}$$

и очевидно, что для всех $p, q \in Q$, $p \neq q$, выполнено следующее:

$$\bigcup_{p \in Q} \mathrm{Kl}[p] = \Sigma^* \;\text{ и }\; \mathrm{Kl}[p] \cap \mathrm{Kl}[q] = \emptyset.$$

Используя эту терминологию, мы записываем язык $L(M)$ в виде

$$L(M) = \bigcup_{p \in F} \mathrm{Kl}[p].$$

Иными словами, следующее бинарное отношение

$$x R_\delta y \Leftrightarrow \hat{\delta}(q_0, x) = \hat{\delta}(q_0, y),$$

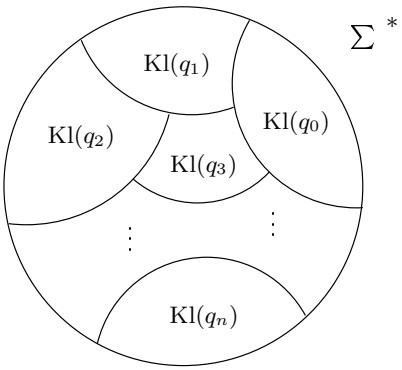


Рис. 3.5.

заданное на множестве Σ^*, является отношением эквивалентности – и при этом определяет конечное число классов эквивалентности $\mathrm{Kl}[p]$ (рис. 3.5).

Итак, удачный вариант задания языка $L(M)$ состоит в том, чтобы определить для рассматриваемого конечного автомата M множества $\mathrm{Kl}[q_0]$, $\mathrm{Kl}[q_1]$, $\mathrm{Kl}[q_2]$ и $\mathrm{Kl}[q_3]$. А для этого мы выдвигаем следующую гипотезу:

$$\begin{aligned}
&\mathrm{Kl}[q_0] = \{w \in \{0,1\}^* \mid |w|_0 \text{ и } |w|_1 \text{ чётные}\};\\
&\mathrm{Kl}[q_1] = \{w \in \{0,1\}^* \mid |w|_0 \text{ чётное, а } |w|_1 \text{ нечётное}\};\\
&\mathrm{Kl}[q_2] = \{w \in \{0,1\}^* \mid |w|_0 \text{ нечётное, а } |w|_1 \text{ чётное}\};\\
&\mathrm{Kl}[q_3] = \{w \in \{0,1\}^* \mid |w|_0 \text{ и } |w|_1 \text{ нечётные}\}.
\end{aligned}$$

А поскольку

$$\mathrm{Kl}[q_0] \cup \mathrm{Kl}[q_3] = \{w \in \{0,1\}^* \mid |w|_0 + |w|_1 \equiv 0 \pmod 2\},$$

утверждение леммы 3.3 является следствием этой гипотезы. Поэтому для завершения доказательства леммы 3.3 достаточно доказать сформулированную гипотезу; мы сделаем это индукцией по длине входного слова.

Базис индукции. Докажем предположение индукции для всех слов, длина которых не превышает 2.

$\hat{\delta}(q_0, \lambda) = q_0 \Rightarrow \lambda \in \mathrm{Kl}[q_0]$.
$\hat{\delta}(q_0, 1) = q_1 \Rightarrow 1 \in \mathrm{Kl}[q_1]$.
$\hat{\delta}(q_0, 0) = q_2 \Rightarrow 0 \in \mathrm{Kl}[q_2]$.
$(q_0, 00) \mid\!\overline{}_M\, (q_2, 0) \mid\!\overline{}_M\, (q_0, \lambda) \Rightarrow 00 \in \mathrm{Kl}[q_0]$.
$(q_0, 01) \mid\!\overline{}_M\, (q_2, 1) \mid\!\overline{}_M\, (q_3, \lambda) \Rightarrow 01 \in \mathrm{Kl}[q_3]$.
$(q_0, 10) \mid\!\overline{}_M\, (q_1, 0) \mid\!\overline{}_M\, (q_3, \lambda) \Rightarrow 10 \in \mathrm{Kl}[q_3]$.
$(q_0, 11) \mid\!\overline{}_M\, (q_1, 1) \mid\!\overline{}_M\, (q_0, \lambda) \Rightarrow 11 \in \mathrm{Kl}[q_0]$.

Итак, для всех слов длины 0, 1 и 2 предположение выполняется.

Шаг индукции. Пусть $i \geq 2$ – некоторое целое число. Считаем, что предположение индукции выполняется для всех $x \in \{0,1\}^*$, где $|x| \leq i$ – и нашей целью является доказательство того, что этот же факт выполнен и для всех слов длины $i+1$. Если

мы докажем шаг индукции для всех $i \geq 2$, то мы тем самым получим выполнение предположения индукции для всех слов множества $(\Sigma_{\text{bool}})^*$.

Итак, пусть w – произвольное слово языка $(\Sigma_{\text{bool}})^{i+1}$. Тогда w можно записать в виде $w = za$, где $z \in \Sigma^i$ и $a \in \Sigma$. Для значений $|z|_0$ и $|z|_1$ мы будем различать 4 возможных варианта.

(a) $|z|_0$ и $|z|_1$ – оба чётные.
(b) $|z|_0$ и $|z|_1$ – оба нечётные.
(c) $|z|_0$ чётное, а $|z|_1$ – нечётное.
(d) $|z|_0$ нечётное, а $|z|_1$ – чётное.

Рассмотрим каждый из этих вариантов.

(a) Вследствие выполнеия сделанного предположения для слова z (заметим, что $|z| = i$) мы получаем, что $\hat{\delta}(q_0, z) = q_0$, т. е. $z \in \mathrm{Kl}[q_0]$. Поэтому

$$\hat{\delta}(q_0, za) = \delta(\hat{\delta}(q_0, z), a) \underset{(\text{инд.})}{=} \delta(q_0, a) = \begin{cases} q_1, \text{ если } a = 1, \\ q_2, \text{ если } a = 0. \end{cases}$$

Поскольку $|z1|_0$ – чётное, а $|z1|_1$ – нечётное, результат $\hat{\delta}(q_0, z1) = q_1$ соответствует предположению индукции $z1 \in \mathrm{Kl}[q_1]$.
Поскольку $|z0|_0$ – нечётное, а $|z0|_1$ – чётное, полученный выше результат $\hat{\delta}(q_0, z0) = q_2$ согласуется с предположением индукции $z0 \in \mathrm{Kl}[q_2]$.

(b) Применяя предположение индукции для слова z, мы получаем, что $\hat{\delta}(q_0, z) = q_3$, т. е. $z \in \mathrm{Kl}[q_3]$.

$$\hat{\delta}(q_0, za) = \delta(\hat{\delta}(q_0, z), a) \underset{(\text{инд.})}{=} \delta(q_3, a) = \begin{cases} q_2, \text{ если } a = 1, \\ q_1, \text{ если } a = 0. \end{cases}$$

Этот результат также согласуется с предположением индукции – которое требует выполнения условий $z0 \in \mathrm{Kl}[q_1]$ и $z1 \in \mathrm{Kl}[q_2]$.

Варианты (c) и (d) аналогичны, и мы оставляем завершение доказательства читателю. □

Упражнение 3.4. Завершите доказательство предположения индукции в лемме 3.3, т. е. рассмотрите варианты (c) и (d).

Упражнение 3.5. Пусть использование обозначения $\hat{\delta}$ «запрещено». Перепешите доказательство леммы 3.3, используя обозначения $|^*_{\overline{M}}$ и $|_{\overline{M}}$ вместо $\hat{\delta}$.

Упражнение 3.6. Пусть $L = \{w \in (\Sigma_{\text{bool}})^* \mid |w|_0 \text{ нечётное}\}$. Опишите автомат M, определяющий язык $L(M) = L$, и докажите, что описанный автомат действительно является искомым.

Упражнение 3.7. Опишите конечные автоматы для языков $\emptyset, \Sigma^*$ и Σ^+ – для произвольного алфавита Σ. При этом необходимы как графическое представление, так и описание пятёрки с помощью определённого выше формализма.

Если представлено детальное описание некоторого конечного автомата A, то обычно можно привести и описание языка этого автомата $L(A)$ – причём без формального доказательства того, что язык действительно определяется заданным автоматом. При

этом, как правило, мы разделяем эти две задачи – описание автомата и формальное доказательство того, что он действительно определяет требуемый язык.[11] Однако при доказательстве леммы 3.3 мы рассмотрели очень важные приёмы, применяемые в процессе описания, в процессе проектирования конкретного конечного автомата. Хорошая стратегия (технология) проектирования автомата состоит в следующем. Во-первых, множество всех возможных входных слов (т. е. Σ^*) необходимо разделить на классы – в соответствии с некоторыми свойствами слов, формулируемыми на основе заданного языка. И, во-вторых, необходимо определить переходы между этими классами – на основе рассмотрения конкатенаций букв алфавита Σ, соответствующих таким переходам, и слов, входящих в эти классы.

Рассмотрим подобную технологию для разработки конечного автомата, задающего язык

$$U = \{w \in (\Sigma_{\text{bool}})^* \mid |w|_0 = 3 \text{ и } (|w|_1 \geq 2 \text{ или } |w|_1 = 0)\}.$$

Отметим, что если $|w|_0 = 3$, то для любого слова w каждый автомат B, задающий язык $L(B) = U$, должен различать случаи

$$|w|_0 = 0,\ |w|_0 = 1,\ |w|_0 = 2,\ |w|_0 = 3 \ \text{ и } \ |w|_0 \geq 4$$

– т. е. B должен считать число символов 0 (до значения 4) в префиксах всех входных слов. Одновременно с этим автомат должен считать число вхождений символа 1 – для того, чтобы иметь возможность различать следующие три случая:

$$|w|_1 = 0,\ |w|_1 = 1 \text{ и } |w|_1 \geq 2.$$

Следовательно, надо рассмотреть такое множество состояний (т. е. разделение множества слов $\{0,1\}^*$ на классы):

$$Q = \{q_{i,j} \mid i \in \{0,1,2,3,4\}, j \in \{0,1,2\}\}.$$

Смысл этих состояний следующий. Для всех $i \in \{0,1,2,3\}$ и $j \in \{0,1\}$:

$$\text{Kl}[q_{i,j}] = \{w \in (\Sigma_{\text{bool}})^* \mid |w|_0 = i \text{ и } |w|_1 = j\};$$

$$\text{Kl}[q_{i,2}] = \{w \in (\Sigma_{\text{bool}})^* \mid |w|_0 = i \text{ и } |w|_1 \geq 2\};$$

$$\text{Kl}[q_{4,j}] = \{w \in (\Sigma_{\text{bool}})^* \mid |w|_0 \geq 4 \text{ и } |w|_1 = j\};$$

$$\text{Kl}[q_{4,2}] = \{w \in (\Sigma_{\text{bool}})^* \mid |w|_0 \geq 4 \text{ и } |w|_1 \geq 2\}.$$

Очевидно, что инициальным состоянием автомата B является $q_{0,0}$. Функцию переходов B можно точно определить, опираясь на смысл состояний $q_{i,j}$ – как мы делали ранее, при рассмотрении примера на рис. 3.6.[12] Мы получаем, что

$$U = \text{Kl}[q_{3,0}] \cup \text{Kl}[q_{3,2}],$$

после чего выбираем $F = \{q_{3,0}, q_{3,2}\}$.

[11] Заметим, что эта ситуация аналогична разработке «обычных» программ – поскольку конечные автоматы фактически представляют собой специальный язык программирования. Действительно, из-за очень большого объёма соответствующей работы мы практически никогда не рассматриваем формальное доказательство корректности созданной программы.

[12] Точнее – опираясь на определения классов $Kl[q_{i,j}]$.

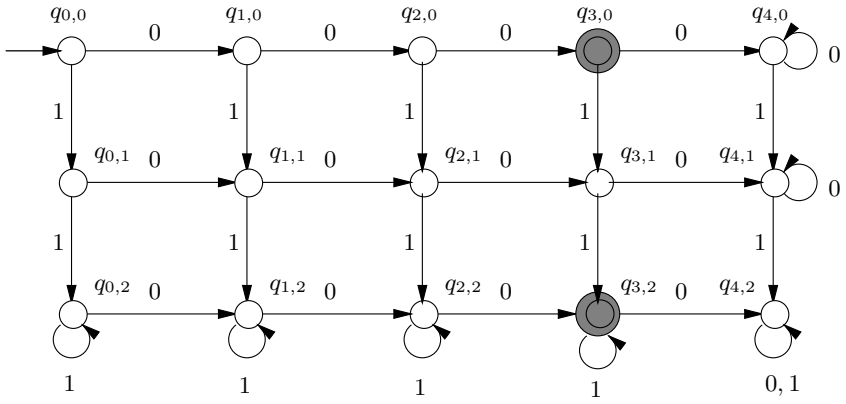


Рис. 3.6.

Упражнение 3.8. Рассмотрим конечный автомат B, приведённый на рис. 3.6.

- Докажите, что $L(B) = U$.
- Опишите автомат A, такой что $L(A) = U$, и при этом A имеет меньше состояний, чем B. Определите $\mathrm{Kl}[q]$ для каждого состояния q автомата A.

Упражнение 3.9. Определите конечный автомат для каждого из следующих регулярных языков:

- $\{w \in \{0,1,2\}^* \mid w = 002122x,\ x \in (\Sigma_{\mathrm{bool}})^*\}$;
- $\{w \in \{a,b,c\}^* \mid w = xabcabc,\ x \in \{a,b,c\}^*\}$;
- $\{w \in \{a,b,c\}^* \mid w = xaabby,\ x,y \in \{a,b,c\}^*$;
- $\{w \in \{0,1\}^* \mid |w|_0 \equiv 1 \pmod 3$ и $w = x111y$ для $x,y \in \{0,1\}^*\}$;
- $\{abbxb^3y \mid x,y \in \{a,b\}^*\}$;
- $\{w \in \{a,b\}^* \mid w = abbz$ и $w = ub^3v$ для некоторых $z,u,v \in \{a,b\}^*\}$.

(Достаточно дать графическое представление этих автоматов.) Опишите класс $\mathrm{Kl}[q]$ для каждого состояния q каждого из этих автоматов.

3.3 Моделирование конечных автоматов

Термин «моделирование» в теоретической информатике употребляется очень часто. Однако, несмотря на всю его важность, строгое формальное определение этого термина никто даже не пытался ввести. Причина заключается в том, что в пределах различных структур у этого термина имеются различные интерпретации.

Самое детальное определение моделирования вычисления требует, чтобы каждый (элементарный) шаг этого вычисления имитировался в точности одним шагом работы модели; немного ослабленное требование разрешает несколько шагов работы модели. Но можно также моделировать вычисления без имитации каждого шага – накладывая при этом дополнительное требование, заключающееся в достижении моделируемым вычислением некоторых важных конфигураций. А наиболее общее определение требует лишь необходимой зависимости между входными и выходными данными – и при

этом не имеет значения, как именно вычислены эти выходные данные на основе соответствующих входных.

В этом разделе мы описываем моделирование в узком смысле этого слова – путём создания конечного автомата, который пошагово моделирует одновременные вычисления (одновременную работу) двух других конечных автоматов.

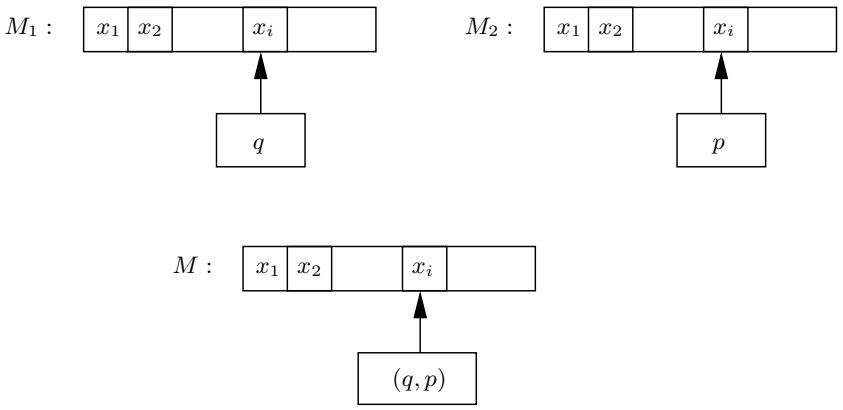


Рис. 3.7.

Лемма 3.10. *Пусть Σ – некоторый алфавит, а $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ и $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ – некоторые конечные автоматы. Тогда для каждой из операций $\odot \in \{\cup, \cap, -\}$ существует конечный автомат M, такой что*

$$L(M) = L(M_1) \odot L(M_2).$$

Доказательство. Идея доказательства состоит в построении такого автомата M, который моделирует одновременную работу обоих заданных автоматов, M_1 и M_2, над одним и тем же словом.[13]

Сама идея моделирования проста. Состояния автомата M суть пары (q, p), где q – состояние M_1, а p – состояние M_2. Первый элемент такой пары должен быть равен q тогда и только тогда, когда M_1 находится в состоянии q. Аналогично, второй элемент пары (состояния автомата M) должен быть равен p тогда и только тогда, когда M_2 находится в состоянии p. (См. рис. 3.7).

Представленное далее формальное доказательство – как и обычно в подобных случаях – можно разделить на следующие два этапа. Сначала мы дадим формальное описание автомата M – а потом докажем, что M действительно моделирует оба автомата, M_1 и M_2.

Описание M. Положим $M = (Q, \Sigma, \delta, q_0, F_\odot)$, где:

- $Q = Q_1 \times Q_2$;

[13] Более того, в данной ситуации каких-либо иных вариантов просто не может существовать – поскольку M может считывать входное слово лишь один раз, и, следовательно, M не может сначала моделировать M_1, а потом M_2.

- $q_0 = (q_{01}, q_{02})$;
- для всех $q \in Q_1$, $p \in Q_2$ и $a \in \Sigma$ $\quad \delta((q,p),a) = (\delta_1(q,a), \delta_2(p,a))$;
- если $\odot = \cup$, то $F = F_1 \times Q_2 \cup Q_1 \times F_2$
  {по крайней мере один из автоматов, M_1 или M_2, допускает входное слово, т. е. останавливается в допускающем состоянии};
  если $\odot = \cap$, то $F = F_1 \times F_2$
  {оба автомата допускают входное слово};
  если $\odot = -$, то $F = F_1 \times (Q_2 - F_2)$
  {M_1 должен допускать входное слово, а M_2 должен его не допускать}.

Доказательство утверждения $L(M) = L(M_1) \odot L(M_2)$. Чтобы доказать лемму для каждой из операций $\odot \in \{\cup, \cap, -\}$, достаточно показать выполнение равенства

$$\hat{\delta}((q_{01}, q_{02}), x) = (\hat{\delta}_1(q_{01}, x), \hat{\delta}_2(q_{02}, x)) \tag{3.1}$$

для всех $x \in \Sigma^*$. Докажем это равенство индукцией по $|x|$.

Базис индукции. Для $x = \lambda$ выполнение условия (3.1) очевидно.

Шаг индукции. Для каждого натурального i докажем, что выполнение условия (3.1) для всех слов $x \in \Sigma^*$, имеющих длину $|x| \leq i$, влечёт выполнение этого же условия для всех слов $w \in \Sigma^{i+1}$ – т. е. (3.1) с заменой x на w.

Пусть w – произвольное слово из Σ^{i+1}. Мы можем представить w как

$$w = za, \text{ где } z \in \Sigma^i \text{ и } a \in \Sigma.$$

Согласно определению $\hat{\delta}$, получаем

$$\begin{aligned}
\hat{\delta}((q_{01}, q_{02}), w) &= \hat{\delta}((q_{01}, q_{02}), za) \\
&= \delta(\hat{\delta}((q_{01}, q_{02}), z), a) \\
&= \delta((\hat{\delta}_1(q_{01}, z), \hat{\delta}_2(q_{02}, z)), a) \\
&\underset{(3.1)}{=} (\delta_1(\hat{\delta}_1(q_{01}, z), a), \delta_2(\hat{\delta}_2(q_{02}, z), a)) \\
&\underset{\text{Опр.}\delta}{=} (\hat{\delta}(q_{01}, za), \hat{\delta}(q_{02}, za)) \\
&= (\hat{\delta}(q_{01}, w), \hat{\delta}(q_{02}, w)).
\end{aligned}$$

□

Упражнение 3.11. Пусть $L \subseteq \Sigma^*$ – некоторый регулярный язык. Докажите, что $L^{\complement} = \Sigma^* - L$ – также регулярный язык.

3.4 Доказательства неразрешимости

Для доказательства того факта, что некоторый конкретный язык L *не* является регулярным, достаточно показать, что не существует ни одного конечного автомата, допускающего L. Обычно доказательство неразрешимости некоторой заданной проблемы в определённом классе алгоритмов – т. е. доказательство того, что никакая программа из этого класса не решает данную проблему – относится к самым трудным задачам теоретической информатики.

Такие доказательства называют доказательствами неразрешимости (несуществования).[14] В отличие от конструктивных доказательств – в которых мы доказываем существование некоторого объекта с нужными нам свойствами, просто создавая, конструируя, явно указывая такой объект (например, мы создаём конечный автомат M с четырьмя состояниями для некоторого заданного регулярного языка) – мы не можем доказывать факт *не*существования объекта с необходимыми свойствами среди бесконечного набора кандидатов (например, среди всех конечных автоматов), просто проверяя выполнение необходимого свойства для каждого из них. Доказывая несуществование объекта с данными свойствами в бесконечном классе кандидатов, мы должны продемонстрировать своё глубокое понимание тех свойств каждого кандидата из этого класса, которые противоречат необходимым свойствам требуемого объекта.

Поскольку класс конечных автоматов – это класс программ с очень сильными ограничениями, доказательства несуществования типа «не существует ни одного конечного автомата, допускающего заданный язык L» являются относительно простыми. Воспользуемся этим фактом для несложного введения в методологию создания подобных доказательств неразрешимости (несуществования).

Итак, мы знаем, что конечные автоматы получили своё название потому, что их единственно возможная память[15] – это текущеее состояние (т. е. номер текущей строки программы). Отсюда следует вывод: если автомат A, прочитав два разных слова x и y, достигает одного и того же состояния (т. е. если $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$), то A больше не может отличить x от y. Формально это означает, что равенство

$$\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$$

подразумевает для всех $z \in \Sigma^*$ выполнение и другого равенства:

$$\hat{\delta}_A(q_0, xz) = \hat{\delta}_A(q_0, yz).$$

Сформулируем это важное свойство конечных автоматов в виде следующей леммы.

Лемма 3.12. *Пусть $A = (Q, \Sigma, \delta_A, q_0, F)$ – некоторый конечный автомат. Пусть также для некоторых $x, y \in \Sigma^*$, таких что $x \neq y$, и для некоторого состояния $p \in Q$ выполнены условия*

$$(q_0, x) \vdash_A^* (p, \lambda) \quad \text{и} \quad (q_0, y) \vdash_A^* (p, \lambda)$$

(т. е. $\hat{\delta}_A(q_0, x) = \hat{\delta}_A(q_0, y) = p$, или, что то же самое, $x, y \in \mathrm{Kl}[p]$). Тогда для каждого $z \in \Sigma^*$ существует некоторое состояние $r \in Q$, такое что $xz, yz \in \mathrm{Kl}[r]$, и, соответственно,

$$xz \in L(M) \Longleftrightarrow yz \in L(M).$$

Доказательство. Существование вычислений

$$(q_0, x) \vdash_A^* (p, \lambda) \text{ и } (q_0, y) \vdash_A^* (p, \lambda)$$

автомата A над входными словами x и y подразумевает для каждого $z \in \Sigma^*$ существование следующих его вычислений над словами xz и yz:

[14] В английском варианте «nonexistence» – что может быть воспринято как игра слов, поскольку одним из важнейших значений слова «existence» является философское понятие «бытие». *(Прим. перев.)*

[15] Или – единственная возможность хранения информации.

$$(q_0, xz) \vdash_A^* (p, z) \quad \text{и} \quad (q_0, yz) \vdash_A^* (p, z).$$

Если $r = \hat{\delta}_A(p, z)$ (т. е. если $(p, z) \vdash_A^* (r, \lambda)$ – вычисление автомата A над словом z, начинающееся в состоянии p), то вычисление автомата A над словом xz – следующее:

$$(q_0, xz) \vdash_A^* (p, z) \vdash_A^* (r, \lambda),$$

а вычисление A над yz –

$$(q_0, yz) \vdash_A^* (p, z) \vdash_A^* (r, \lambda).$$

Если $r \in F$, то оба слова, xz и yz, принадлежат языку $L(A)$. А если $r \notin F$ – то наоборот, оба этих слова не принадлежат $L(A)$. □

Лемма 3.12 формулирует такое общее свойство всех детерминированных вычислительных моделей. Если детерминированная машина (детерминированный алгоритм) достигает одной и той же конфигурации[16] при вычислениях над двумя различными входами, то любые возможные оставшиеся вычисления в обоих случаях идентичны. Следствием этого факта в случае проблемы принадлежности является то, что оба входа или одновременно принимаются, или одновременно отклоняются.

Мы можем легко применить лемму 3.12 для доказательства того факта, что некоторые конкретные языки не являются регулярными. Проиллюстрируем это на примере языка

$$L = \{0^n 1^n \mid n \in \mathbb{N}_0\}.$$

Интуитивно ясно, что язык L «слишком сложен» для любого конечного автомата – поскольку для того, чтобы сравнить число нулей с числом единиц, необходимо прежде всего запоминать число нулей (т. е. фактически *посчитать* их). Однако число нулей в префиксе 0^n может быть сколь угодно большим – а поскольку любой конечный автомат имеет ограниченный размер (т. е. конечное число состояний), то никакой автомат не может посчитать число нулей во входных словах. Осталось привести формальное доказательство того, что такой подсчёт необходим для принятия языка L. Итак, докажем от противного, что $L \notin \mathcal{L}(\text{КА})$.

Предположим, что $L \in \mathcal{L}(\text{КА})$. Пусть $A = (Q, \Sigma_{\text{bool}}, \delta_A, q_0, F)$ – некоторый конечный автомат, такой что $L(A) = L$. Рассмотрим слова

$$0^1,\ 0^2,\ 0^3,\ \ldots,\ 0^{|Q|+1}.$$

Поскольку число этих слов равно $|Q| + 1$, существуют числа $i, j \in \{1, 2, \ldots, |Q| + 1\}$, такие что $i < j$ и

$$\hat{\delta}_A(q_0, 0^i) = \hat{\delta}_A(q_0, 0^j).$$

Согласно лемме 3.12,

$$0^i z \in L \Leftrightarrow 0^j z \in L$$

для всех $z \in (\Sigma_{\text{bool}})^*$. Однако последнее неверно – поскольку для $z = 1^i$

$$0^i 1^i \in L, \text{ но } 0^j 1^i \notin L.$$

[16] При этом конфигурация рассматривается как полное, обобщённое описание состояния машины, включающее не только само состояние, но и доступную к текущему моменту часть ввода.

Упражнение 3.13. Докажите на основе леммы 3.12, что следующие языки не принадлежат классу $\mathcal{L}(\mathrm{KA})$:

- $\{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$;
- $\{a^n b^m c^n \mid n, m \in \mathbb{N}_0\}$;
- $\{w \in \{0,1,\#\}^* \mid w = x\#x \text{ для некоторого } x \in \{0,1\}^*\}$;
- $\{x1y \in \{0,1\}^* \mid |x| = |y|\}$.

Для того, чтобы продемонстрировать ещё один – тоже несложный – метод доказательства нерегулярности конкретных языков, можно найти некоторые поддающиеся простой проверке свойства (условия), которым удовлетворяет *каждый* регулярный язык. И если некоторый язык *не* удовлетворяет какому-либо из этих условий – то можно сразу заключить, что этот язык нерегулярен. Далее мы приводим два метода проверки условий вида $L \notin \mathcal{L}(\mathrm{KA})$.

Первый из них называется основан на т. н. «разрастании»[17] – т. е. на следующей идее. Если

$$(p, x) \vdash_A^* (p, \lambda),$$

для некоторых состояния p и слова x, то

$$(p, x^i) \vdash_A^* (p, \lambda)$$

для всех натуральных i (рис. 3.8). Это означает, что автомат A не может запомнить, как много раз он уже прочёл слово x – и, следовательно, A не может сделать различие между словами с различным числом повторений x. Поэтому если $\hat{\delta}_A(q_0, y) = p$ для некоторого $y \in \Sigma^*$, и при этом $\hat{\delta}_A(p, z) = r$ для некоторого $z \in \Sigma^*$ (рис. 3.8), то

$$(q_0, yx^i z) \vdash_A^* (p, x^i z) \vdash_A^* (p, z) \vdash_A^* (r, \lambda)$$

– вычисление автомата A над словом $yx^i z$ для всех $i \in \mathbb{N}_0$, т. е.

$$\{yx^i z \mid i \in \mathbb{N}_0\} \subseteq \mathrm{Kl}[r]$$

для некоторого $r \in Q$. Это означает, что A либо принимает все слова типа $yx^i z$ для любого $i \in \mathbb{N}_0$ (если $r \in F$), либо не принимает ни одного такого слова (если $r \in Q - F$).

Лемма 3.14 (о разрастании для регулярных языков).
Для каждого регулярного языка L существует константа $n_0 \in \mathbb{N}$, такая что каждое слово $w \in \Sigma^*$ с $|w| \geq n_0$ может быть записано в виде

$$w = yxz,$$

причём:

(a) $|yx| \leq n_0$;
(b) $|x| \geq 1$;
(c) *либо* $\{yx^k z \mid k \in \mathbb{N}_0\} \subseteq L$, *либо* $\{yx^k z \mid k \in \mathbb{N}_0\} \cap L = \emptyset$.

[17] Или, в других переводах – «накачивании». Соответственно, приводимая далее лемма 3.14 называется «леммой о разрастании» или «леммой о накачке». *(Прим. перев.)*

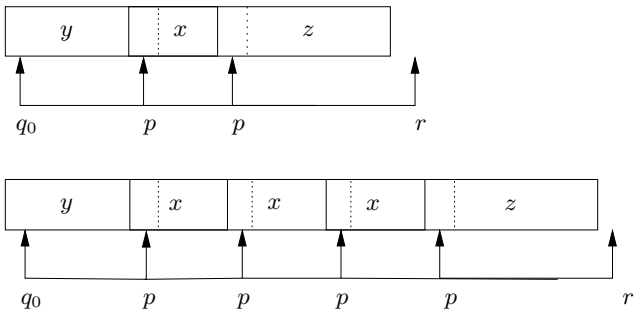


Рис. 3.8.

Доказательство. Пусть $L \subseteq \Sigma^*$ – некоторый регулярный язык. Тогда существует конечный автомат $A = (Q, \Sigma, \delta_A, q_0, F)$, такой что $L(A) = L$. Положим

$$n_0 = |Q|.$$

Пусть w – произвольное слово над алфавитом Σ, имеющее длину $|w| \geq n_0$. Это слово может быть записано в виде

$$w = w_1 w_2 \dots w_{n_0} u,$$

где $w_i \in \Sigma$ для $i = 1, \dots, n_0$ и $u \in \Sigma^*$.

Рассмотрим такое вычисление автомата A над словом $w_1 w_2 \dots w_{n_0}$:

$$\begin{aligned}(q_0, w_1 w_2 w_3 \dots w_{n_0}) &\vdash_A (q_1, w_2 w_3 \dots w_{n_0}) \\ &\vdash_A (q_2, w_3 \dots w_{n_0}) \vdash_A \cdots \vdash_A (q_{n_0-1}, w_{n_0}) \vdash_A (q_{n_0}, \lambda).\end{aligned} \tag{3.2}$$

Всего имеется $n_0 + 1$ состояний $q_0, q_1, q_2, \dots, q_{n_0}$ – в $n_0 + 1$ конфигурациях этого вычисления. Поскольку $|Q| = n_0$, существуют такие $i, j \in \{0, 1, \dots, n_0\}$, $i < j$, что $q_i = q_j$. Следовательно, вычисление (3.2) может быть записано в виде

$$(q_0, w_1 \dots w_{n_0}) \vdash_A^* (q_i, w_{i+1} \dots w_{n_0}) \vdash_A^* (q_i, w_{j+1} \dots w_{n_0}) \vdash_A^* (q_{n_0}, \lambda). \tag{3.3}$$

Положим

$$y = w_1 \dots w_i, \ x = w_{i+1} \dots w_j \ \text{ и } \ z = w_{j+1} \dots w_{n_0} u.$$

Очевидно, что $w = yxz$. Проверим, что условия (a), (b) и (c) выполнины для указанного нами разделения слова w на три подслова y, x, и z.

(a) $yx = w_1 \dots w_i w_{i+1} \dots w_j$ – и, следовательно, $|yx| = j \leq n_0$.
(b) Вследствие $i < j$ и $|x| = j - i$ мы получаем, что $x \neq \lambda$ (т. е. $|x| \geq 1$).
(c) Если мы в вычислении (3.3) заменим обозначение y на $w_1 \dots w_i$, а x на $w_{i+1} \dots w_j$, то вычисление автомата A над словом yx может быть записано следующим образом:

$$(q_0, yx) \vdash_A^* (q_i, x) \vdash_A^* (q_i, \lambda). \tag{3.4}$$

Следовательно, существование вычисления (3.3) показывает, что для любого $k \in \mathbb{N}_0$

$$(q_i, x^k) \overset{*}{\underset{A}{\vdash}} (q_i, \lambda).$$

Поэтому для любого $k \in \mathbb{N}_0$

$$(q_0, yx^k z) \overset{*}{\underset{A}{\vdash}} (q_i, x^k z) \overset{*}{\underset{A}{\vdash}} (q_i, z) \overset{*}{\underset{A}{\vdash}} (\hat{\delta}_A(q_i, z), \lambda)$$

является вычислением автомата A над словом $yx^k z$. Таким образом, для любого $k \in \mathbb{N}_0$ данное вычисление заканчивается в одном и том же состоянии $\hat{\delta}_A(q_i, z)$.

Если $\hat{\delta}_A(q_i, z) \in F$, то A принимает все слова языка $\{yx^k z \mid k \in \mathbb{N}_0\}$.

А если $\hat{\delta}_A(q_i, z) \notin F$, то A не принимает ни одного слова этого языка.

Последний факт завершает доказательство леммы 3.14. □

Как применить лемму 3.14 для доказательства того, что некоторый конкретный язык не является регулярным? Для этого мы снова будем использовать язык $L = \{0^n 1^n \mid n \in \mathbb{N}_0\}$. Наша цель – показать, что этот язык не обладает свойством регулярных языков, сформулированным в лемме о разрастании. Сделаем это путём доказательства от противного.

Итак, предположим, что язык L является регулярным. Тогда существует некоторая константа n_0, обладающая свойствами, сформулированными в лемме 3.14. Это означает, что для каждого слова w длины не менее n_0 должно существовать разбиение этого слова на части со свойствами (a), (b) и (c). Поэтому для доказательства условия $L \notin \mathcal{L}(\text{КА})$ достаточно найти слово w с длиной $|w| \geq n_0$, такое что ни одно из его разбиений на 3 части *не* обладает одновременно всеми тремя свойствами (a), (b) и (c).

В нашем случае достаточно выбрать слово

$$w = 0^{n_0} 1^{n_0}.$$

Отметим, что $|w| = 2n_0 \geq n_0$, и поэтому можно рассматривать представление слова w в виде $w = yxz$; предположим, что оно обладает требуемыми тремя свойствами. Согласно (a), мы получаем неравенство $|yx| \leq n_0$ – и поэтому $y = 0^l$ и $x = 0^m$ для некоторых $l, m \in \mathbb{N}_0$. Вследствие (b), $m \neq 0$. Поскольку

$$w = 0^{n_0} 1^{n_0} = yxz \in L \text{ и } yxz \in \{yx^k z \mid k \in \mathbb{N}_0\},$$

свойство (c) влечёт

$$\{yx^k z \mid k \in \mathbb{N}_0\} = \{0^{n_0 - m + km} 1^{n_0} \mid k \in \mathbb{N}_0\} \subseteq L.$$

А это является противоречием – поскольку

$$yx^0 z = yz = 0^{n_0 - m} 1^{n_0} \notin L.$$

(Более того, $0^{n_0} 1^{n_0}$ – это единственное слово языка $\{yx^k z \mid k \in \mathbb{N}_0\}$, которое принадлежит и языку L.) Таким образом, язык L не является регулярным.

Очень важный момент при использовании леммы о разрастании состоит в том, что мы имеем возможность выбора длинного слова w – поскольку эта лемма выполняется для всех достаточно длинных слов. Такой выбор особенно важен по следующим двум причинам.

Во-первых, сначала мы можем сделать «плохой» выбор – который в конечном итоге для доказательства условия $L \notin \mathcal{L}(\text{КА})$ окажется бесполезным. Для языка $L = \{0^n 1^n \mid n \in \mathbb{N}_0\}$ примером такого плохого выбора является слово

$$w = 0^{n_0} \notin L :$$

мы можем представить слово w как

$$w = yxz \quad \text{с} \quad y = 0,\ x = 0 \quad \text{и} \quad z = 0^{n_0 - 2}.$$

Очевидно, что для этого слова w лемма о разрастании выполняется – поскольку все слова языка

$$\{yx^k z \mid k \in \mathbb{N}_0\} = \{0^{n_0 - 1 + k} \mid k \in \mathbb{N}_0\}$$

не принадлежат L. Следовательно, все три свойства (a), (b) и (c) для слова $w = yxz$ выполнены.

Во-вторых, выбор действительно может помочь доказать, что для некоторого языка L условие $L \notin \mathcal{L}(\text{КА})$ выполнено – но при этом может оказаться всё же не столь удачным, как другие возможные варианты: а именно, иногда мы должны выполнить значительно больше работы для доказательства того факта, что *каждое* разделение выбранного нами слова действительно не удовлетворяет какому-либо из условий (a), (b) и (c), чем в случае какого-либо более удачного выбора. Например, рассмотрим слово

$$w = 0^{\lceil n_0/2 \rceil} 1^{\lceil n_0/2 \rceil}.$$

Это слово действительно может быть использовано для доказательства того, что язык $L = \{0^n 1^n \mid n \in \mathbb{N}_0\}$ не является регулярным – однако, рассматривая все возможные разбиения yxz слова w, мы должны проверить три следующих случая.[18]

(A) $y = 0^i$, $x = 0^m$, $z = 0^{\lceil n_0/2 \rceil - m - i} 1^{\lceil n_0/2 \rceil}$ для любого $i \in \mathbb{N}_0$ и любого $m \in \mathbb{N}$ – т. е. x состоит только из символов 0.
{В этом случае для доказательства невыполнения свойства (c) используется простой аргумент – аналогичный рассмотренному выше для выбора $w = 0^{n_0} 1^{n_0}$.}

(B) $y = 0^{\lceil n_0/2 \rceil - m}$, $x = 0^m 1^j$, $z = 1^{\lceil n_0/2 \rceil - j}$ для всех натуральных m и j – т. е. x включает по крайней мере один символ 0 и один символ 1.
{В этом случае свойство (c) не выполнено, т. к. $x = yxz \in L$, а $yx^2z \notin L$ – поскольку слово yx^2z нельзя записать в виде a^*b^*.}

(C) $y = 0^{\lceil n_0/2 \rceil} 1^i$, $x = 1^m$, $z = 1^{\lceil n_0/2 \rceil - i - m}$ для любых $i \in \mathbb{N}_0$ и $m \in \mathbb{N}$.
{В этом случае мы можем получить «разрастание» единиц без увеличения числа нулей – поэтому, аналогично случаю (A), свойство (c) не выполнено.}

Таким образом, использование слова $0^{\lceil n_0/2 \rceil} 1^{\lceil n_0/2 \rceil}$ для доказательства условия $L \notin \mathcal{L}(\text{КА})$ с помощью леммы о разрастании «требует больше работы», чем использование слова $0^{n_0} 1^{n_0}$.

Упражнение 3.15. Докажите с помощью леммы о разрастании, что следующие языки не являются регулярными:

- $\{ww \mid w \in \{0,1\}^*\}$;
- $\{a^n b^n c^n \mid n \in \mathbb{N}_0\}$;
- $\{w \in \{0,1\}^* \mid |w|_0 = |w|_1\}$;
- $\{a^{n^2} \mid n \in \mathbb{N}_0\}$;
- $\{a^{2^n} \mid n \in \mathbb{N}_0\}$;

[18] Для слова $0^{n_0} 1^{n_0}$ свойство (a) гарантирует, что x состоит только из символов 0. Однако при рассмотрении слова $w = 0^{\lceil n_0/2 \rceil} 1^{\lceil n_0/2 \rceil}$ мы получаем, что x может быть подсловом слова w.

- $\{w \in \{0,1\}^* \mid |w|_0 = 2|w|_1\}$;
- $\{x1y \mid x, y \in \{0,1\}^*, |x| = |y|\}$.

Упражнение 3.16. Для каждого языка над алфавитом Σ_{bool} из упражнения 3.15 найдите такое слово, что любое его представление в виде yxz удовлетворяет всем трём условиям (a), (b) и (c) леммы о разрастании.

Упражнение 3.17. Докажите следующую версию леммы о разрастании.

Для каждого регулярного языка $L \subseteq \Sigma^*$ существует некоторая константа $n_0 \in \mathbb{N}_0$, такая что любое слово $w \in \Sigma^*$ с длиной $|w| \geq n_0$ может быть представлено в виде $w = yxz$, где:

- $|xz| \leq n_0$;
- $|x| \geq 1$;
- либо $\{yx^kz \mid k \in \mathbb{N}_0\} \subseteq L$, либо $\{yx^kz \mid k \in \mathbb{N}_0\} \cap L = \emptyset$.

Упражнение 3.18.* Сформулируйте и докажите обобщённую форму леммы о разрастании, которая включала бы саму лемму 3.14 и лемму о разрастании из упражнения 3.17 как специальные случаи.

А другой метод доказательства условия $L \notin \mathcal{L}(\text{KA})$ базируется на использовании сложности по Колмогорову. Этот метод использует специальный комбинаторный параметр и с его помощью показывает, что конечные автоматы не могут считать произвольно далеко – или, что то же самое, не могут сохранять слишком много информации о читаемом слове. Метод основан на сформулированной далее теореме 3.19 – о том, что все суффиксы всех слов произвольного регулярного языка имеют небольшое значение сложности по Колмогорову.

Теорема 3.19.* *Пусть $L \subseteq (\Sigma_{\text{bool}})^*$ – некоторый регулярный язык. Для каждого $x \in \Sigma^*$ положим*

$$L_x = \{y \in \Sigma^* \mid xy \in L\}.$$

Тогда существует некоторая константа const, такая что для всех $x, y \in \Sigma^*$ выполнено неравенство

$$K(y) \leq \lceil \log_2(n+1) \rceil + const.$$

(Здесь мы считаем, что y – n-е по порядку слово языка L_x.)

Доказательство. Поскольку язык L регулярен, существует некоторый конечный автомат M, такой что $L(M) = L$. Идея нашего доказательства в чём-то аналогична доказательству теоремы 2.57 – хотя далеко не полностью его повторяет. Если в точности следовать доказательству теоремы 2.57, то мы должны генерировать все слова $z \in \Sigma^*$ в каноническом порядке и моделировать вычисления автомата M для каждого слова xz – чтобы определить, выполнено ли условие $xz \in L = L(M)$.[19] Обнаружение n-го слова xz с фиксированным префиксом x соответствует тому факту, что z – n-е слово языка L_x; таким образом, мы фактически получаем программу для генерации слова z. Некоторый недостаток этого подхода состоит в том, что такая программа, генерирующая n-е слово y языка L_x, должна иметь на входе не только n и M, но и x – однако слово x может иметь сколь угодно большое значение сложности по Колмогорову $K(x)$ по сравнению со значением $K(y)$.

[19] Иными словами, мы должны ответить на вопрос, выполняется ли включение $z \in L_x$

Очень важный момент здесь заключается в том, что программа, генерирующая слово y, не нуждается в полной информации о слове x; поясним это. Достаточно включить в программу M состояние $\hat{\delta}(q_0, x)$, а затем моделировать вычисления над генерируемым словом z, всегда начиная при этом с состояния $\hat{\delta}(q_0, x)$ – вместо моделирования работы программы M над всем словом xz (для каждого генерируемого слова z), начиная со стартового состояния q_0.

Итак, пусть y – n-е слово языка L_x для $x \in \Sigma^*$. Программа $A_{x,y}$, генерирующая y, может быть описана следующим образом:

```
A_{x,y}:  begin
            z := λ; i := 0;
            while i < n do begin
              Моделировать работу автомата M,
              начиная с состояния δ̂(q_0, x), над словом z;
              if δ̂(δ̂(q_0, x), z) ∈ F then
                begin i := i + 1;
                  y := z
                end;
                z := слово, следующее за словом z
                в каноническом порядке.
              end;
              write(y);
            end
```

Для всех $x, y \in \Sigma^*$ программа $A_{x,y}$ одна и та же – за исключением значения n и состояния $\hat{\delta}(q_0, x)$. Особая роль этого состояния может быть отмечена при описании автомата M с помощью специального указателя на состояние $\hat{\delta}(q_0, x)$. Поскольку возможны только $|Q|$ различных вариантов для этого указателя, существует константа $const_M$, являющаяся верхней границей двоичной длины представления автомата M с дополнительным указателем на специальное состояние (причём не имеет значения, на какое именно). Длина двоичного представления программы $A_{x,y}$, исключая параметры n, M и $\hat{\delta}(q_0, x)$, есть некоторая константа d – причём эта константа не зависит от x и y. Поскольку число n может быть закодировано в двоичном виде с использованием $\lceil \log_2(n+1) \rceil$ бит, мы получаем такую оценку:

$$K(y) \leq \lceil \log_2(n+1) \rceil + const_M + d.$$

Итак, теорема 3.19 доказана. □

Приведём ещё одно доказательство того, что

$$L = \{0^n 1^n \mid n \in \mathbb{N}_0\} \notin \mathcal{L}(\mathrm{KA}).$$

– и для этого применим теорему 3.19. Доказательство, как обычно, проведём от противного. Предположим, что язык L регулярен. Для каждого $m \in \mathbb{N}_0$ слово 1^m – первое слово в языке

$$L_{0^m} = \{y \mid 0^m y \in L\} = \{0^j 1^{m+j} \mid j \in \mathbb{N}_0\}.$$

Согласно теореме 3.19, существует константа c, не зависящая от m (т. е. не зависящая от слова 1^m), такая что

$$K(1^m) \leq K(1) + c.$$

Поскольку $K(1)$ – тоже константа, мы получаем неравенство

$$K(1^m) \leq d \tag{3.5}$$

– для константы $d = K(1) + c$ и всех $m \in \mathbb{N}$. Но всё это приводит к противоречию со следующими двумя фактами:

- число возможных программ, длина которых не превышает d, конечно,[20]
- множество $\{1^m \mid m \in \mathbb{N}_0\}$ бесконечно.

Для завершения доказательства мы можем воспользоваться утверждением о том, что конечное число программ не могут генерировать бесконечное число различных слов, но можем использовать и другой аргумент – применить утверждение, доказанное в упражнении 2.29. Последнее утверждение гарантирует существование бесконечного числа натуральных m, для которых выполнено неравенство

$$K(m) \geq \lceil \log_2(m+1) \rceil - 1. \tag{3.6}$$

И, поскольку существует некоторая константа $b \in \mathbb{N}_0$, такая что для всех $m \in \mathbb{N}_0$ выполнено неравенство $|K(1^m) - K(m)| \leq b$, неравенство (3.6) противоречит неравенству (3.5) для бесконечного числа значений $m \in \mathbb{N}_0$.

Упражнение 3.20. Докажите, что существует константа b, такая что для любого $m \in \mathbb{N}_0$ выполнено неравенство

$$|K(1^m) - K(m)| \leq b.$$

Упражнение 3.21. С помощью теоремы 3.19 докажите, что следующие языки не являются регулярными:

- $\{0^{n^2} \mid n \in \mathbb{N}_0\}$;
- $\{0^{2^{2^n}} \mid n \in \mathbb{N}_0\}$;
- $\{w \in \{0,1\}^* \mid |w|_0 = 2 \cdot |w|_1\}$;
- $\{w \in \{0,1\}^* \mid w = xx \text{ для некоторого } x \in \{0,1\}^*\}$.

3.5 Недетерминизм

Обычные программы – так же, как и введённые нами конечные автоматы – являются моделью детерминированных вычислений. Детерминизм при этом означает, что каждая конфигурация однозначно определяет, что именно будет происходить на следующем шаге вычисления. Поэтому (детерминированная) программа и её вход однозначно определяют её вычисление над этим входом.

В отличие от детерминизма, недетерминизм разрешает в каждой конфигурации выбор одного из нескольких (из конечного числа) возможных действий.[21] Вследствие этого недетерминированная программа может иметь несколько различных вычислений для одного и того же входа. Единственное требование здесь таково: по крайней

[20] А именно, не превосходит 2^d.

[21] Таким образом, имеется, вообще говоря, несколько возможностей для дальнейшего продолжения вычислений.

мере одна из этих возможностей должна приводить к правильному результату. Это требование может показаться неестественным – поскольку оно соответствует предположению, что недетерминированная программа всегда выбирает правильное вычисление; такой выбор из нескольких возможностей называется **недетерминированным решением**. Для проблемы принадлежности (Σ, L) это означает, что недетерминированная программа (недетерминированный конечный автомат) A принимает язык L, если для каждого $x \in L$ имеется по крайней мере одно допускающее вычисление автомата A над словом x – и наоборот, для каждого $y \in \Sigma^* - L$ ни одно из вычислений A над y не является допускающим. Хотя на первый взгляд может показаться, что недетерминированная программа для практических целей бесполезна,[22,23] исследование недетерминизма и недетерминированных вычислений внесло существенный вклад в наше понимание детерминированных вычислений, и особенно – в определение возможных пределов для алгоритмического решения проблем. Как мы увидем позже, недетерминизм стал очень мощным инструментом для исследования количественных законов вычислений – и это является одной из главных причин для рассмотрения его в нашей книге и изучения методов работы с ним.

Итак, цель данного раздела состоит в том, чтобы ввести и изучить недетерминизм для нашего случая – модели конечных автоматов. При этом мы хотим ответить на следующие вопросы. Можно ли моделировать недетерминированные вычисления, используя детерминированные? Если да – то «какую цену[24] надо за это заплатить»? Для программ можно ввести недетерминизм, просто разрешив дополнительную возможность – команды типа

«`choose goto` i `or goto` j».

В случае конечных автоматов эта возможность соответствует тому, что мы разрешаем несколько возможных переходов из некоторого состояния при чтении одного и того же самого входного символа – то есть разрешаем несколько возможных дуг, выходящих из одного и того же состояния и помеченных одним и тем же символом входного алфавита (рис. 3.9).

Определение 3.22. Недетерминированный конечный автомат *(***НКА***)* *– это пятёрка* $M = (Q, \Sigma, \delta, q_0, F)$*, где:*

- Q *– некоторое конечное множество, называемое* **множеством состояний**,
- Σ *– алфавит, называемый* **входным алфавитом**,
- $q_0 \in Q$ – **инициальное (стартовое) состояние**,
- $F \subseteq Q$ *– множество* **допускающих состояний**,

[22] Потому что у нас нет т.н. «оракула» который помогал бы нам принимать правильные решения.

[23] Впрочем, в книге *[Ф.Новиков. «Дискретная математика для программистов», СПб, Питер, 2002]*, ставшей в последние годы в российских ВУЗах одним из основных учебников, часто приводится запись именно недетерминированных программ.

Стоит ещё отметить, что термин «вычисления с оракулом» и соответствующая методика построения недетерминированных программ часто применяются в *теории* недетерминированных вычислений. Подобная методика будет использована и в последующих главах – причём обычно без явного указания на этот термин. Практически в таком же смысле (как «оракул») употребляется и термин «навигатор». *(Прим. перев.)*

[24] Объём дополнительных вычислительных средств и т.п.

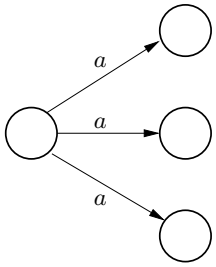


Рис. 3.9.

- *δ – функция, действующая из $Q \times \Sigma$ в $\mathcal{P}(Q)$, называемая* **функцией переходов**.[25,26]

{Заметим, что Q, Σ, q_0 и F имеют тот же самый смысл, что и для (детерминированного) конечного автомата. Но, применяя функцию δ для текущих состояния q и читаемого символа b, некоторый НКА *может иметь несколько (в том числе 0) последующих состояний, т. е. $\delta(q, b)$ является множеством состояний (возможных действий) – в отличие от необходимости иметь ровно одно состояние (действие) в случае детерминированных автоматов.}*

Конфигурация *недетерминированного конечного автомата M – элемент множества $Q \times \Sigma^*$. Конфигурация (q_0, x) является* **инициальной конфигурацией для входного слова x**.

Шаг автомата M *– это отношение*

$$|\overline{}_M \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*),$$

определённое следующим образом:

$$(q, w) \,|\overline{}_M\, (p, x) \Leftrightarrow w = ax \text{ для некоторых } a \in \Sigma \text{ и } p \in \delta(q, a).$$

Вычисление M *– это конечная последовательность конфигураций $D_1, D_2, \ldots, D_k$, такая что для $i = 1, \ldots, k-1$ выполнено отношение $D_i \,|\overline{}_M\, D_{i+1}$.*

Вычисление над словом x *– это вычисление $C_0, C_1, \ldots, C_m$ автомата M, в котором:*

- *$C_0 = (q_0, x)$, т. е. C_0 – инициальная конфигурация M для x,*
- *либо $C_m \in Q \times \{\lambda\}$;*
  либо $C_m = (q, ax)$ для некоторых $a \in \Sigma$ и $q \in Q$, таких что $\delta(q, a) = \emptyset$.

{Вычисление автомата M над x останавливается либо когда прочтено всё входное слово, либо – в отличие от (детерминированных) конечных автоматов – если не

[25] Мы можем определить δ альтернативным образом – как некоторое отношение на $(Q \times \Sigma) \times Q$.

[26] Имеются и другие подходы к определению детерминированных и недетерминированных конечных автоматов. По-видимому, наиболее важный «критерий», разделяющий эти подходы на группы, таков: являются ли детерминированные конечные автоматы (собственным) подмножеством недетерминированных – или представляют собой иной формализм? *(Прим. перев.)*

существует возможности продолжить вычисления для некоторой текущей пары аргументов (q, a).*}*
$C_0, C_1, \ldots, C_m$ – *некоторое* **допускающее вычисление M над x**, *если* $C_m = (p, \lambda)$ *для некоторого состояния* $p \in F$.
{M *допускает* x *в вычислении* C *только в том случае, когда, во-первых, слово* x *прочитано полностью, и, во-вторых, вычисление заканчивается в некотором допускающем состоянии.}*
Если существует допускающее вычисление M *над* x, *то мы говорим, что* **M допускает слово x**.
Отношение $|\frac{*}{M}$ *обозначает рефлексивно-транзитивное замыкание*[27] *отношения* $|\frac{}{M}$.
Язык, допускаемый M *есть*

$$\boldsymbol{L(M)} = \{w \in \Sigma^* \mid (q_0, w) \,|\frac{*}{M}\, (p, \lambda) \text{ для некоторого } p \in F\}.$$

Для функции переходов δ *функцию* $\hat{\boldsymbol{\delta}}$, *действующую из* $Q \times \Sigma^*$ *в* $\mathcal{P}(Q)$, *определим для всех* $q \in Q, a \in \Sigma$ *и* $w \in \Sigma^*$ *следующим образом:*

- $\hat{\delta}(q, \lambda) = \{q\}$,
- $\hat{\delta}(q, wa) = \{p \mid$ *существует* $r \in \hat{\delta}(q, w)$, *такое что* $p \in \delta(r, a)\}$
  $= \bigcup_{r \in \hat{\delta}(q,w)} \delta(r, a)$.

Мы видим, что некоторое слово x принадлежит языку $L(M)$ тогда и только тогда, когда у автомата M имеется хотя бы одно допускающее вычисление над входом x. Для любого допускающего вычисления требуется, во-первых, чтобы были прочитаны все символы входного слова, и, во-вторых, чтобы M непосредственно после прочтения последнего символа этого слова попадал в некоторое допускающее состояние.[28] Но, в отличие от (детерминированных) конечных автоматов, отклоняющее вычисление может заканчиваться и без прочтения всего входного слова. Это происходит, если для некоторой текущей пары аргументов не существует ни одного перехода – т. е. если $\delta(q, a) = \emptyset$ для текущего состояния q и читаемого символа a.

Согласно определению $\hat{\delta}$, множество $\hat{\delta}(q_0, w)$ – это подмножество таких состояний множества Q, которые достигаются после прочтения всего входного слова w из состояния q_0, т. е.

$$\hat{\delta}(q_0, w) = \{p \in Q \mid (q_0, w) \,|\frac{*}{M}\, (p, \lambda)\}.$$

Следовательно,

$$L(M) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

– это эквивалентное определение языка, допускаемого недетерминированным конечным автоматом M.

Рассмотрим следующий пример недетерминированного автомата. Пусть $M = (Q, \Sigma, \delta, q_0, F)$, где:

$Q = \{q_0, q_1, q_2\}$,
$\Sigma = \{0, 1\}$,
$F = \{q_2\}$,

[27] В точности как и в определении (детерминированных) конечных автоматов.
[28] Второе требование совпадает с аналогичным в случае (детерминированных) конечных автоматов.

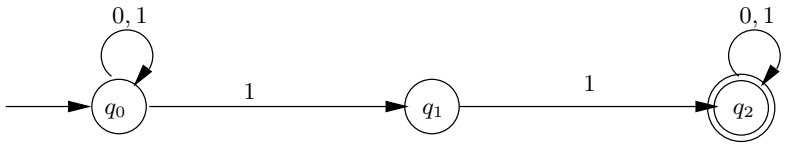


Рис. 3.10.

$\delta(q_0, 0) = \{q_0\}$, $\delta(q_0, 1) = \{q_0, q_1\}$,
$\delta(q_1, 0) = \emptyset$, $\delta(q_1, 1) = \{q_2\}$,
$\delta(q_2, 0) = \{q_2\}$, $\delta(q_2, 1) = \{q_2\}$.

Используя ту же самую процедуру, что и для (детерминированных) конечных автоматов, мы можем получить графическое представление недетерминированных. Рассмотренный выше НКА M изображён на рис. 3.10.

Языку $L(M)$ принадлежит, например, слово 10110 – поскольку

$$(q_0, 10110) \,|\overline{}_M\, (q_0, 0110) \,|\overline{}_M\, (q_0, 110) \,|\overline{}_M\, (q_1, 10) \,|\overline{}_M\, (q_2, 0) \,|\overline{}_M\, (q_2, \lambda)$$

является допускающим вычислением M над словом 10110.

Чтобы ответить на вопрос, принимает ли НКА M слово x, мы должны следовать всем возможным вычислениям M над x. Описание всех этих вычислений может быть представлено с помощью т. н. **дерева вычислений $\mathcal{B}_M(x)$ автомата M над x**. Вершины этого дерева суть конфигурации автомата M. Корнем $\mathcal{B}_M(x)$ является инициальная конфигурация M над x. Дочерние вершины для (q, α) описывают конфигурации, которые могут быть достигнуты за *один* шаг вычислений из рассматриваемой конфигурации (q, α) – т. е. все конфигурации (p, β), такие что $(q, \alpha) \,|\overline{}_M\, (p, \beta)$. Любой лист дерева $\mathcal{B}_M(x)$ – это либо финальная конфигурация (r, λ), либо конфигурация $(s, a\beta)$ с некоторым $a \in \Sigma$, такая что $\delta(s, a) = \emptyset$. Следовательно, листья – это такие конфигурации, в которых дальнейшие шаги вычислений невозможны. В этом представлении любой путь дерева $\mathcal{B}_M(x)$ от корня до листа соответствует вычислению автомата M над x, и наоборот. Поэтому число листьев дерева $\mathcal{B}_M(x)$ в точности равно числу различных вычислений M над x.

Упражнение 3.23. Приведите пример НКА, который имеет в точности $2^{|x|}$ различных вычислений на каждом входе $x \in (\Sigma_{\text{bool}})^*$.

Пусть M – автомат, изображённый на рис. 3.10. Дерево вычислений $\mathcal{B}_M(x)$ автомата M над входом $x = 10110$ приведено на рис. 3.11. У дерева $B_M(10110)$ имеется четыре листа. Два из них – т. е. $(q_1, 0110)$ и $(q_1, 0)$ – соответствуют вычислениям, для которых M не дочитывает входное слово до конца (поскольку $\delta(q_1, 0) = \emptyset$). Следовательно, эти два вычисления не являются допускающими. А листья (q_0, λ) и (q_2, λ) соответствуют двум вычислениям, в которых слово 10110 прочитывается полностью. Поскольку $q_2 \in F$, соответствующее вычисление[29] является допускающим – и, следовательно, $10110 \in L(M)$.

Поскольку q_2 – единственное допускающее состояние автомата M, а единственная возможность достигнуть q_2 из q_0 – это прочитать подслово 11, то можно предположить,

[29] Вычисление, которое кончается в конфигурации (q_2, λ).

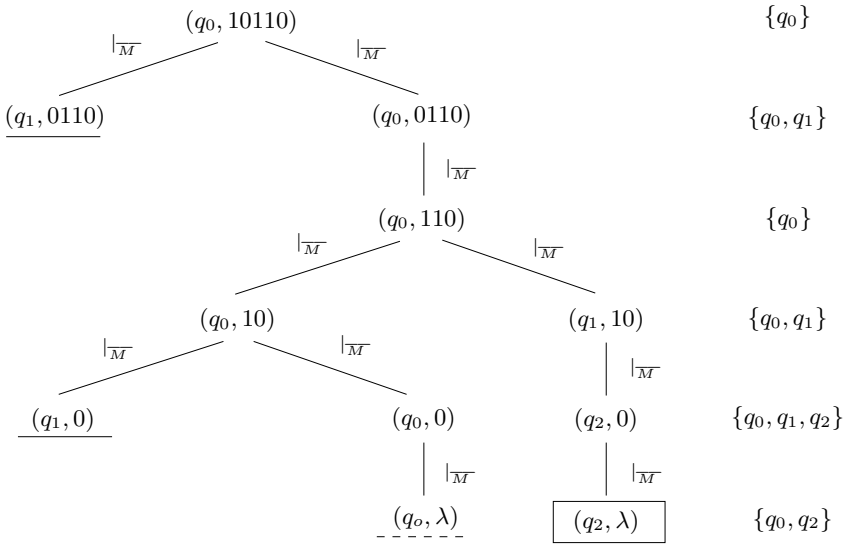


Рис. 3.11.

что язык $L(M)$ – множество всех слов вида $x11y$, где x и y – произвольные слова из $(\Sigma_{\text{bool}})^*$. Следующая лемма подтверждает эту гипотезу.

Лемма 3.24. *Пусть M – автомат, изображённый на рис. 3.10. Тогда*

$$L(M) = \{x11y \mid x, y \in (\Sigma_{\text{bool}})^*\}.$$

Доказательство. Докажем равенство этих множеств путём доказательства двух соответствующих включений.

$\supseteq$ Во-первых, покажем, что $\{x11y \mid x, y \in (\Sigma_{\text{bool}})^*\} \subseteq L(M)$.
Пусть $w \in \{x11y \mid x, y \in (\Sigma_{\text{bool}})^*\}$, т. е. $w = x11y$ для некоторых $x, y \in (\Sigma_{\text{bool}})^*$. Нам достаточно доказать существование допускающего вычисления автомата M над словом w.
Вследствие $q_0 \in \delta(q_0, 0) \cap \delta(q_0, 1)$ имеется следующее вычисление автомата M над любым словом $x \in (\Sigma_{\text{bool}})^*$:

$$(q_0, x) \vdash_M^* (q_0, \lambda). \tag{3.7}$$

А поскольку $q_2 \in \delta(q_2, 0) \cap \delta(q_2, 1)$, существует следующее вычисление автомата M над любым $y \in (\Sigma_{\text{bool}})^*$:

$$(q_2, y) \vdash_M^* (q_2, \lambda). \tag{3.8}$$

Следовательно, вычисление

$$(q_0, x11y) \vdash_M^* (q_0, 11y) \vdash_M (q_1, 1y) \vdash_M (q_2, y) \vdash_M^* (q_2, \lambda)$$

автомата M над словом $x11y$ является допускающим.

$\subseteq$ Теперь докажем, что $L(M) \subseteq \{x11y \mid x, y \in (\Sigma_{\text{bool}})^*\}$.
Пусть $w \in L(M)$. Тогда существует допускающее вычисление C автомата M над словом w. Поскольку это допускающее вычисление C над w должно начинаться

в инициальном состоянии q_0 и заканчиваться в единственном допускающем состоянии q_2, а единственный путь из q_0 в q_2 проходит через q_1, вычисление C имеет следующий вид:

$$(q_0, w) \mid\!\!\frac{*}{M} (q_1, z) \mid\!\!\frac{*}{M} (q_2, \lambda). \tag{3.9}$$

Каждое вычисление автомата M может содержать не более одной конфигурации с состоянием q_1, поскольку:

- $q_1 \notin \delta(q_1, a)$ для любого $a \in \Sigma_{\text{bool}}$;
- если автомат M проходит состояние q_1, то он не может больше попасть в это же состояние q_1.

Поэтому можно представить допускающее вычисление C следующим образом:

$$(q_0, w) \mid\!\!\frac{*}{M} (q_0, abu) \mid\!\!\frac{}{M} (q_1, bu) \mid\!\!\frac{}{M} (q_2, u) \mid\!\!\frac{*}{M} (q_2, \lambda), \tag{3.10}$$

где $a, b \in \Sigma_{\text{bool}}$, а $u \in \Sigma^*_{\text{bool}}$. Единственная возможность достигнуть состояния q_1 – это применить переход $q_1 \in \delta(q_0, 1)$, т. е. в состоянии q_0 нужно прочитать символ 1. Это означает, что в вычислении (3.10) $a = 1$. Поскольку $(q_1, 0) = \emptyset$ и $\delta(q_1, 1) = \{q_2\}$, единственная возможность выполнить шаг вычислений из состояния q_1 – это прочитать символ 1, и, следовательно, $b = 1$. Переписывая вычисление (3.10),[30] мы получаем, что C должно иметь следующий вид:

$$(q_0, w) \mid\!\!\frac{*}{M} (q_0, 11u) \mid\!\!\frac{}{M} (q_1, 1u) \mid\!\!\frac{}{M} (q_2, u) \mid\!\!\frac{*}{M} (q_2, \lambda).$$

Вследствие этого слово w должно включать подслово 11, и поэтому

$$w \in \{x11y \mid x, y \in (\Sigma_{\text{bool}})^*\}.$$

□

Упражнение 3.25. Приведите описание недетерминированных конечных автоматов для следующих языков:

(a) $\{1011x00y \mid x, y \in (\Sigma_{\text{bool}})^*\}$;
(b) $\{01, 101\}^*$;
(c) $\{x \in (\Sigma_{\text{bool}})^* \mid x$ содержит подслова 01011 и 01100$\}$;
(d) $\{x \in (\Sigma_{10})^* \mid Number_{10}(x)$ делится на 3$\}$.

При этом постарайтесь сконструировать НКА как можно более простым – т. е. минимизировать как число состояний (вершин), так и число переходов (дуг).

Пусть

$$\mathcal{L}(\text{НКА}) = \{L(M) \mid M - \text{некоторый НКА}\}.$$

Главный вопрос этого раздела – верно ли равенство $\mathcal{L}(\text{НКА}) = \mathcal{L}(\text{КА})$; этот вопрос можно сформулировать следующим образом: верно ли, что (детерминированные) конечные автоматы могут моделировать работу недетерминированных. Подобный вопрос является важнейшим и для более общих вычислительных моделей. На основе большого научного опыта мы знаем, что моделирование недетерминизма детерминизмом

[30] Т. е. заменяя в вычислении (3.10) a и b на 1.

выполнимо только в том случае, когда есть возможность имитации *одним* детерминированным вычислением *всех* вычислений данной недетерминированной модели на любом заданном входе. Для случая конечных автоматов это возможно.

Идея моделирования некоторого недетерминированного конечного автомата M с помощью (детерминированного) автомата A базируется на поиске в ширину в деревьях вычислений M. Важное предположение для этого моделирования таково: у всех конфигураций дерева вычислений, находящихся на одном и том же расстоянии i от корня, совпадают вторые элементы – потому что они достигаются после чтения первых i символов входного слова. Итак, эти конфигурации на одном и том же расстоянии от корня могут отличаться только своими состояниями. И, хотя число конфигураций на расстоянии i от корня может экспоненциально зависеть от i – этот факт не означает, что мы должны моделировать экспоненциально много различных вычислений. У автомата M имеется конечное число состояний – и, следовательно, для любого возможного $i \in \mathbb{N}_0$ на расстоянии i от корня имеется конечное число различных возможных конфигураций. Если две разные вершины u и v дерева вычислений помечены одной и той же конфигурацией, то поддеревья, соответствующие u и v, идентичны – и, следовательно, нам достаточно найти допускающую конфигурацию только в одном из этих двух поддеревьев (рис. 3.12). Это означает, что для моделирования работы НКА M над входным словом x достаточно для каждого $i \in \{0, 1, \ldots, |x|\}$ определить множество всех состояний, достижимых после i шагов вычислений M над x. Для любого такого i это множество в точности совпадает с $\hat{\delta}(q_0, z)$, где z – префикс слова x длины i. На рис. 3.11 справа отмечены множества $\hat{\delta}(q_0, 10) = \{q_0\}$, $\hat{\delta}(q_0, 101) = \{q_0, q_1\}$, $\hat{\delta}(q_0, 1011) = \{q_0, q_1, q_2\}$ и $\hat{\delta}(q_0, 10110) = \{q_0, q_2\}$ – которые соответствуют множествам достижимых состояний после $i = 0, 1, \ldots, 5$ шагов (т. е. после прочтения префикса длины i).

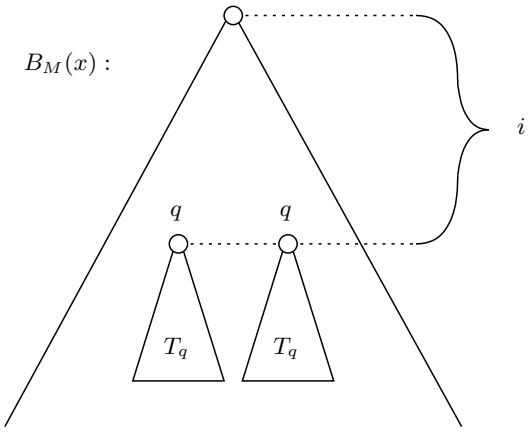


Рис. 3.12.

Это наблюдение даёт такую идею: взять *подмножества* множества состояний Q заданного НКА $M = (Q, \Sigma, \delta, q_0, F)$ в качестве состояний моделирующего (детерминированного) конечного автомата A. Поэтому следующая конструкция автомата A в теории автоматов называется супермножественной конструкцией (powerset construction).

Состояние $\langle P\rangle$ автомата A для множества $P \subseteq Q$ имеет следующую интерпретацию: это – в точности те состояния множества P, которые достижимы[31] в вычислении автомата M на входе z, т. е. $P = \hat{\delta}(q_0, z)$. Шаг вычисления автомата A из состояния $\langle P\rangle$ при чтении символа a определяется как

$$\bigcup_{p\in P} \delta(p, a)$$

– т. е. как множество всех состояний, которые могут быть достигнуты при чтении символа a автоматом M, находящемся в некотором состоянии множества P. Формализация этой идеи приводится в следующей теореме.

Теорема 3.26. *Для каждого* НКА M *существует (детерминированный) конечный автомат* A*, такой что*

$$L(M) = L(A).$$

Доказательство. Пусть $M = (Q, \Sigma, \delta_M, q_0, F)$ – некоторый НКА. Мы строим эквивалентный[32] автомат $A = (Q_A, \Sigma_A, \delta_A, q_{0A}, F_A)$ следующим образом:

- $Q_A = \{\langle P\rangle \mid P \subseteq Q\}$,
- $\Sigma_A = \Sigma$,
- $q_{0A} = \langle\{q_0\}\rangle$,
- $F_A = \{\langle P\rangle \mid P \subseteq Q \text{ и } P \cap F \neq \emptyset\}$,
- δ_A – функция, действующая из $Q_A \times \Sigma_A$ в Q_A, такая что для любых $\langle P\rangle \in Q_A$ и $a \in \Sigma_A$ выполнено следующее:

$$\begin{aligned}\delta_A(\langle P\rangle, a) &= \left\langle \bigcup_{p\in P} \delta_M(p, a) \right\rangle \\ &= \langle\{q \in Q \mid \exists p \in P, \text{ такое что } q \in \delta_M(p, a)\}\rangle .\end{aligned}$$

Очевидно, что автомат A – детерминированный. Например, на рис. 3.13 приведён автомат A, созданный с помощью супермножественной конструкции[33] на основе НКА M, приведённого на рис. 3.10. Для доказательства того, что автомат A действительно эквивалентен НКА M, достаточно показать, что для всех $x \in \Sigma^*$ выполнено следующее:

$$\hat{\delta}_M(q_0, x) = P \iff \hat{\delta}_A(q_{0A}, x) = \langle P\rangle . \tag{3.11}$$

Докажем эквивалентность (3.11) индукцией по $|x|$.

1. *Базис индукции.*
   Пусть $|x| = 0$, т. е. $x = \lambda$. Поскольку $\hat{\delta}_M(q_0, \lambda) = \{q_0\}$ и $q_{0A} = \langle\{q_0\}\rangle$, то утверждение (3.11) для $x = \lambda$ выполнено.

[31] Мы используем обозначение $\langle P\rangle$ вместо P для пояснения разницы между *одним* состоянием $\langle P\rangle$ автомата A, соответствующим *множеству* P достижимых состояний автомата M – и этим множеством состояний P.

[32] Т. е. определяющий тот же самый регулярный язык.

[33] Заметим, что состояния $\langle\emptyset\rangle$, $\langle\{q_1\}\rangle$, $\langle\{q_2\}\rangle$ и $\langle\{q_1, q_2\}\rangle$ автомата A, приведённого на рис. 3.10, не являются достижимыми из инициального состояния $\langle\{q_0\}\rangle$ – т. е. не существует ни одного слова, после чтения которого автомат A финишировал бы в каком-либо из этих состояний. Следовательно, удаление этих состояний (называющихся недостижимыми) не изменяет язык, допускаемый автоматом A.

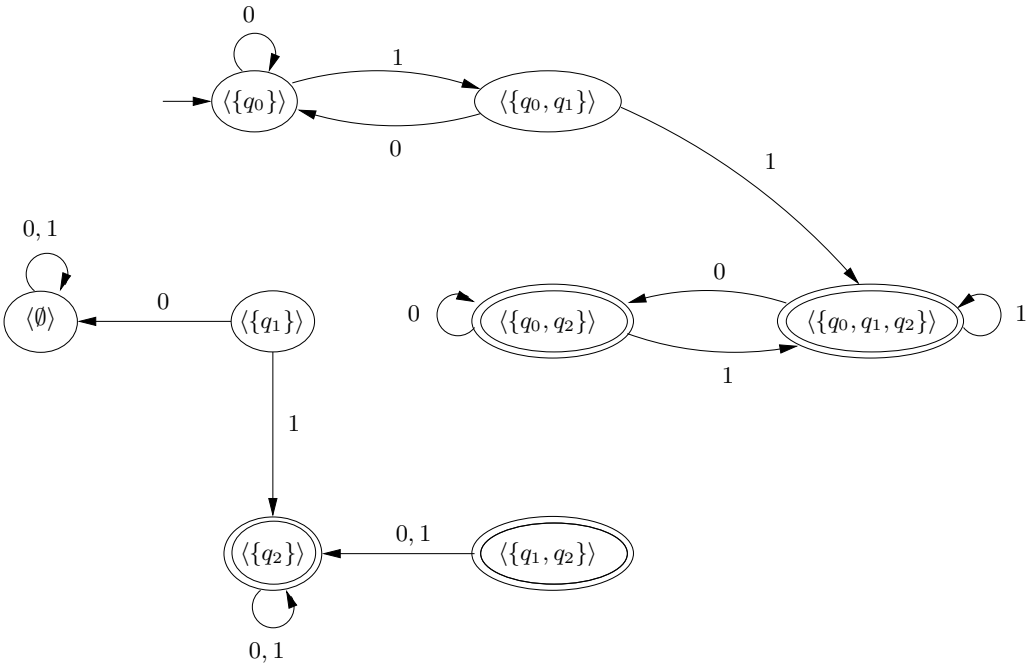


Рис. 3.13.

2. *Шаг индукции.*
Пусть эквивалентность (3.11) выполнена для некоторого $m \in \mathbb{N}_0$ – т. е. для всех слов $z \in \Sigma^*$, таких что $|z| \leq m$. Докажем, что такое же условие выполнено и для всех слов множества Σ^{m+1}.
Пусть y – произвольное слово множества Σ^{m+1}. Тогда $y = xa$ для некоторых $x \in \Sigma^m$ и $a \in \Sigma$. Следуя определению функции $\hat{\delta}_A$, мы получаем равенство

$$\hat{\delta}_A(q_{0A}, xa) = \delta_A(\hat{\delta}_A(q_{0A}, x), a). \tag{3.12}$$

Применяя предположение индукции (3.11) для слова x, мы имеем

$$\hat{\delta}_A(q_{0A}, x) = \langle R \rangle \Leftrightarrow \hat{\delta}_M(q_0, x) = R$$

– и, следовательно,

$$\hat{\delta}_A(q_{0A}, x) = \left\langle \hat{\delta}_M(q_0, x) \right\rangle. \tag{3.13}$$

Согласно способу построения автомата A (его функции переходов δ_A) мы получаем, что для всех $R \subseteq Q$ и $a \in \Sigma$ выполнено следующее:

$$\delta_A(\langle R \rangle, a) = \left\langle \bigcup_{p \in R} \delta_M(p, a) \right\rangle. \tag{3.14}$$

Итак,

$$\begin{aligned}\hat{\delta}_A(q_{0A}, xa) &\underset{(3.12)}{=} \delta_A(\hat{\delta}_A(q_{0A}, x), a)\\ &\underset{(3.13)}{=} \delta_A(\left\langle \hat{\delta}_M(q_0, x)\right\rangle, a)\\ &\underset{(3.14)}{=} \left\langle \bigcup_{p\in\hat{\delta}_M(q_0,x)} \delta_M(p, a)\right\rangle\\ &= \left\langle \hat{\delta}_M(q_0, xa)\right\rangle.\end{aligned}$$

Этим заканчивается доказательство условия (3.11) – и всей теоремы. □

В дальнейшем мы будем говорить, что два автомата A и B **эквивалентны**, если $L(A) = L(B)$.

Упражнение 3.27. Примените супермножественную конструкцию теоремы 3.26 для описания детерминированного автомата, эквивалентного НКА, приведённому на рис. 3.14.

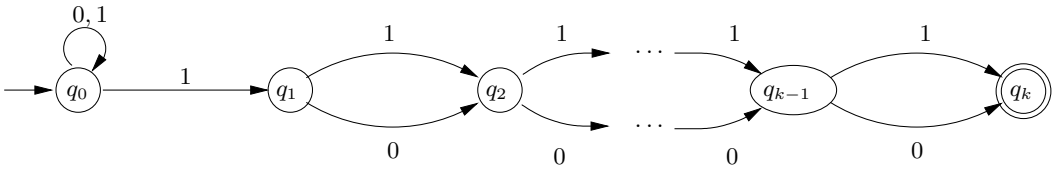


Рис. 3.14.

Упражнение 3.28. Примените супермножественную конструкцию для конструирования конечных автоматов, которые эквивалентны недетерминированным конечным автоматам, созданным ранее в пунктах (b) и (d) упражнения 3.25.

Следствием теоремы 3.26 является равенство

$$\mathcal{L}(\text{КА}) = \mathcal{L}(\text{НКА})$$

– т. е. с точки зрения множества допускаемых языков (детерминированные) конечные автоматы столь же мощны, как и недетерминированные. Однако стоит отметить, что конечные автоматы, построенные с помощью супермножественной конструкции, имеют экспоненциально много состояний по сравнению с соответствующими недетерминированными.

Далее рассмотрим такой вопрос. Существует ли альтернативный вариант моделирования НКА с помощью (детерминированных) автоматов, гарантирующий существование такого автомата, число состояний которого не больше, чем число состояний исходного? Или, другими словами, существуют ли регулярные языки, для которых моделирование недетерминизма детерминизмом неизбежно приводит к показательному росту числа состояний? Мы ответим на этот вопрос, показав, что супермножественную конструкцию нельзя улучшить.

Для этого при любом натуральном k рассмотрим регулярный язык

$$L_k = \{x1y \mid x \in (\Sigma_{\text{bool}})^*, y \in (\Sigma_{\text{bool}})^{k-1}\}.$$

Недетерминированный конечный автомат A_k, приведённый на рис. 3.14, принимает язык L_k – отвечая в состоянии q_0 для каждого символа 1 входного слова на вопрос, является ли этот символ k-м символом с конца. При этом A_k определяет – причём детерминированным образом – было ли это предположение истинным.

Упражнение 3.29. Дайте формальное описание автомата A_k, приведённого на рис. 3.14, и докажите, что A_k допускает язык L_k. Для каждого натурального k приведите описание автомата B_k, такого что $L_k = L(B_k)$.

Число состояний автомата A_k равно $k+1$. Мы покажем, что (детерминированный) конечный автомат, допускающий язык L_k, имеет число состояний, экспоненциально зависящее от размера автомата A_k (числа его состояний).

Лемма 3.30. *Для любого $k \in \mathbb{N}$ любой (детерминированный) конечный автомат, допускающий язык L_k, имеет не менее 2^k состояний.*

Доказательство. Пусть $B_k = (Q_k, \Sigma_{\text{bool}}, \delta_k, q_{0k}, F_k)$ – некоторый конечный автомат, допускающий язык $L(B_k) = L_k$. Для доказательства того, что B_k имеет по крайней мере 2^k состояний, применим такую же технику доказательства, которая в разделе 3.4 была использована для доказательства несуществования конечных автоматов, допускающих некоторые конкретные регулярные языки.[34]

Итак, если

$$\hat{\delta}_k(q_{0k}, x) = \hat{\delta}_k(q_{0k}, y) \tag{3.15}$$

для некоторых слов x и y над Σ_{bool}, то для всех слов $z \in (\Sigma_{\text{bool}})^*$ выполнено условие

$$xz \in L(B_k) \Longleftrightarrow yz \in L(B_k). \tag{3.16}$$

Идея доказательства состоит в том, чтобы указать достаточно большое множество слов S_k, обладающее следующим свойством. Для любых двух различных слов $x, y \in S_k$ не выполнено равенство (3.15) – поскольку существует некоторое слово z, такое что

$$xz \in L(B_k) \quad \text{и} \quad yz \notin L(B_k).$$

Выполнение этого свойства влечёт существование у автомата B_k по крайней мере $|S_k|$ различных состояний.[35]

Выберем

$$S_k = (\Sigma_{\text{bool}})^k$$

и докажем методом от противного, что для всех $u \in S_k$ состояния $\hat{\delta}_k(q_{0k}, u)$ должны попарно различаться. Пусть

[34] Это не должно являться неожиданностью – поскольку мы и в данном случае фактически доказываем несуществование. Точнее – мы доказываем, что не существует конечного автомата, который допускал бы язык L_k и имел бы менее 2^k состояний.

[35] Отметим ещё один возможный приём работы с конечными автоматами. Если бы значение $|S_k|$ было бы равно бесконечности, то отсюда следовало бы несуществование конечных автоматов, допускающих рассматриваемые языки.

$$x = x_1 x_2 \dots x_k \quad \text{и} \quad y = y_1 y_2 \dots y_k,$$

– два различных слова множества $S_k = (\Sigma_{\text{bool}})^k$, где x_i, $y_i \in \Sigma_{\text{bool}}$ для $i = 1, \dots, k$. Предположим, что

$$\hat{\delta}_k(q_{0k}, x) = \hat{\delta}_k(q_{0k}, y).$$

Поскольку $x \neq y$, существует некоторое $j \in \{1, \dots, k\}$, такое что $x_j \neq y_j$. Без ограничения общности мы можем предположить, что

$$x_j = 1 \quad \text{и} \quad y_j = 0.$$

Теперь рассмотрим слово $z = 0^{j-1}$. Для него

$$xz = x_1 \dots x_{j-1} 1 x_{j+1} \dots x_k 0^{j-1}, \quad yz = y_1 \dots y_{j-1} 0 y_{j+1} \dots y_k 0^{j-1}$$

– и поэтому

$$xz \in L_k, \ \text{но} \ yz \notin L_k.$$

Это приводит к противоречию с условием (3.16) – т. е. состояния $\hat{\delta}_k(q_{0k}, x)$ и $\hat{\delta}_k(q_{0k}, y)$ должны различаться. Следовательно, B_k имеет по крайней мере $|S_k| = 2^k$ состояний. □

Лемма 3.30 определяет простую технику получения нижней границы размера конечного автомата, допускающего конкретный регулярный язык. Чтобы показать, как на практике использовать эту технику, мы получим нижнюю границу числа состояний автомата, определяющего регулярный язык

$$L = \{x11y \mid x, y \in \{0,1\}^*\},$$

использовавшийся ранее при рассмотрении супермножественной конструкции (рис. 3.10).

Для доказательства того, что каждый конечный автомат, допускающий L, имеет по крайней мере 3 состояния, рассмотрим 3 следующих слова:

$$\lambda, \ 1, \ 11.$$

Для каждой из трёх пар (x, y) различных слов языка $S = \{\lambda, 1, 11\}$ мы должны показать существование некоторого слова z – такого что ровно одно из слов xz и yz входит в язык L (т. е. что x и y не удовлетворяют условию (3.16)).

Для $x = \lambda$ и $y = 1$ выбираем $z = 1$. Тогда

$$xz = 1 \notin L, \ \text{но} \ yz = 11 \in L$$

– и поэтому $\hat{\delta}(q_0, x) \neq \hat{\delta}(q_0, y)$ для любого конечного автомата, допускающего L.

Для $x = \lambda$ и $y = 11$ выбираем $z = 0$. Тогда

$$xz = 0 \notin L, \ \text{но} \ yz = 110 \in L.$$

А для $x = 1$ и $y = 11$ выбираем $z = \lambda$. Тогда

$$xz = 1 \notin L, \ \text{но} \ yz = 11 \in L.$$

Итак, состояния

$$\hat{\delta}(q_0, \lambda), \quad \hat{\delta}(q_0, 1) \quad \text{и} \quad \hat{\delta}(q_0, 11)$$

для любого автомата $A = (Q, \Sigma_{\text{bool}}, \delta, q_0, F)$, допускающего язык L, должны попарно различаться.

Упражнение 3.31. Рассмотрите языки $L = \{x11y \mid x, y \in \{0,1\}^*\}$ и $S = \{\lambda, 1, 11\}$. Для любой пары x, y различных слов языка S постройте множество $Z(x,y) \subseteq (\Sigma_{\text{bool}})^*$ – такое что для любого слова $z \in Z(x,y)$ выполнено следующее:

$$(xz \notin L \text{ и } yz \in L) \text{ или } (xz \in L \text{ и } yz \notin L).$$

Упражнение 3.32. Докажите, что любой конечный автомат, допускающий язык $L = \{x11y \mid x, y \in \{0,1\}^*\}$, имеет по крайней мере 3 состояния – при этом выбрав 3 слова, отличные от λ, 1, и 11.

Упражнение 3.33. Пусть $L = \{x011y \mid x, y \in \{0,1\}^*\}$.

- Опишите недетерминированный автомат M, допускающий L и имеющий 4 состояния, и докажите, что $L = L(M)$.
- Докажите, что не существует конечного автомата M с 3 состояниями, такого что $L(M) = L$.
- Примените супермножественную конструкцию для описания конечного автомата, допускающего L.

Упражнение 3.34.* Некоторый конечный автомат A называется **минимальным** для заданного регулярного языка $L(A)$, если не существует меньшего (с точки зрения числа состояний) конечного автомата, также допускающего $L(A)$. Опишите минимальные конечные автоматы для языков из упр. 3.25 – и докажите их минимальность.

Упражнение 3.35. Опишите НКА, имеющий не более 6 состояний, такой что $L(M) = \{0x \mid x \in \{0,1\}^*$, причём x содержит по крайней мере одно из слов 11 и 100 как подслово$\}$.

3.6 Заключение

Итак, в этой главе мы ввели конечные автоматы как модель простых вычислений – модель, которая не использует никаких вспомогательных переменных, и, следовательно, не имеет памяти. Основной целью рассмотрения конечных автоматов было не их подробное исследование, не изучение соответствующих основных результатов – а простое и ясное, но в то же время строгое, введение в моделирование процесса вычислений.

Стандартное определение вычислительной модели начинается с описания компонентов этой модели и её возможных команд (элементарных действий). При этом определяется понятие «конфигурация», являющееся обобщением понятия «состояние» – т. е. формируется связь состояния с временны́ми единицами. Выполнение одного шага (одного этапа процесса вычисления) соответствует выполнению рассматриваемой команды (элементарной операции модели) на текущей конфигурации. Следовательно, шаг – это переход от одной конфигурации C к другой конфигурации D, и различие между C и D достигается именно в результате выполнения элементарной операции. Вычисление над входом x начинается в начальной конфигурации (включающей это слово x как ввод); заключительная конфигурация определяет выход (вывод, результат).

Конечные автоматы соответствуют подклассу простых алгоритмов для решения проблемы распознавания языков. Конечный автомат не использует никаких переменных – т. е. никакой памяти. Работу конечного автомата можно рассматривать как движение среди конечного числа состояний (строк программы) – и принятие входного

слова происходит в том случае, когда после прочтения всего входного слова эта работа завершается в одном из допускающих состояний.

То, что некоторая проблема не решается ни одним алгоритмом, принадлежащим специальному подклассу, является т. н. проблемой неразрешимости (несуществования). Подобные доказательства обычно требуют глубокого понимания природы соответствующего подкласса алгоритмов. В случае конечных автоматов доказательство того, что $L \notin \mathcal{L}(\mathrm{KA})$, основано на следующей идее: структура языка L слишком сложна для того, чтобы быть описанной конечным отношением эквивалентности на множестве Σ^*. Иными словами, для хранения всех важных характеристик префиксов входных слов недостаточно конечного числа классов этого отношения (т. е. конечного числа состояний автомата). Та же самая методика может быть использована и для получения нижней границы числа состояний любого конечного автомата, допускающего некоторый заданный регулярный язык L.

В отличие от своих детерминированных аналогов, недетерминированные вычислительные модели (алгоритмы)[36] позволяют на любом шаге вычислений выбирать одно из нескольких возможных действий.[37] Таким образом, для некоторого заданного входа недетерминированный алгоритм может иметь экспоненциально много вычислений – относительно длины этого входа.

Несмотря на кажущуюся сложность, у недетерминизма «оптимистическая репутация»: мы предполагаем, что недетерминированный алгоритм всегда сделает правильный выбор – если, конечно, такой выбор вообще существует. То есть недетерминированный алгоритм считается успешно решающим некоторую заданную проблему в том случае, когда для каждого частного случая x рассматриваемой проблемы существует вычисление, дающее правильный результат. Для проблемы принадлежности (Σ, L) это означает, что недетерминированный алгоритм A, решающий (Σ, L), должен иметь *хотя бы одно* допускающее вычисление для каждого $x \in L$ – и наоборот, для любого $y \notin L$ ни одно из вычислений A над входом y допускающим быть не должно.

Вообще говоря, не существует более эффективного моделирования недетерминированного алгоритма A детерминированным алгоритмом B каким-либо иным способом, чем простым выполнением алгоритмом B всех возможных вычислений A. Это имеет место и для конечных автоматов – где B для заданного входного слова фактически производит поиск в ширину в дереве вывода (дереве вычислений) алгоритма A. Поскольку B при чтении слова z сохраняет набор всех тех состояний алгоритма A, в которых может быть прочитано это z, построение (детерминированного) конечного автомата B по заданному недетерминированному автомату A называется супермножественной конструкцией.

Итак, эта глава была посвящена только некоторым элементарным аспектам теории автоматов – и не должна рассматриваться как исчерпывающее введение в эту область. Особенно – когда мы говорим о регулярных языках: в данной главе приведено лишь очень краткое введение в эту тему. Регулярные языки могут быть описаны и представлены не только конечными автоматами, но и некоторыми другими формализмами –

[36] Здесь *и всюду ниже* под термином «детерминированные аналоги алгоритмов» подразумеваются детерминированные алгоритмы, которые для каждого возможного входа дают такой же результат работы. При этом *не* требуется, чтобы *действия* алгоритмов были теми же самыми. *(Прим. перев.)*

[37] Отметим, что число этих возможных действий всегда конечно – и, возможно, равно 0.

которые, в отличие от конечных автоматов, *не* основаны на вычислительных моделях. Самые важные из этих формализмов следующие[38]:

- регулярные (праволинейные) грамматики – как механизм генерации слов заданного регулярного языка;
- регулярные выражения – как алгебраическое представление языка.

Более полную информация о классе регулярных языков можно найти в учебнике Хопкрофта, Мотвани и Ульмана [27].

[38] Отметим, что все эти формализмы, в том числе конечные автоматы, имеют аналоги и в более широких классах языков. Например – в классе контекстно-свободных языков, которые наиболее близки к описанию «реальных» языков программирования. *(Прим. перев.)*

Только тот, у кого есть терпение
доводить до конца простые дела,
сможет овладеть искусством
легко делать сложные.
Ф. Шиллер

4

Машины Тьюринга

4.1 Цели и задачи главы

С давних пор всё происходило так: если какой-нибудь математик хотел объяснить свой подход к решению некоторой проблемы, то он должен был описать этот подход формально – как специальный математический метод. Преимущество подробного формального описания заключается в том, что «пользователю» для успешного применения метода к частным случаям проблемы не обязательно понимать, как и почему этот метод работает. Единственным требованием для успешного применения является понимание формального языка, с помощью которого представлен сам метод. И не удивительно, что задолго до появления компьютеров учёные-математики связывали разрешимость математических проблем с существованием общих методов их автоматического[1] решения.

А создание и развитие компьютеров привело к описанию подобных методов решения с помощью компьютерных программ. Теперь в качестве формализма для описания методов решения обычно используются языки программирования – но при этом могут «остаться за кадром» наиболее важные особенности первоначального описания метода. Но компьютер не обладает интеллектом – и, следовательно, не может понимать ни самой проблемы, ни методов, необходимых для её решения. Однако, несмотря на это, компьютер может *выполнить* программу[2] для некоторых конкретных входных данных – решая тем самым соответствующий частный случай проблемы. Именно поэтому можно говорить об автоматической – или алгоритмической – разрешимости проблем. Чтобы показать, что некоторая проблема автоматически разрешима, достаточно найти метод, решающий эту проблему – и записать этот метод в виде программы (или алгоритма). Именно поэтому нам не нужно давать формальное определение термина «алгоритм» («программа») для формулировки *положительных* утверждений об алгоритмической разрешимости проблем: достаточно дать грубое, обычно неформальное описание алгоритма – после чего он может быть легко реализован в виде компьютерной программы.

Потребность дать точное, формальное определение понятия «алгоритм» – как метода решения проблем – не возникала до тех пор, пока математики не начали рас-

[1] Сегодня мы сказали бы – «алгоритмического».

[2] Предназначенную для автоматического решения рассматриваемой математической проблемы.

сматривать доказательства *неразрешимости* некоторых из них.[3] В связи с этим были предложены и разработаны несколько формальных определений этого понятия – причём вскоре было показано, что все эти определения эквивалентны. Кроме того, любой язык программирования можно рассматривать как выполнимую формализацию этой алгоритмической разрешимости. Но такая формализация не очень удобна для доказательства *не*существования алгоритмов решения конкретных проблем – потому что языки программирования, обладающие т. н. «дружественным интерфейсом» (т. е. удобством для пользователя), содержат слишком сложные команды. Поэтому нам нужны более простые модели алгоритмов – те, которые используют только элементарные команды, но при этом обладают полной вычислительной мощью программ, написанных на любом языке программирования высокого уровня.

Именно такой моделью является *машина Тьюринга*; она и стала стандартом в теории вычислимости[4]. Цель данной главы – введение этой модели и формулировка тех её свойств, которые необходимы для рассмотрения в следующих главах основ теории вычислимости и теории сложности.

Глава организована следующим образом. В разделе 4.2 мы вводим формальную модель машины Тьюринга и рассматриваем практические приёмы работы с ней.[5] Раздел 4.3 посвящён многоленточным версиям машины Тьюринга, являющимся основной вычислительной моделью теории сложности. В этом же разделе мы обсуждаем эквивалентность машин Тьюринга и языков программирования. В разделе 4.4 вводятся недетерминированные машины Тьюринга и исследуется возможность моделирования недетерминированных машин детерминированными. Кодированию машин Тьюринга в алфавите Σ_{bool} посвящён раздел 4.5.

4.2 Формальная модель машин Тьюринга

Машину Тьюринга можно рассматривать как обобщение конечного автомата. Она состоит из (рис. 4.1):

- конечного устройства управления, содержащего программу;
- ленты, которая используется одновременно как входное устройство и как память;
- читающе-пишущей головки, которая может двигаться в обоих направлениях.

Сходство машины Тьюринга и конечного автомата заключается в использовании конечного набора состояний (представляющих собой программу) и в ленте, содержащей в начале вычислений входное слово. Главное же различие между машиной Тьюринга и конечным автоматом заключается в самом использовании ленты. В то время как конечный автомат может только читать ленту, машина Тьюринга может использовать ленту как память, записывая на ней необходимые символы. Эта лента рассматривается как бесконечная – в том смысле, что машина Тьюринга в любом своём вычислении может использовать произвольно большую (но конечную) часть этой ленты.[6] С

[3] Т. е. формулировать *отрицательные* утверждения об алгоритмической разрешимости проблем. *(Прим. перев.)*

[4] Называемой также теорией алгоритмической разрешимости.

[5] Иными словами – технологию создания программ на таком необычном (для программиста-практика) языке. *(Прим. перев.)*

[6] Это означает, что машина Тьюринга может использовать столь много элементов ленты (ячеек памяти), сколько это необходимо. Всё это подобно реальным компьютерным про-

технической точки зрения это различие достигается путём замены читающей головки конечного автомата на «читающе-пишущую» головку машины Тьюринга – с дополнительной возможностью двигаться налево. Инструкция (элементарная операция) машины Тьюринга может быть описана как действие со следующими аргументами:

- текущим состоянием машины;
- символом ячейки ленты, который в данный момент обозревает читающе-пишущая головка.

В зависимости от значений этих аргументов машина Тьюринга выполняет следующие действия:

- изменяет своё состояние (т. е. переходит в новое состояние);
- записывает символ в ту ячейку ленты, где находится читающе-пишущая головка (это действие может рассматриваться как замена читаемого символа на новый);
- перемещает головку на одну ячейку направо или налево – либо не выполяет такого перемещения (оставляет головку на месте).

При этом нужно иметь в виду такие важные технические особенности:

- Крайний левый символ ленты – так называемый **левый граничный маркер** ¢. Этот символ не может быть заменён машиной Тьюринга на какой-либо иной, и, кроме того, головка машины Тьюринга после прочтения этого символа не может передвинуться налево. Использование левого граничного маркера ¢ позволяет осуществлять перебор ячеек ленты слева направо – перебор, начинающийся с нулевой ячейки ленты, которая содержит символ ¢.
- В правом направлении лента счиается бесконечной.[7]

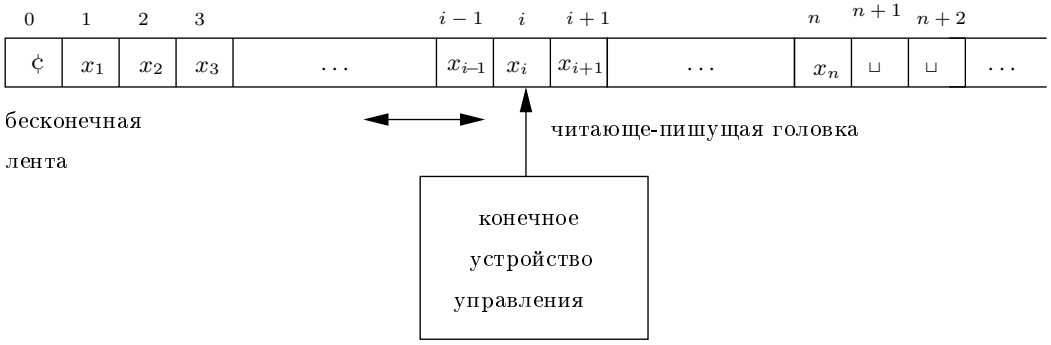


Рис. 4.1.

граммам – где можно использовать необходимое число переменных, которое может увеличиваться в зависимости от длины входа.

[7] Заметим, что в пределах конечного отрезка времени машина Тьюринга может посетить не более чем конечное число ячеек – и, таким образом, фактический размер её памяти (т. е. число ячеек, которые содержат символы, отличные от ⊔), всегда конечен. Смысл рассмотрения памяти неограниченного объёма (т. е. бесконечной ленты) заключается в том, чтобы всегда иметь столь много памяти, сколь это необходимо для конкретной программы.

Теперь дадим *формальное* определение машины Тьюринга – сделаем это тем же способом, который мы использовали для конечных автоматов. Сначала опишем главные компоненты и набор инструкций (команд, элементарных действий). Далее выберем представление конфигураций и определим шаг вычисления – как бинарное отношение на множестве введённых конфигураций. И, наконец, дадим определения вычисления и языка, принимаемого некоторой заданной машиной Тьюринга.

Определение 4.1. Машина Тьюринга (*сокращённо* – **MT**) – *это семёрка* $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, *где*

- Q – *конечное множество, называемое* **множеством состояний** *машины* M;
- Σ – **входной алфавит**, *причём* ¢ *и символ пробела* $\sqcup$ *не входят в* Σ
  {Σ *используется для представления входных слов – тем же способом, что и в случае конечных автоматов*};
- Γ – *алфавит, называемый* **рабочим алфавитом**, *где* $\Sigma \subseteq \Gamma$, ¢, $\sqcup \in \Gamma$
  {Γ *содержит все символы, которые могут появляться в ячейках памяти ленты, т. е. все символы, используемые в памяти как значения переменных*};
- $\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{\text{L}, \text{R}, \text{N}\}$ – *отображение, называемое* **функцией переходов машины** $\boldsymbol{M}$, *обладающей свойством*

  $$\delta(q, ¢) \in Q \times \{¢\} \times \{\text{R}, \text{N}\}$$

  для всех $q \in Q - \{q_{\text{accept}}, q_{\text{reject}}\}$
  {δ *определяет команды машины* M. M *может выполнить команду* $(q, X, Z) \in Q \times \Gamma \times \{\text{L}, \text{R}, \text{N}\}$, *если* M *находится в состоянии* p, *читает символ* $Y \in \Gamma$, *и при этом* $\delta(p, Y) = (q, X, Z)$. *Выполнение этой команды подразумевает переход из состояния* p *в состояние* q, *замену на ленте символа* Y *на* X, *а также перемещение читающе-пишущей головки согласно* Z. *Если* $Z = \text{L}$, *то головка* M *должна переместиться на одну ячейку налево, если* $Z = \text{R}$ – *то на одну ячейку направо, а* $Z = \text{N}$ *означает, что головка остаётся на месте. Свойство* $\delta(q, ¢) \in Q \times \{¢\} \times \{\text{R}, \text{N}\}$ *означает, что* M *никогда не перезаписывает левый граничный маркер* ¢, *а также не перемещается налево относительно него*};
- $q_0 \in Q$ – **инициальное сотояние**;
- $q_{\text{accept}} \in Q$ – **допускающее состояние**
  {M *содержит в точности одно допускающее состояние. Достижение машиной* M *состояния* q_{accept} *означает принятие входа – причём независимо от положения головки на ленте. После достижения* q_{accept} *вычисления прерываются.*[8]};
- $q_{\text{reject}} \in Q - \{q_{\text{accept}}\}$ – **отклоняющее состояние**
  {*Если* M *достигает состояния* q_{reject}, *то* M *останавливается и отклоняет (не принимает) входное слово.*}.

Конфигурация C *машины* M – *элемент множества*

$$\mathbf{conf}(\boldsymbol{M}) = \{¢\} \cdot \Gamma^* \cdot Q \cdot \Gamma^+ \cup Q \cdot \{¢\} \cdot \Gamma^*.$$

{*Конфигурация* $w_1 q a w_2$, $w_1 \in \{¢\}\Gamma^*$, $w_2 \in \Gamma^*$, $a \in \Gamma$, $q \in Q$ (*рис. 4.2*) – *это полное описание следующей ситуации: машина* M *находится в состоянии* q, *содержание*

[8] Среди прочего, это означает, что – в отличие от случая конечных автоматов – машина Тьюринга M не обязана прочесть *весь* вход перед тем, как решить, принять его или отклонить.

ленты $-\text{¢}w_1aw_2\sqcup\sqcup\sqcup\ldots$, *а головка указывает на* $(|w_1|+1)$*-ю ячейку ленты, в которой находится символ* a. *Конфигурация* $p\text{¢}w$, $p \in Q$, $w \in \Gamma^*$ *описывает ситуацию, в которой содержание ленты* $-\text{¢}w_\sqcup\sqcup\sqcup\ldots$, *а головка находится в* 0*-й ячейке ленты, на левом граничном маркере* ¢.$\}$[9]

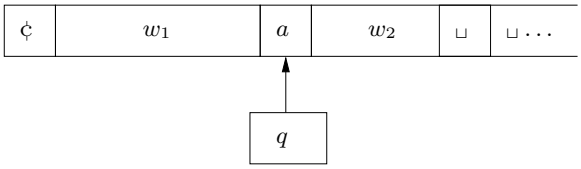


Рис. 4.2.

Стартовая конфигурация *машины* M *для входного слова* x *есть* $q_0\text{¢}x$.
Шаг $\boldsymbol{M}$ – *бинарное отношение* $\vdash_M$ *на множестве конфигураций (т. е.* $\vdash_M \subseteq \mathrm{conf}(M) \times \mathrm{conf}(M)$*), определённое следующим образом.*

- $x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n \vdash_M x_1x_2\ldots x_{i-1}px_{i+1}\ldots x_n$,
  если $\delta(q,x_i) = (p,y,\mathrm{N})$ *(рис. 4.3 a);*
- $x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n \vdash_M x_1x_2\ldots x_{i-2}px_{i-1}yx_{i+1}\ldots x_n$,
  если $\delta(q,x_i) = (p,y,\mathrm{L})$ *(рис. 4.3 b);*
- $x_1x_2\ldots x_{i-1}qx_ix_{i+1}\ldots x_n \vdash_M x_1x_2\ldots x_{i-1}ypx_{i+1}\ldots x_n$,
  если $\delta(q,x_i) = (p,y,\mathrm{R})$ *для* $i < n$ *(рис. 4.3 c);*
- $x_1x_2\ldots x_{n-1}qx_n \vdash_M x_1x_2\ldots x_{n-1}yp\sqcup$,
  если $\delta(q,x_n) = (p,y,\mathrm{R})$ *(рис. 4.3 d).*

Вычисление машины $\boldsymbol{M}$ – *последовательность конфигураций (возможно, бесконечная)* $C_0, C_1, C_2, \ldots$, *такая что* $C_i \vdash_M C_{i+1}$ *для всех* $i = 0, 1, 2, \ldots$ *Если*

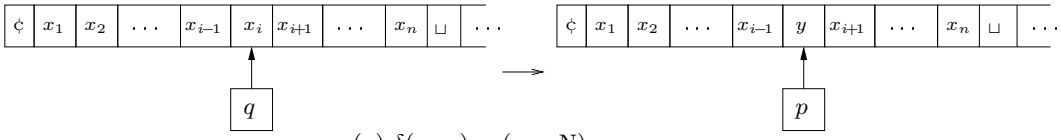
$$C_0 \vdash_M C_1 \vdash_M \cdots \vdash_M C_i$$

для некоторого $i \in \mathbb{N}_0$, *то мы будем писать*

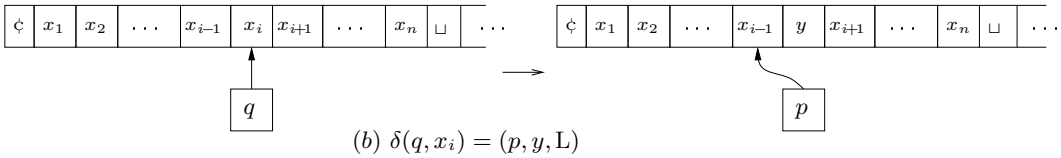
$$C_0 \vdash_M^* C_i.$$

Вычисление машины $\boldsymbol{M}$ **над входом** $\boldsymbol{x}$ – *это такое вычисление, которое начинается с инициальной конфигурации* $C_0 = q_0\text{¢}x$, *причём либо является бесконечным, либо останавливается в некоторой конфигурации* w_1qw_2, *где* $q \in \{q_{\mathrm{accept}}, q_{\mathrm{reject}}\}$.
Вычисление машины M *над* x *называется* **допускающим**, *если оно заканчивается в некоторой допускающей конфигурации* $w_1q_{\mathrm{accept}}w_2$. *Вычисление* M *над* x *называется* **отклоняющим**, *если оно заканчивается в некоторой отклоняющей конфигурации* $w_1q_{\mathrm{reject}}w_2$. *Если вычисление машины* M *над* x *является допускающим*

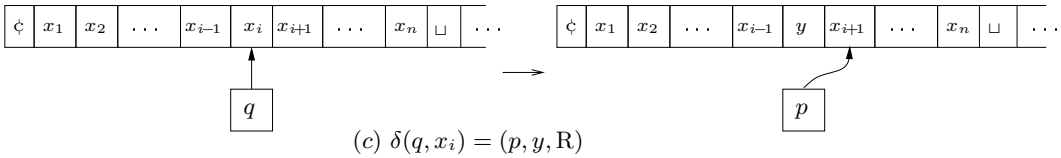
[9] Вообще говоря, имеется выбор между несколькими альтернативными путями представления конфигурации машины Тьюринга; кроме приведённого здесь, можно выбрать, например, такое представление – кажущееся на первый взгляд более простым. Тройка $(q,w,i) \in Q \times \Gamma^* \times \mathbb{N}_0$ описывает обобщённое состояние следующим образом: M находится во (внутреннем) состоянии q, $w\sqcup\sqcup\sqcup\ldots$ – содержание ленты, а головка указывает на i-ю ячейку ленты.



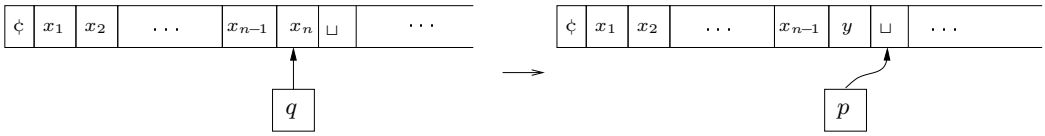
(*a*) $\delta(q, x_i) = (p, y, \mathrm{N})$



(*b*) $\delta(q, x_i) = (p, y, \mathrm{L})$



(*c*) $\delta(q, x_i) = (p, y, \mathrm{R})$



(*d*) $\delta(q, x_n) = (p, y, \mathrm{R})$

Рис. 4.3.

[*отклоняющим*], *то мы говорим, что* M **допускает** [**отклоняет**] x. *Если вычисление машины* M *над* x – *отклоняющее или бесконечное, то мы говорим, что* M **не принимает**[10] *слово* x.

Язык $\boldsymbol{L(M)}$, *допускаемый машиной* M, *определяется как*

$$\begin{aligned} \boldsymbol{L(M)} &= \{w \in \Sigma^* \mid q_0 ¢ w \mid\!\!\overset{*}{}_{M}\; y q_{\mathrm{accept}} z, \ \textit{для некоторых } y, z \in \Gamma^*\} \\ &= \{w \in \Sigma^* \mid \ M \textit{ принимает } w \ \}. \end{aligned}$$

Будем говорить, что M **вычисляет** *функцию* $F : \Sigma^* \to \Gamma^*$, *если*

$$\textit{для всех } \ x \in \Sigma^* \ \ \textit{выполнено условие } \ q_0 ¢ x \mid\!\!\overset{*}{}_{M}\; q_{\mathrm{accept}} ¢ F(x).$$

Язык называется **рекурсивно перечислимым**, *если существует некоторая машина Тьюринга* M, *такая что* $L = L(M)$.

Множество

$$\boldsymbol{\mathcal{L}}_{\mathrm{RE}} = \{L(M) \mid M \textit{ – некоторая } \mathrm{MT}\}$$

называется **классом рекурсивно перечислимых языков**.

[10] Заметим, что в данном случае мы различаем понятия «отклоняет» и «не принимает». Точнее, отклонение – частный случай непринятия.

Язык $L \subseteq \Sigma^*$ называется **рекурсивным** *– а проблема принадлежности (Σ, L) называется* **разрешимой** *– если $L = L(M)$ для некоторой машины Тьюринга M, такой что для любого $x \in \Sigma^*$:*

- $q_0 ¢ x \mid\!\!\overset{*}{}_M\, y q_{\text{accept}} z$, *$y, z \in \Gamma^*$, если $x \in L$;*
- $q_0 ¢ x \mid\!\!\overset{*}{}_M\, u q_{\text{reject}} v$, *$u, v \in \Gamma^*$, если $x \notin L$.*

{Это означает, что у машины M нет бесконечных вычислений.}
Другими словами – если выполнены эти два условия, то мы говорим, что $\boldsymbol{M}$ **останавливается для любого входа**, *или* $\boldsymbol{M}$ **всегда останавливается**.
{Машина Тьюринга, которая всегда останавливается, – это формальная модель понятия «алгоритм».}
Множество

$$\boldsymbol{\mathcal{L}}_{\text{R}} = \{L(M) \mid M \text{ – некоторая МТ, которая всегда останавливается}\}$$

– это **класс рекурсивных (алгоритмически распознаваемых) языков**.
Для двух заданных алфавитов Σ_1 и Σ_2 некоторая функция $F : \Sigma_1^* \to \Sigma_2^*$ называется **вычислимой**, *если существует некоторая машина Тьюринга, которая вычисляет F, и при этом всегда останавливается.*

Машина Тьюринга, которая всегда останавливается, представляет собой алгоритм – т. е. программу, которая всегда завершает работу и выдаёт правильный ответ.

Упражнение 4.2. Приведите альтернативную формулировку определения машины Тьюринга – с использованием тройки $(q, ¢w, i) \in Q \times \{¢\}\Gamma^* \times \mathbb{N}_0$ для описания конфигурации. Тройка $(q, ¢w, i)$ описывает ситуацию, в которой машина Тьюринга находится в состоянии q, содержание ленты – $¢w␣␣␣\ldots$, а головка находится в i-й ячейке ленты. Дайте определение шага (вычисления) и всего вычисления, используя это представление данных.

Далее приведём описания нескольких конкретных машин Тьюринга – и, аналогично случаю конечных автоматов, нас будет интересовать их графическое представление. Итак, пусть

$$L_{\text{middle}} = \{w \in (\Sigma_{bool})^* \mid w = x1y, \text{ где } |x| = |y|\}.$$

(Следовательно, L_{middle} содержит все слова нечётной длины с символом 1 посередине.)

Упражнение 4.3. Докажите, что $L_{\text{middle}} \notin \mathcal{L}(\text{КА})$.

Теперь опишем машину M, такую что $L(M) = L_{\text{middle}}$. Идея построения заключается в том, чтобы сначала проверить, является ли длина входа нечётной, а затем найти средний символ входого слова и проверить, является ли этот символ единицей. Итак, положим $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, где:
$Q = \{q_0, q_{\text{even}}, q_{\text{odd}}, q_{\text{accept}}, q_{\text{reject}}, q_A, q_B, q_1, q_{\text{left}}, q_{\text{right}}, q_{\text{middle}}\}$;
$\Sigma = \{0, 1\}$;
$\Gamma = \Sigma \cup \{¢, ␣\} \cup (\Sigma \times \{A, B\}) = \{0, 1, ¢, ␣, \binom{0}{A}, \binom{0}{B}, \binom{1}{A}, \binom{1}{B}\}$;

$$\begin{aligned}
\delta(q_0, ¢) &= (q_{\text{even}}, ¢, \text{R}),\\
\delta(q_0, a) &= (q_{\text{reject}}, a, \text{N}) \text{ для всех } a \in \{0, 1, ␣\},\\
\delta(q_{\text{even}}, b) &= (q_{\text{odd}}, b, \text{R}) \text{ для всех } b \in \Sigma,
\end{aligned}$$

$$\begin{aligned}
\delta(q_{\text{even}}, \sqcup) &= (q_{\text{reject}}, \sqcup, \text{N}),\\
\delta(q_{\text{odd}}, b) &= (q_{\text{even}}, b, \text{R}) \text{ для всех } b \in \Sigma,\\
\delta(q_{\text{odd}}, \sqcup) &= (q_B, \sqcup, \text{L}).
\end{aligned}$$

Мы видим, что после чтения любого префикса чётной [нечётной] длины M переходит в состояние q_{even} [q_{odd}]. Следовательно, если M читает символ $\sqcup$ в состоянии q_{even}, то входное слово имеет чётную длину и должно быть отклонено. Если же M читает символ $\sqcup$ в состоянии q_{odd}, то M перемещается в состояние q_B – в котором запускается вторая стадия вычисления.

На этой второй стадии M определяет середину входного слова, одновременно заменяя (переписывая) крайний левый символ $a \in \{0,1\}$ на $\binom{a}{B}$ и крайний правый символ $b \in \Sigma$ на $\binom{b}{A}$. Всё это может быть выполнено с помощью следующих переходов:

$$\begin{aligned}
\delta(q_B, a) &= (q_1, \tbinom{a}{B}, \text{L}) \text{ для всех } a \in \{0,1\},\\
\delta(q_1, a) &= (q_{\text{left}}, a, \text{L}) \text{ для всех } a \in \{0,1\},\\
\delta(q_1, c) &= (q_{\text{middle}}, c, R) \text{ для всех } c \in \{¢, \tbinom{0}{A}, \tbinom{1}{A}\},\\
\delta(q_{\text{middle}}, \tbinom{0}{B}) &= (q_{\text{reject}}, 0, \text{N}),\\
\delta(q_{\text{middle}}, \tbinom{1}{B}) &= (q_{\text{accept}}, 1, \text{N}),\\
\delta(q_{\text{left}}, a) &= (q_{\text{left}}, a, \text{L}) \text{ для всех } a \in \{0,1\},\\
\delta(q_{\text{left}}, c) &= (q_A, c, \text{R}) \text{ для всех } c \in \{\tbinom{0}{A}, \tbinom{1}{A}, ¢\},\\
\delta(q_A, b) &= (q_{\text{right}}, \tbinom{b}{A}, \text{R}) \text{ для всех } b \in \{0,1\},\\
\delta(q_{\text{right}}, b) &= (q_{\text{right}}, b, \text{R}) \text{ для всех } b \in \{0,1\},\\
\delta(q_{\text{right}}, d) &= (q_B, d, \text{L}) \text{ для всех } d \in \{\tbinom{0}{B}, \tbinom{1}{B}\}.
\end{aligned}$$

Отсутствующие пары аргументов (например, $(q_{\text{right}}, ¢)$) не могут встретиться ни в каком вычислении – и, следовательно, мы можем для них положить значение функции δ равным q_{reject}.

Если для любых $q, p \in Q$, $a, b \in \Sigma$ и $X \in \{\text{L}, \text{R}, \text{N}\}$ мы будем применять графическое представление одной команды

$$\delta(q, a) = (p, b, X),$$

приведённое на рис. 4.4, – то получим графическое представление всей машины M, изображённое на рис. 4.5.

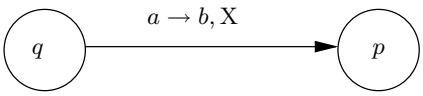


Рис. 4.4.

Графическое представление машины Тьюринга похоже на представление конечных автоматов. Различие состоит только в пометках дуг – потому что теперь мы должны

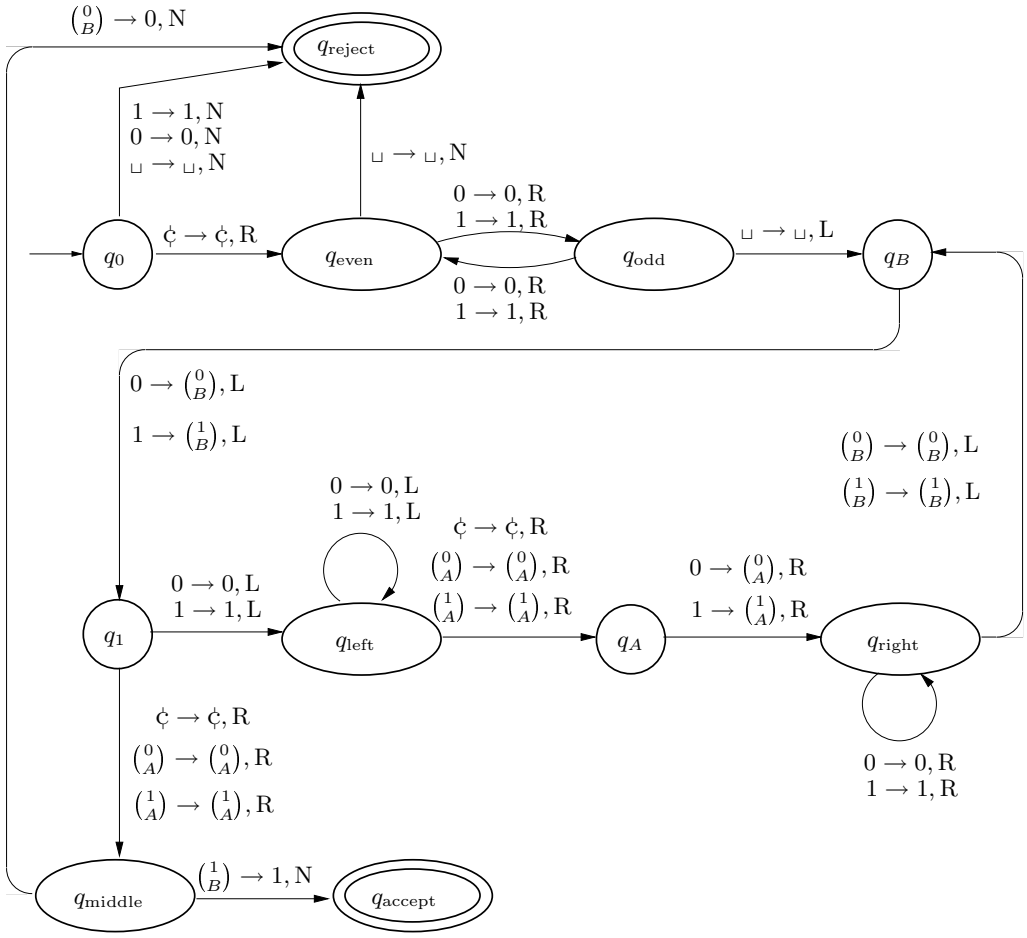


Рис. 4.5.

включить для каждого читаемого символа a как новый символ b, так и направление X движения головки по ленте.

Рассмотрим работу M над входным словом $x = 1001101$. Первая стадия вычисления над x такова:

$$\begin{aligned} q_0\text{¢}1001101 \;&\vdash_{\overline{M}}\; \text{¢}q_{\mathrm{even}}1001101 \vdash_{\overline{M}} \text{¢}1q_{\mathrm{odd}}001101 \vdash_{\overline{M}} \text{¢}10q_{\mathrm{even}}01101 \\ &\vdash_{\overline{M}}\; \text{¢}100q_{\mathrm{odd}}1101 \vdash_{\overline{M}} \text{¢}1001q_{\mathrm{even}}101 \vdash_{\overline{M}} \text{¢}10011q_{\mathrm{odd}}01 \\ &\vdash_{\overline{M}}\; 100110q_{\mathrm{even}}1 \vdash_{\overline{M}} \text{¢}1001101q_{\mathrm{odd}}\sqcup \vdash_{\overline{M}} \text{¢}100110q_B1 \end{aligned}$$

Окончание первой стадии работы в состоянии q_B означает, что x имеет нечётную длину.

Теперь, на второй стадии вычисления, M альтернативно заменяет символы a на правой стороне ленты (у правой границы) на символы $\binom{a}{B}$, а на левой стороне ленты (у левой границы) на символы $\binom{a}{A}$:

$$\text{¢}100110q_B1 \;\vdash_{\overline{M}}\; \text{¢}10011q_1 0\tbinom{1}{B} \vdash_{\overline{M}} \text{¢}1001q_{\mathrm{left}}10\tbinom{1}{B}$$

$$
\begin{aligned}
&\vdash_{\overline{M}} \text{¢}100q_{\text{left}}110\binom{1}{B} \vdash_{\overline{M}} \text{¢}10q_{\text{left}}0110\binom{1}{B}\\
&\vdash_{\overline{M}} \text{¢}1q_{\text{left}}00110\binom{1}{B} \vdash_{\overline{M}} \text{¢}q_{\text{left}}100110\binom{1}{B}\\
&\vdash_{\overline{M}} q_{\text{left}}\text{¢}100110\binom{1}{B} \vdash_{\overline{M}} \text{¢}q_A100110\binom{1}{B}\\
&\vdash_{\overline{M}} \text{¢}\binom{1}{A}q_{\text{right}}00110\binom{1}{B} \vdash_{\overline{M}} \text{¢}\binom{1}{A}0q_{\text{right}}0110\binom{1}{B}\\
&\vdash^{*}_{\overline{M}} \text{¢}\binom{1}{A}00110q_{\text{right}}\binom{1}{B} \vdash_{\overline{M}} \text{¢}\binom{1}{A}0011q_B0\binom{1}{B}\\
&\vdash_{\overline{M}} \text{¢}\binom{1}{A}001q_11\binom{0}{B}\binom{1}{B} \vdash_{\overline{M}} \text{¢}\binom{1}{A}00q_{\text{left}}11\binom{0}{B}\binom{1}{B}\\
&\vdash^{*}_{\overline{M}} \text{¢}q_{left}\binom{1}{A}0011\binom{0}{B}\binom{1}{B} \vdash_{\overline{M}} \text{¢}\binom{1}{A}q_A0011\binom{0}{B}\binom{1}{B}\\
&\vdash_{\overline{M}} \text{¢}\binom{1}{A}\binom{0}{A}q_{\text{right}}011\binom{0}{B}\binom{1}{B}\\
&\vdash^{*}_{\overline{M}} \text{¢}\binom{1}{A}\binom{0}{A}011q_{\text{right}}\binom{0}{B}\binom{1}{B}\\
&\vdash_{\overline{M}} \text{¢}\binom{1}{A}\binom{0}{A}01q_B1\binom{0}{B}\binom{1}{B}\\
&\vdash_{\overline{M}} \text{¢}\binom{1}{A}\binom{0}{A}0q_11\binom{1}{B}\binom{0}{B}\binom{1}{B}\\
&\vdash^{*}_{\overline{M}} \text{¢}\binom{1}{A}q_{\text{left}}\binom{0}{A}01\binom{1}{B}\binom{0}{B}\binom{1}{B}\\
&\vdash_{\overline{M}} \text{¢}\binom{1}{A}\binom{0}{A}q_A01\binom{1}{B}\binom{0}{B}\binom{1}{B}\\
&\vdash_{\overline{M}} \text{¢}\binom{1}{A}\binom{0}{A}\binom{0}{A}q_{\text{right}}1\binom{1}{B}\binom{0}{B}\binom{1}{B}\\
&\vdash_{\overline{M}} \text{¢}\binom{1}{A}\binom{0}{A}\binom{0}{A}1q_{\text{right}}\binom{1}{B}\binom{0}{B}\binom{1}{B}\\
&\vdash_{\overline{M}} \text{¢}\binom{1}{A}\binom{0}{A}\binom{0}{A}q_B1\binom{1}{B}\binom{0}{B}\binom{1}{B}\\
&\vdash_{\overline{M}} \text{¢}\binom{1}{A}\binom{0}{A}q_1\binom{0}{A}\binom{1}{B}\binom{1}{B}\binom{0}{B}\binom{1}{B}\\
&\vdash_{\overline{M}} \text{¢}\binom{1}{A}\binom{0}{A}\binom{0}{A}q_{\text{middle}}\binom{1}{B}\binom{1}{B}\binom{0}{B}\binom{1}{B}\\
&\vdash_{\overline{M}} \text{¢}\binom{1}{A}\binom{0}{A}\binom{0}{A}q_{\text{accept}}1\binom{1}{B}\binom{0}{B}\binom{1}{B}.
\end{aligned}
$$

Данная МТ заканчивает работу в состоянии q_{accept}. Это означает, что в средней позиции ленты находится символ 1 – т. е. МТ принимает слово 1001101.

Упражнение 4.4. Опишите вычисления машины Тьюринга M, представленной на рис. 4.5, над входными словами 010011 и 101.

Упражнение 4.5. При работе машины Тьюринга M, представленной на рис. 4.5, сохраняется полная информация об исходном входном слове. Используйте это свойство для обобщения M – т. е. построения некоторой машины M', которая принимает язык

$$L(M') = \{w \in \{0,1\}^* \mid w = x1x \text{ для некоторого } x \in \{0,1\}^*\}.$$

Теперь рассмотрим язык $L_{\mathcal{P}} = \{0^{2^n} \mid n \in \mathbb{N}\}$. Программа для машины Тьюринга, принимающей L, может быть построена на основе следующей стратегии.

1. Головка перемещается по ленте от левого граничного маркера ¢ к первому символу $\sqcup$ справа – при этом удаляя каждый второй символ 0 (точнее, заменяя каждый второй 0 на символ a). Если число символов 0 на ленте нечётно, то машина останавливается в состоянии q_{reject}. Иначе переходит к шагу 2.
2. За одно перемещение справа (начиная с символа $\sqcup$) налево (к левому граничному маркеру ¢) машина проверяет, встречается ли символ 0 на ленте более 1 раза.
   - Если на ленте ровно один символ 0, то машина принимает вход.

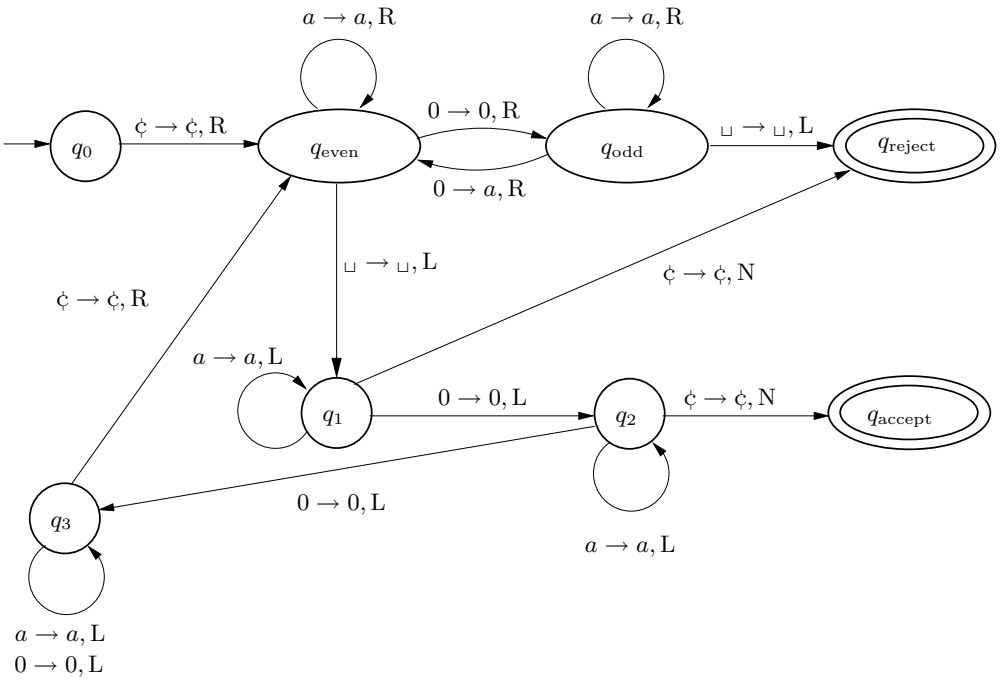


Рис. 4.6.

- Если же существуют по крайней мере два вхождения символа 0, то машина переходит к шагу 1.

Идея описанной стратегии состоит в том, что для любого натурального числа i число 2^i можно несколько раз делить на 2 без остатка – пока не получится значение 1. Соответствующая возможная формализация машины Тьюринга такова: $A = (\{q_0, q_{\text{even}}, q_{\text{odd}}, q_1, q_2, q_3, q_{\text{accept}}, q_{\text{reject}}\}, \{0\}, \{0, a, ¢, ␣\}, \delta_A, q_0, q_{\text{accept}}, q_{\text{reject}})$, функция переходов δ_A приведена на рис. 4.6.

Упражнение 4.6. Другая возможная стратегия распознавания языка $L_{\mathcal{P}}$ заключается в следующем. Во-первых, если $i = 2j$ – некоторое чётное число, то мы переписываем вход 0^i в виде $0^j 1^j$. Затем мы проверяем, является ли чётным j, и если является, то переписываем $0^j 1^j$ в виде $0^{\frac{j}{2}} 1^{\frac{j}{2}} 1^j$ – и так далее. Вход может быть принят только если эта «стратегия деления на два» завершает работу с содержимым ленты 01^{i-1}. А чтобы осуществлять это деление на два с помощью некоторой машины Тьюринга, мы можем использовать описанный выше способ поиска середины входного слова.

Приведите полное описание машины, которая принимает язык $L_{\mathcal{P}}$, используя эту стратегию.

Упражнение 4.7. Разработайте машины Тьюринга для следующих языков:

- $\{a^n b^n \mid n \in \mathbb{N}_0\}$;
- $\{0^{n^2} \mid n \in \mathbb{N}_0\}$;
- $\{w\#w \mid w \in \{0,1\}^*\}$;

- $\{x\#y \mid x, y \in \{0,1\}^*, Number(x) = Number(y) + 1\}$.

Упражнение 4.8. Разработайте машину Тьюринга, которая для любого входного слова $x \in (\Sigma_{\text{bool}})^*$ заканчивает работу в состоянии q_{accept} со следующим содержимым ленты:

- $y \in (\Sigma_{\text{bool}})^*$, причём $Number(y) = Number(x) + 1$;
- $x\#x$;
- $z \in (\Sigma_{\text{bool}})^*$, причём $Number(z) = 2 \cdot Number(x)$;
- $\#\#\#x$.

4.3 Многоленточные машины Тьюринга и тезис Чёрча

Вследствие своей простоты машины Тьюринга представляют собой стандартную вычислительную модель – используемую в теории вычислимости для классификации проблем относительно их рекурсивности или рекурсивной перечислимости. Но эта модель не всегда удобна для задач теории сложности. Главный недостаток введённой модели машины Тьюринга состоит в том, что она не вписывается в общую структуру модели фон-Неймана.[11] Модель фон Неймана требует, чтобы все главные компоненты компьютера – а именно, память для программы, память для данных, центральный процессор, средства ввода – были бы физически независимыми частями компьютера. Однако у машин Тьюринга средства ввода и память суть одно и то же – это просто лента. Вторым недостатком использования машин Тьюринга в теории сложности является т. н. линейность её памяти – т. е. ограниченный доступ к ленте. Если нужно сравнить содержимое двух различных ячеек ленты, то необходимо выполнить столь много операций (шагов вычисления), сколь велико расстояние[12] между этими двумя ячейками.

Следующая модель многоленточной машины Тьюринга до некоторой степени преодолевает вышеупомянутые недостатки – и, вследствие этого, является фундаментальной вычислительной моделью теории сложности.[13] k-ленточная машина Тьюринга для любого заданного натурального k имеет следующие компоненты (рис.4.7):

- конечное устройство управления, выполняющее программу,
- конечную ленту с читающей головкой – как средства ввода,
- k рабочих лент, каждая из которых снабжена читающе/пишущей головкой – как память для хранения данных.

Перед началом вычислений над входным словом w k-ленточная машина Тьюринга всегда находится в следующем обобщённом состоянии (начальной конфигурации):

- конечная входная лента содержит ¢w\$, где ¢ и \$ - символы, которыми помечены соответственно левая и правая границы входного слова;

[11] Называемой также компьютером фон Неймана, «Von Neumann computer».

[12] И если это расстояние велико, то для выполнения простого сравнения необходимо слишком много работы.

[13] Дальнейшее обсуждение применимости модели многоленточной машины Тьюринга в теории сложности приведено в главе 6.

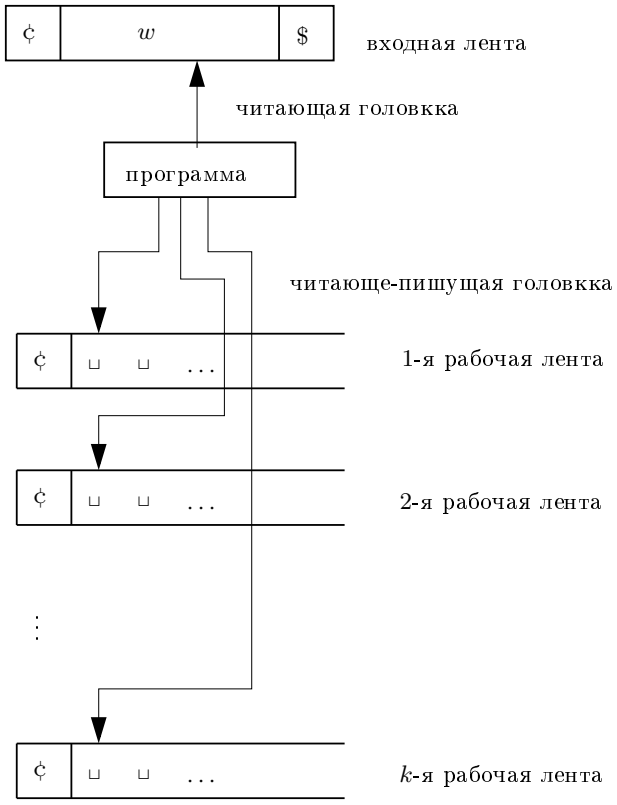


Рис. 4.7.

- читающая головка на входной ленте установлена на левый граничный маркер ¢;
- содержимое каждой рабочей ленты – ¢⊔⊔⊔ ..., и все читающе-пишущие головки рабочих лент установлены на ячейках, содержащих символы ¢;
- конечное устройство управления находится в инициальном состоянии q_0.

Во время вычислений все головки могут перемещаться налево и направо – за исключением перемещения влево от левого граничного маркера ¢, а также, для входной ленты, перемещения вправо от правого граничного маркера \$. Считывающая головка входной ленты не переписывает содержимое ячеек, и, таким образом, начальное состояние входной ленты ¢w\$ остаётся неизменным в течение всего процесса вычислений над словом w. Аналогично (собственно) машинам Тьюринга, содержимое любой ячейки рабочих лент может быть любым символом рабочего алфавита Γ. Ячейки всех $k+1$ лент пронумерованы слева направо – начиная с 0 для ячейки, содержащей левый граничный маркер ¢. Поэтому можно использовать приведённое далее представление конфигураций k-ленточной машина Тьюринга.

Конфигурация

$$(q, w, i, u_1, i_1, u_2, i_2, \ldots, u_k, i_k)$$

является элементом множества

$$Q \times \Sigma^* \times \mathbb{N}_0 \times (\Gamma^* \times \mathbb{N}_0)^k.$$

Она представляет обобщённое состояние машины M:

- M находится в состоянии q;
- содержимое входной ленты – ¢w\$, а её считывающая головка установлена на i-ю ячейку входной ленты (т. е. если $w = a_1 a_2 \dots a_n$ для $a_i \in \Sigma$, то считывающая головка читает символ a_i);
- для любого $j \in \{1, 2, \dots, k\}$ содержимым j-й ленты является ¢$u_j \sqcup\sqcup\sqcup \dots$, при этом $i_j \leq |u_j|$ – позиция ячейки, на которую установлена головка j-й рабочей ленты.

Шаг (вычисления) машины M описывается функцией переходов

$$\delta : Q \times (\Sigma \cup \{¢, \$\}) \times \Gamma^k \to Q \times \{\mathrm{L}, \mathrm{R}, \mathrm{N}\} \times (\Gamma \times \{\mathrm{L}, \mathrm{R}, \mathrm{N}\})^k.$$

Аргументы $(q, a, b_1, \dots, b_k) \in Q \times (\Sigma \cup \{¢, \$\}) \times \Gamma^k$ следующие:

- текущее состояние q;
- символ $a \in \Sigma \cup \{¢, \$\}$, обозреваемый считывающей головкой входной ленты;
- k символов $b_1, \dots, b_k$, читаемых на k рабочих лентах.

Согласно описанию этих аргументов, шаг вычисления машины M соответствует следующим действиям:

- каждый из k символов $b_1, \dots, b_k$, читаемых на рабочих лентах, может быть заменён на некоторый другой символ;
- M переходит в некоторое новое состояние;
- каждая из $k + 1$ головок может переместиться на одну ячейку налево или направо – за исключением случаев чтения соответствующих граничных маркеров.

Если вычисление машины M над словом w заканчивается в состоянии q_{accept}, то M допускает w. M не допускает w, если вычисление M над w бесконечно или заканчивается в состоянии q_{reject}. Мы продолжаем использовать термин «M отклоняет w» – если M достигает q_{reject} в процессе вычисления над словом w.

Мы опускаем формальное определение k-ленточной машины Тьюринга. Строгое определение на основе вышеупомянутого неформального описания k-ленточной машины Тьюринга – рутинный вопрос, который мы оставляем читателю как упражнение.

Упражнение 4.9. Приведите точное формальное определение k-ленточной машины Тьюринга – следуя определению (собственно) машины Тьюринга.

Будем использовать сокращение ***k*-ленточная-МТ**; k-ленточная-МТ называется **многоленточной машиной Тьюринга**, ММТ. Так как инструкции (команды) ММТ немного сложнее, чем инструкции МТ, то можно было бы ожидать, что многоленточная машина Тьюринга в состоянии решать некоторые проблемы более простым или более эффективным способом, чем (собственно) машина Тьюринга.

В качестве примера рассмотрим язык

$$L_{\text{equal}} = \{w\#w \mid w \in (\Sigma_{\text{bool}})^*\}.$$

Пусть $x\#y$ – входное слово. Некоторая МТ, который сравнивает x (префикс перед первым вхождением символа #) с y (суффикс после первого вхождения #) должна много раз перемещаться по своей ленте.

1-ленточная-МТ A допускает язык L_{equal}, используя следующую стратегию.

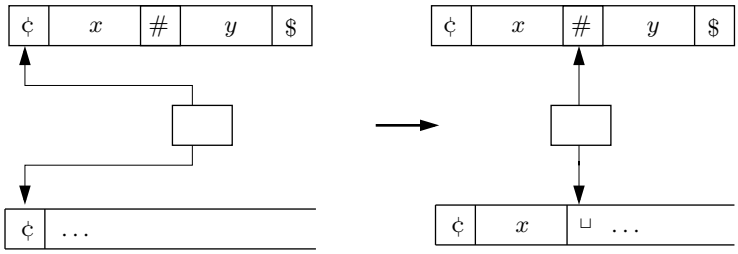


Рис. 4.8.

1. A проверяет, представим ли вход в виде $x\#y$, где $x, y \in (\Sigma_{\text{bool}})^*$.[14] Если нет, то A отклоняет этот вход.
2. Для входа $x\#y$ машина A делает копию x на рабочей ленте – т. е. в конце этого этапа вычислений рабочая лента содержит $¢x$ (рис. 4.8).
3. A устанавливает головку рабочей ленты на левый граничный маркер ¢, а головка входной ленты устанавливается на #. После этого A одновременно перемещает обе головки направо – и при этом сравнивает x и y. Если на каком-либо шаге вычисления головки читают различные символы, то A делает вывод, что $x \neq y$, – и отклоняет вход. Если же все пары символов равны, и обе головки достигают символа ␣ на одном и том же шаге вычисления, то A принимает вход.

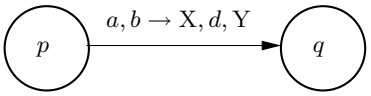


Рис. 4.9.

Графическое представление перехода

$$\delta(p, a, b) = (q, \mathrm{X}, d, \mathrm{Y})$$

некоторой 1-ленточной-МТ приведено на рис. 4.9. Переход из состояния p в состояние q осуществляется в том случае, когда символ a читается на входной ленте, а символ b – на рабочей. Символ b при этом заменяется символом d. $\mathrm{X} \in \{\mathrm{L}, \mathrm{R}, \mathrm{N}\}$ определяет движение головки на входной ленте, а $\mathrm{Y} \in \{\mathrm{L}, \mathrm{R}, \mathrm{N}\}$ – на рабочей.

На основе графического представления инструкции (перехода), показанного на рис. 4.9, мы можем получить и графическое описание вышеприведённой 1-ленточной-МТ, которая распознаёт язык L_{equal} (рис. 4.10). Состояния q_0, q_1, q_2 и q_{reject} используются для выполнннения первого этапа этой стратегии. Если вход содержит в точности один символ #, то M достигает состояния q_2 с головкой входной ленты, установленной на последнем символе входа. В противном случае M завершает работу в состоянии q_{reject}. Состояние q_2 используется для возвращения считывающей головки к левому граничному маркеру ¢. С помощью состояния q_{copy} машина M копирует на рабочую

[14] Т. е. A проверяет, содержит ли вход в точности один символ #.

ленту префикс входного слова – вплоть до символа # (рис. 4.10). Состояние q_{adjust} используется для возвращения головки рабочей ленты к левому граничному маркеру ¢. Сравнение подслов x и y входа $x\#y$ выполняется в состоянии q_{compare}. Если $x = y$, то M останавливается в состоянии q_{accept}. А если x и y отличаются в каком-либо символе (или просто имеют различные длины), то вычисление заканчивается в состоянии q_{reject}.

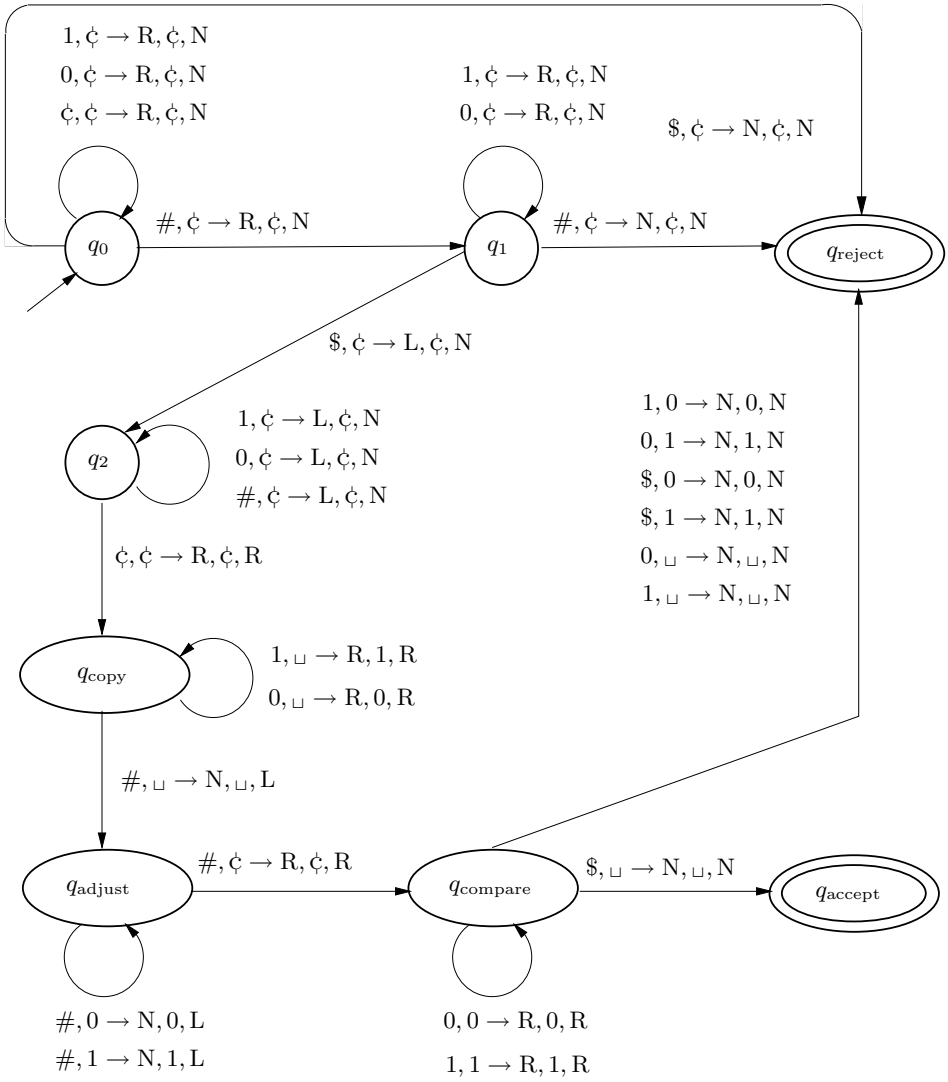


Рис. 4.10.

Упражнение 4.10. Опишите неформально, а затем графически, 1-ленточные машины Тьюринга для следующих рекурсивных языков:

- $L = \{a^n b^n \mid n \in \mathbb{N}_0\}$;
- $L = \{w \in (\Sigma_{\text{bool}})^* \mid |w|_0 = |w|_1\}$;
- $L = \{a^n b^n c^n \mid n \in \mathbb{N}_0\}$;
- $L = \{www \mid w \in (\Sigma_{\text{bool}})^*\}$;
- $L = \{a^{n^2} \mid n \in \mathbb{N}_0\}$.

Итак, у нас есть две различные вычислительные модели – (собственно) машина Тьюринга и многоленточная машина Тьюринга. Чтобы одновременно использовать их в задачах теории вычислимости, мы должны доказать их эквивалентность – относительно классов допускаемых ими языков.

Пусть A и B – две машины (две машины Тьюринга, две многоленточные машины Тьюринга, и др.), которые работают над одним и тем же входным алфавитом Σ. Будем говорить, что A **эквивалентна** B, если для любого входа $x \in (\Sigma_{\text{bool}})^*$ выполняются следующие условия:

- A принимает x тогда и только тогда, когда B принимает x;
- A отклоняет x тогда и только тогда, когда B отклоняет x;
- вычисление машины A над x бесконечно тогда и только тогда, когда вычисление машины B над x также бесконечно.

Очевидно, что эквивалентность машин A и B влечёт равенство $L(A) = L(B)$, однако из равенства $L(A) = L(B)$ не следует, что A и B эквивалентны.

Лемма 4.11. *Для любой машины Тьюринга A существует некоторая* 1*-ленточная-*МТ B*, такая что A и B эквивалентны.*

Доказательство. Рассмотрим моделирование A машиной B – но при этом опустим формальное описание машины B.[15] Работа 1-ленточной-МТ B состоит из следующих двух этапов.

1. B полностью копирует вход w на рабочую ленту.
2. B моделирует работу A над словом w на своей рабочей ленте в пошаговом режиме. {Это означает, что B делает те же самые действия на своей рабочей ленте, что и A на своей входной ленте.}

Эквивалентность A и B очевидна. □

Упражнение 4.12. Приведите формальное описание 1-ленточной-МТ B, которая эквивалентна некоторой МТ $A = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$.

Далее мы обычно будем обходиться без формальных описаний машин Тьюринга, а также без формальных доказательств фактов типа $L(A) = L(B)$ для двух машин A и B – аналогично тем ситуациям, когда мы опускаем как детальные описания алгоритмов, так и доводы в пользу правильности конкретных программ. При этом мы экономим много времени, опуская формальные доказательства в тех ситуациях, когда многое интуитивно ясно и без них – например, что некоторая машина Тьюринга принимает данный язык, или что некоторая программа выполняет требуемую работу.

Лемма 4.13. *Для любой многоленточной машины Тьюринга A существует некоторая (собственно) машина Тьюринга B, такая что A и B эквивалентны.*

[15] Которое, как обычно, несложно – но требует большого количества рутинной работы.

Доказательство. Пусть A – некоторая k-ленточная-МТ. Покажем, как сконструировать машину Тьюринга B, которая в пошаговом режиме моделирует A. Удобнее сначала показать, как конфигурации моделируемой машины A представляются конфигурациями моделирующей машины B, а затем объяснить само моделирование шага машины A.

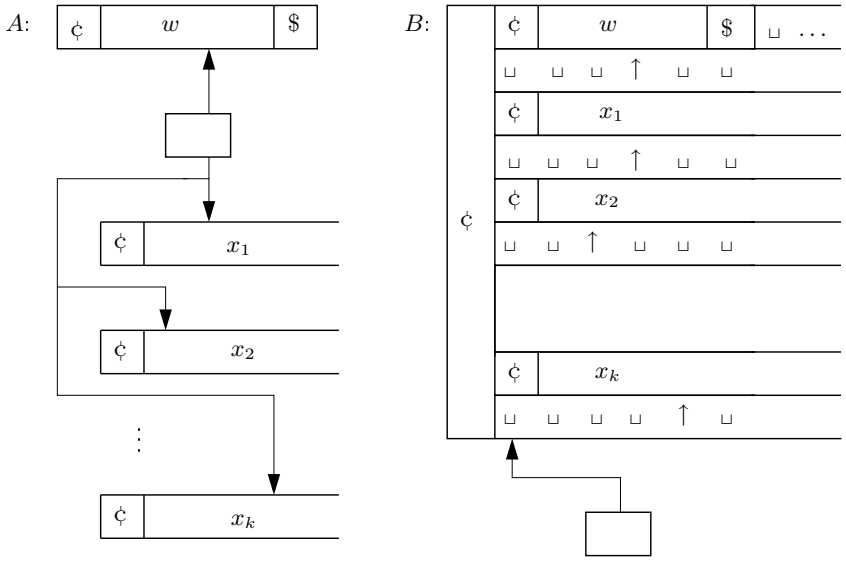


Рис. 4.11.

Идея представления некоторой конфигурации A конфигурацией B описана на рис. 4.11. B сохраняет на своей единственной ленте содержимое всех $k+1$ лент машины A. Интерпретация этих действий заключается в том, что B разделяет свою ленту на $2(k+1)$ дорожек – и использует эти дорожки для хранения содержимого лент машины A и позиций её головок. Технически это может быть осуществлено следующим образом.

Пусть Γ_A – рабочий алфавит машины A. Тогда в качестве рабочего алфавита машины B выбирается множество

$$\Gamma_B = (\Sigma \cup \{¢, \$, \sqcup\}) \times \{\sqcup, \uparrow\} \times (\Gamma_A \times \{\sqcup, \uparrow\})^k \cup \Sigma_A \cup \{¢\}.$$

Заметим, что последовательность

$$\alpha = (a_0, a_1, a_2, \ldots, a_{2k+1}) \in \Gamma_B$$

является *одним* символом нового алфавита, и мы будем говорить, что элемент a_i этой последовательности находится на i-й дорожке. Таким образом, i-е элементы всех символов ленты машины B определяют содержимое гипотетической i-й дорожки.

Конфигурация

$$(q, w, i, x_1, i_1, x_2, i_2, \ldots, x_k, i_k)$$

хранится машиной B следующим образом. Состояние q машины A сохраняется в конечной памяти[16] машины B. В 0-й дорожке ленты машины B находится $¢w\$$ – т. е. содержимое входной ленты машины A. Для каждого $i \in \{1, \ldots, k\}$ содержимым $(2i)$-й дорожки ленты B является слово $¢x_i$ – т. е. содержимое i-й рабочей ленты A. Позиция единственного символа $\uparrow$ на $(2i+1)$-й дорожке определяет позицию головки на i-й ленте машины A.

Шаг вычислений машины A может быть смоделирован с помощью следующей процедуры машины B.

1. B читает всё содержимое своей ленты слева направо. В течение этого просмотра ленты машина B сохраняет все $k+1$ символов,[17] прочитанных $k+1$ головкой машины A. (Эти символы являются в точности теми символами чётных дорожек, под которыми находятся символы $\uparrow$ нечётных.)
2. После первой фазы машина B знает все аргументы функции переходов машины A.[18] Поэтому B может за одно перемещение по своей ленте справа налево выполнить соответствующие действия: переместить символы $\uparrow$ согласно движениям головок машины A; заменить прочитанный символ; изменить внутреннее состояние.

Упражнение 4.14. Приведите формальное описание машины Тьюринга, которая выполняет первую фазу моделирования шага машины A в доказательстве леммы 4.13.

Определение 4.15. *Две модели машин (два класса машин) $\mathcal{A}$ и $\mathcal{B}$, решающих проблемы принадлежности, называются* **эквивалентными**, *если:*

- *для каждой машины $A \in \mathcal{A}$ существует машина $B \in \mathcal{B}$, эквивалентная A;*
- *и наоборот, для каждой машины $B \in \mathcal{B}$ существует машина $A \in \mathcal{A}$, эквивалентная B.*

Упражнение 4.16. Приведите формальное определение эквивалентности двух машин, которые вычисляют функции, действующие из Σ^* в Γ^*. После этого дайте определение эквивалентности классов машин для вычисление этих функций.

Непосредственно на основе лемм 4.11 и 4.13 мы получаем следующее утверждение.

Теорема 4.17. *Модели машины Тьюринга и многоленточной машины Тьюринга эквивалентны.* □

Знание того, что эти две вычислительные модели являются взаимозаменяемыми относительно алгоритмической разрешимости, облегчает нашу дальнейшую работу. Для доказательства принадлежности некоторого языка классу рекурсивных или рекурсивно перечислимых языков достаточно построить соответствующую многоленточную машину Тьюринга – что обычно легче построения (собственно) машины Тьюринга. Однако если нам нужно доказать, что некоторый язык не является рекурсивным (или рекурсивно перечислимым), то мы должны доказать *не*существование (собственно) машины Тьюринга для этого языка. Такая стратегия подобна возможности использования языка программирования высокого уровня для доказательства алгоритмической

[16] Т. е. состояние машины A сохраняется не на ленте, а как состояние машины B.

[17] Формально это означает, что множество состояний машины B включает множество $Q \times (\Sigma \cup \{¢, \$\}) \times \Gamma^k$ – и это возможно, поскольку последнее множество конечно.

[18] Напомним, что состояние машины A также сохраняется – как состояние машины B.

разрешимости некоторой проблемы – и необходимости использовать именно ассемблер (или даже машинный код) для доказательства алгоритмической *не*разрешимости проблемы. Эта тема будет подробнее рассмотрена в следующем разделе.

Поэтому представляет интерес доказательство эквивалентности машин Тьюринга и языков программирования высокого уровня. Формальное доказательство этой эквивалентности потребовало бы большого объёма детальной технической работы – что слишком трудоёмко. Поэтому мы только объясним идею доказательства такой эквивалентности.

Мы надеемся, что любой читатель, имеющий некоторый опыт в программировании, согласится с тем, что для любой машины Тьюринга M можно написать эквивалентную ей программу. Можно пойти и далее, написав интерпретатор $C_{\text{МТ}}$ для машин Тьюринга. Интерпретатор $C_{\text{МТ}}$ получает описание[19] некоторой машины M и некоторое слово w над входным алфавитом машины M в качестве входа – а затем моделирует работу M над словом w.

Как строится машина Тьюринга, эквивалентная некоторой заданной программе, написанной на языке программирования высокого уровня? Для ответа на этот вопрос посмотрим на развитие языков программирования. В самом начале все программы писались на ассемблере или даже в машинных кодах. Возможными командами были только операции сравнения двух целых чисел и арифметические операции. А все сложные команды языков программирования высокого уровня, необходимые для упрощения работы программиста, создавались как небольшие программы, состоящие из таких элементарных инструкций. Поэтому нет сомнения в том, что любой ассемблер и любой язык программирования эквивалентны. Напомним также, что компиляторы транслируют программы, написанные на языках программирования высокого уровня, в машинные коды.

Таким образом, достаточно показать эквивалентность между моделью машины Тьюринга и ассемблером. Можно сначала моделировать ассемблер с помощью так называемых регистровых машин, а затем в пошаговом режиме – регистровые машины с помощью машин Тьюринга. Но этот способ слишком трудоёмкий – и поэтому мы не будет его использовать. Мы упростим нашу задачу моделирования ассемблера машиной Тьюринга – аналогично тому, как операции умножения и деления могут быть выполнены с помощью программы, состоящей только из операций сложения, вычитания и целочисленного сравнения.

Упражнение 4.18. Напишите программу, которая для заданных целых переменных I и J вычисляет произведение $I \cdot J$. Программа может использовать только операции сложения, вычитания, а также целочисленного сравнения в структуре `if` ... `then` ... `else`.

Кроме того, мы можем опустить сравнение двух целых чисел – моделируя его программой, которая использует только операции $+1$ ($I := I + 1$), -1 ($I := I - 1$) и тест на 0 (`if` $I = 0$ `then` ... `else` ...).

Упражнение 4.19. Напишите программу, которая выполняет команду

`if` $I \geq J$ `then goto` 1 `else goto` 2

используя только операции $+1$ ($I := I + 1$), -1 ($I := I - 1$) и тест на 0.

[19] В некотором заранее зафиксированном формальном представлении.

Наконец, мы можем отказаться от сложения и вычитания.

Упражнение 4.20. Напишите программы, которые для двух заданных целых переменных I и J вычисляют разность $I - J$ и сумму $I + J$, используя только операции $+1$, -1 и тест на 0.

Задача моделирования многоленточной машиной Тьюринга программ, состоящих из команд

- $I := I + 1$
- $I := I - 1$
- `if` $I = 0$ `then` ... `else` ...

не требует больших усилий. Переменные такой программы могут храниться на рабочих лентах описываемой ММТ в формате $x\#y$, где x – двоичный код имении переменной I_x, а y – двоичное представление текущего значения I_x. Операции $+1$, -1 и тест $y = 0$ могут быть легко выполнены с помощью ММТ. Машина должна проделать больше работы только в том случае, когда содержимое рабочей ленты есть $¢x\#y\#\#z\#u\#\#\ldots$, а память (число ячеек) для значения переменной I_x является слишком маленькой для хранения текущего значения y. Если это происходит,[20] то ММТ должна переместить суффикс $\#\#z\#u\#\#\ldots$ содержимого ленты на одну ячейку направо – чтобы получить ещё один бит для хранения y.

В теоретической информатике были разработаны сотни формальных моделей (причём не только моделей машин) – для определения понятия алгоритмической разрешимости. Но все разумные модели эквивалентны модели машины Тьюринга – поэтому большой опыт разработки таких моделей привёл к формулировке известного тезиса Чёрча–Тьюринга.

Тезис Чёрча–Тьюринга

> *Машины Тьюринга суть формализация понятия «алгоритм», т. е. класс рекурсивных языков (разрешимых проблем принадлежности) соответствуют классу алгоритмически (автоматически) распознаваемых языков.*

Тезис Чёрча–Тьюринга недоказуем – потому что он описывает формализацию интуитивного понятия «алгоритм», и, следовательно является новой аксиомой – которая вместе с аксиомами математики образует основу теоретической информатики. Никакая аксиома не может быть доказана – потому что формально устанавливает интерпретацию некоторого фундаментального понятия, которая сделана исключительно на основе нашего опыта и веры. Таким образом, применяя эту формализацию понятия «алгоритм», невозможно доказать несуществование другого формального определения этого термина – такого, что оно:

- согласуется с нашим интуитивным понятием термина «алгоритм»,
- позволяет нам алгоритмически решить проблемы принадлежности, которые не являются разрешимыми с помощью машин Тьюринга.

[20] А именно – после добавления 1 к I_x.

Единственное, что может произойти в будущем, – это создание «более сильной» модели алгоритмов; в этом случае нужно будет пересмотреть основные принципы теоретической информатики. Однако поиск такой более мощной модели пока безуспешен – и мы даже знаем, что машины Тьюринга эквивалентны физической модели квантовых компьютеров.[21] Поэтому и получила широкое распространение уверенность учёных в том, что не существует модели алгоритмов, более мощной, чем машины Тьюринга.

Итак, текущее состояние теоретической информатики в чём-то похоже на ситуацию, сложившуюся в математике и физике. Мы принимаем тезис Чёрча–Тьюринга – потому что он согласуется с нашим опытом; мы постулируем его как аксиому: как уже было упомянуто, этот тезис имеет те же особенности, что и аксиомы математики. Этот тезис – единственная собственная аксиома теоретической информатики, все остальные её аксиомы «происходят из математики». Он не может быть доказан, но может быть опровергнут – иными словами, не может быть исключена потенциальная возможность его опровержения.[22]

4.4 Недетерминированные машины Тьюринга

Недетерминизм может быть введён для машин Тьюринга тем же способом, который мы использовали для конечных автоматов. А именно, для каждого набора аргументов существует возможность выбора из конечного числа вариантов. На формальном уровне это означает, что функция переходов δ теперь описывается не в виде

$$Q \times \Gamma \ \rightarrow \ Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}, \mathrm{N}\},$$

а в виде

$$Q \times \Gamma \ \rightarrow \ \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}, \mathrm{N}\}).$$

Другая возможность – рассмотрение δ как отношения на множестве

$$(Q \times \Gamma) \times (Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}, \mathrm{N}\}).$$

Недетерминированная машина Тьюринга M принимает входное слово w тогда и только тогда, когда существует некоторое допускающее вычисление машины M над w. Это определение принятия соответствует «оптимистическому взгляду» – а именно тому, что M всегда делает правильный выбор. Формальное определение недетерминированной машины Тьюринга следующее.

Определение 4.21. Недетерминированная машина Тьюринга (НМТ) – *это семёрка* $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{accept}}, q_{\mathrm{reject}})$*, где:*

- $Q, \Sigma, \Gamma, q_0, q_{\mathrm{accept}}, q_{\mathrm{reject}}$ *имеют тот же смысл, что и в случае (детерминированной)* МТ*,*

[21] Работающих на основе принципов квантовой механики.

[22] Однако важно отметить, что возможное опровержение некоторой аксиомы нельзя считать катастрофой; подобные результаты – неизбежное следствие процесса развития науки. А теория, основанная на опровергнутой аксиоме, не должна быть забыта. Её результаты должны быть только несколько подправлены – потому что они являются истинными в том случае, когда сама аксиома считается истинной, и существует множество структур, где эта аксиома верна. Более того, пересмотр теории при использовании некоторой новой аксиомы обычно относится к наиболее важным шагам в развитии науки.

- $\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\text{L}, \text{R}, \text{N}\})$ – **функция переходов** *машины* M, *удовлетворяющая условию*

$$\delta(p, ¢) \subseteq \{(q, ¢, X) \mid q \in Q, X \in \{\text{R}, \text{N}\}\}$$

для всех $p \in Q - \{q_{\text{accept}}, q_{\text{reject}}\}$.
{*Левый граничный маркер не может быть заменён (перезаписан), а головка не может перемещаться налево, если она установлена на символ* ¢.}

Конфигурация *машины* M – *элемент множества*

$$\mathbf{conf}(\boldsymbol{M}) = (\{¢\} \cdot \Gamma^* \cdot Q \cdot \Gamma^*) \cup (Q \cdot \{¢\} \cdot \Gamma^*).$$

{*Смысл конфигурации тот же самый, что и для (детерминированной)* МТ.}
Для любого входного слова $w \in \Sigma^*$ *конфигурация* $q_0 ¢ w$ *называется* **инициальной конфигурацией машины** $\boldsymbol{M}$ **над** $\boldsymbol{w}$. *Конфигурация называется* **допускающей**, *если её внутренним состоянием является* q_{accept}. *Конфигурация называется* **отклоняющей**, *если её внутренним состоянием является* q_{reject}.
Шаг (вычисления) *машины* M – *это отношение* $\vdash_M$, *определённое над множеством конфигураций* (*т. е.* ($\vdash_M \subseteq \text{conf}(M) \times \text{conf}(M)$) *следующим образом. Для всех* $p, q \in Q$ *и* $x_1, x_2, \ldots, x_n, y \in \Gamma$:

- $x_1 x_2 \ldots x_{i-1} q x_i x_{i+1} \ldots x_n \vdash_M x_1 x_2 \ldots x_{i-1} p y x_{i+1} \ldots x_n$,
  если $(p, y, \text{N}) \in \delta(q, x_i)$,
- $x_1 x_2 \ldots x_{i-2}, x_{i-1} q x_i x_{i+1} \ldots x_n \vdash_M x_1 x_2 \ldots x_{i-2} p x_{i-1} y x_{i+1} \ldots x_n$,
  если $(p, y, \text{L}) \in \delta(q, x_i)$,
- $x_1 x_2 \ldots x_{i-1} q x_i x_{i+1} \ldots x_n \vdash_M x_1 x_2 \ldots x_{i-1} y p x_{i+1} \ldots x_n$,
  если $(p, y, \text{R}) \in \delta(q, x_i)$ *для* $i < n$,
- $x_1 x_2 \ldots x_{n-1} q x_n \vdash_M x_1 x_2 \ldots x_{n-1} y p_{\sqcup}$,
  если $(p, y, \text{R}) \in \delta(q, x_n)$.

Отношение $\vdash_M^*$ – *рефлексивно-транзитивное замыкание отношения* $\vdash_M$.
Вычисление машины $\boldsymbol{M}$ – *последовательность конфигураций* $C_0, C_1, \ldots$, *такая что для* $i = 0, 1, 2, \ldots$

$$C_i \vdash_M C_{i+1}$$

{*Иногда для некоторого вычисления мы будем использовать обозначение* $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \ldots$ *вместо более короткого* $C_0, C_1, C_2 \ldots$}
Вычисление машины $\boldsymbol{M}$ **над входом** $\boldsymbol{x}$ – *это любое вычисление, которое начинается инициальной конфигурацией* $q_0 ¢ x$, *и либо бесконечно, либо заканчивается в конфигурации* $w_1 q w_2$, *где* $q \in \{q_{\text{accept}}, q_{\text{reject}}\}$. *Вычисление машины* M *над* x *называется* **допускающим**, *если оно заканчивается в некоторой допускающей конфигурации. Вычисление машины* M *над* x *называется* **отклоняющим**, *если оно заканчивается в некоторой отклоняющей конфигурации.*
Будем говорить, что M **допускает** *входное слово* w, *если существует допускающее вычисление машины* M *над* w. *Иначе будем говорить, что* M **не допускает** w (*т. е. если все вычисления машины* M *над* w *отклоняющие или бесконечные).*
Язык $\boldsymbol{L(M)}$**, допускаемый недетерминированной машиной Тьюринга** $\boldsymbol{M}$, *определяется следующим образом:*

$$\begin{aligned} \boldsymbol{L(M)} &= \{w \in \Sigma^* \mid q_0 ¢ w \vdash_M^* y q_{\text{accept}} z \text{ для некоторых } y, z \in \Gamma^*\} \\ &= \{w \in \Sigma^* \mid \ M \text{ допускает } w\ \}. \end{aligned}$$

Упражнение 4.22. Опишите неформально, а затем формально недетерминированную k-ленточную машину Тьюринга.

Упражнение 4.23. Пусть M – недетерминированная многоленточная машина Тьюринга. Опишите НМТ M', такую что $L(M) = L(M')$.

Аналогично случаю конечных автоматов, недетерминизм может упрощать вычислительную стратегию машины Тьюринга. Для примера рассмотрим язык

$$L_{\text{unequal}} = \{x\#y \mid x, y \in (\Sigma_{\text{bool}})^*, x \neq \lambda, x \neq y\}.$$

Детерминированная МТ должна сравнить x и y посимвольно – для определения возможной разницы между x и y. Пусть $x = x_1x_2 \ldots x_n$ и $y = y_1y_2 \ldots y_m$ для слов x_j, $y_l \in \Sigma$, $j = 1, \ldots n$, $l = 1, 2, \ldots m$. Если x и y различны, то НМТ может угадать позицию i, в которой x и y различаются – а затем проверить эту догадку путём сравнения x_i с y_i. Опишем недетерминированную 1-ленточную-МТ A, которая принимает язык L_{unequal}, следующим образом. Формальное представление машины A приведено на рис. 4.12. Для любого входного слова w работа машины A состоит из следующих четырёх этапов (рис. 4.12).

1. A детерминированно проверяет, равно ли число вхождений символа # в слово w в точности 1. Эта проверка может быть выполнена путём простого последовательного просмотра входной ленты; при этом мы используем состояния $q_0, q_1, q_{\text{reject}}$. Если неверно, что w содержит в точности один символ #, то A отклоняет вход w. Если же $w = x\#y$ для некоторых $x, y \in (\Sigma_{\text{bool}})^*$, то A продолжает вычисления согласно этапу 2, переходя при этом в состояние q_2.
2. A устанавливает обе головки на левые граничные маркеры ¢ входной и рабочей лент; при этомA продолжает находиться в состоянии q_2.
3. A последовательно перемещает обе головки направо и заменяет символы $\sqcup$ на рабочей ленте символами $a \in \Gamma - \{0, 1, \#\}$. На каждом таком шаге A недетерминированно угадывает, является ли текущая позиция слова x той позицией, в которой различаются слова x и y; это делается в состоянии q_{guess}. Если при чтении символа $b \in \{0, 1\}$ на своей входной ленте машина A предполагает, что именно в данной позиции соответствующие символы слов x и y различаются, – то переходя на этап 4, A продолжает вычисления для проверки сделанного недетерминированного предположения.[23] Если A читает на входной ленте символ #, то она также переходит на этап 4 – в данном случае для проверки условия $|x| \neq |y|$.
4. В текущей ситуации предполагается, что расстояние между символом ¢ и позицией головки на рабочей ленте равно позиции i ($i \in \mathbb{N}$) символа $b \in \{0, 1, \#\}$ слова $x\#$. Машина A перемещает свою считывающую головку направо, до тех пор, пока не достигнет символа # – без перемещения головки рабочей ленты; для этого используются состояния p_0 и p_1. После этого A одновременно перемещает головку входной ленты направо и головку рабочей ленты налево – используя для этого состояния s_0 и s_1. Когда головка рабочей ленты достигает ¢, головка входной ленты оказывается установленной на предполагаемой позиции i слова y.

   A принимает вход $w = x\#y$ в одном из следующих случаев:
   - если сохранённый символ b (определяемый согласно состоянию s_b) отличается от читаемого в данный момент символа слова y входной ленты;

[23] Для этого A переходит в одно из состояний p_0 или p_1 – в зависимости от символа b.

- если $|x| < |y|$ (т. е. если $(q_{\text{accept}}, \text{N}, ¢, \text{N}) \in \delta(p_{\#}, c, \text{N})$ для всех $c \in \{0,1\}$);
- если $|x| > |y|$ (т. е. $\delta(s_b, \$, d) = \{(q_{\text{accept}}, \text{N}, d, \text{N})\}$ для всех $b \in \{0,1\}$ и $d \in \{a, ¢\}$).

Иначе A не принимает w.

Описанная стратегия недетерминированного угадывания, предваряющая последующую детерминированную проверку, является типичной для недетерминированных вычислений.[24]

Другой пример – недетерминированная 2-ленточная-МТ B, допускающая язык

$$L_{\text{quad}} = \{a^{n^2} \mid n \in \mathbb{N}_0\}.$$

Для заданного входа w машина B сначала угадывает натуральное n путём установки головки первой рабочей ленты на n-ю позицию. Потом она проверяет, выполняется ли равенство $|w| = n^2$.

Упражнение 4.24. Приведите детальное описание работы недетерминированной 2-ленточной-МТ B, которая допускает L_{quad}, а также опишите B графически.

Следующий практический вопрос – может ли некоторая недетерминированная машины Тьюринга допускать язык, который не допускается ни одной (детерминированной) машиной Тьюринга? Аналогично случаю конечных автоматов, ответ является отрицательным – а стратегия моделирования недетерминированных вычислений базируется на поиске в ширину в дереве вычислений недетерминированной машина Тьюринга.

Определение 4.25. *Пусть $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ – некоторая* НМТ, *а x – некоторое слово над входным алфавитом Σ машины M.* **Дерево вычислений $T_{M,x}$ машины M над x** *– это (возможно, бесконечное) ориентированное дерево, определяемое следующим образом.*

- *Каждая вершина дерева $T_{M,x}$ помечена некоторой конфигурацией.*
- *Корень дерева $T_{M,x}$ (т. е. единственная его вершина уровня 0) помечен инициальной конфигурацией $q_0¢x$ машины M над словом x.*
- *Если некоторая вершина дерева $T_{M,x}$ помечена конфигурацией C, то число дочерних вершин (непосредственных потомков) этой вершины в точности равно числу конфигураций-последователей данной C – причём сами непосредственные потомки помечены этими конфигурациями-последователями.*

Очевидно, что такое определение дерева вычислений может быть использовано и для недетерминированной многоленточной машины Тьюринга.

Упражнение 4.26. Нарисуйте деревья вычислений недетерминированной 1-ленточной-МТ, приведённой на рис. 4.12 – над входами 01#01#1 и 01#0.

Существуют два принципиальных различия между деревьями вычислений НКА и НМТ.

- Во-первых, деревья вычислений недетерминированных конечных автоматов всегда конечны – а деревья вычислений недетерминированных машин Тьюринга могут быть бесконечными.

[24] В главе 6 мы будем рассмотрим вопрос, насколько строго можно этим способом охарактеризовать недетерминизм.

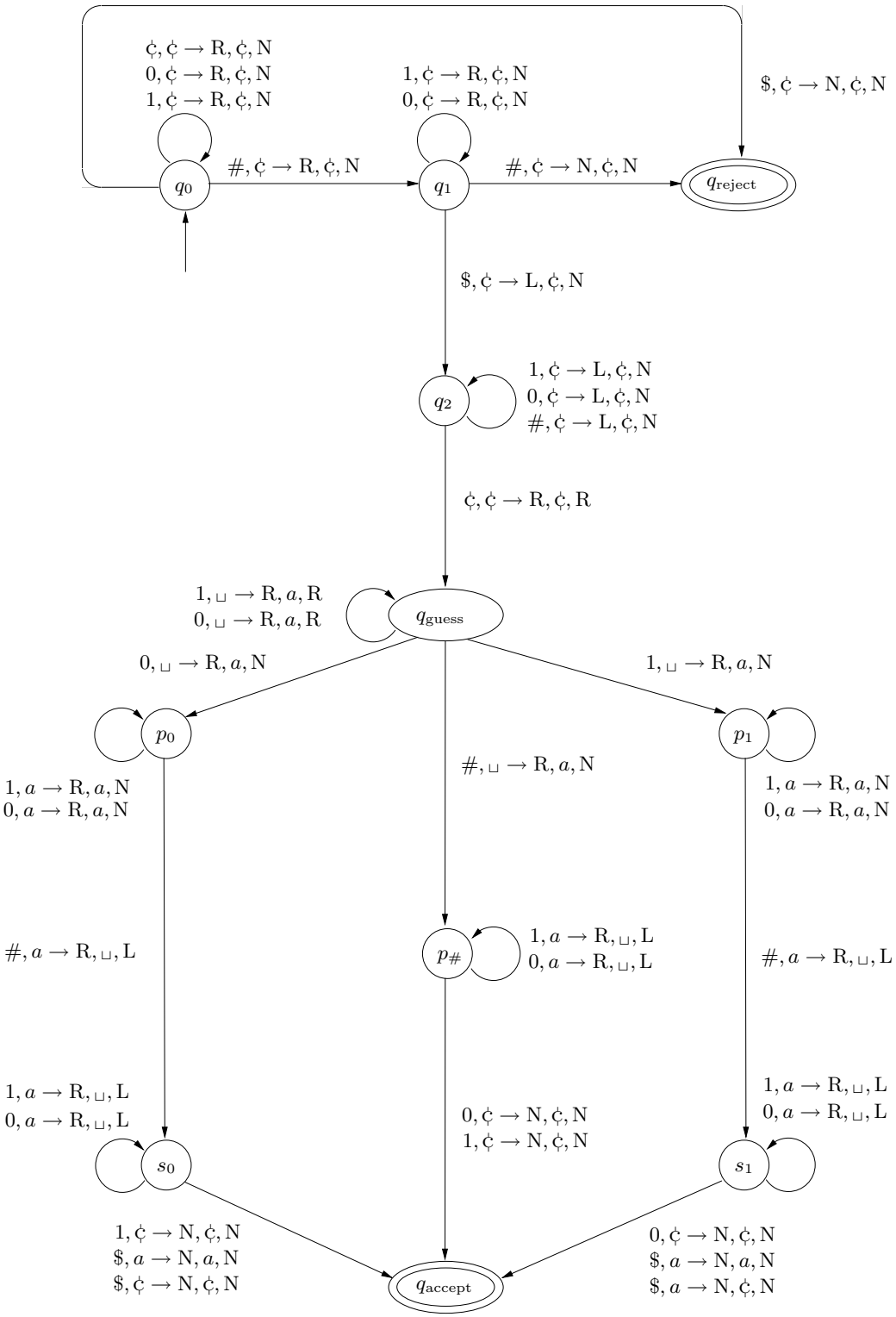

ϛ, ϛ → R, ϛ, N
0, ϛ → R, ϛ, N
1, ϛ → R, ϛ, N
1, ϛ → R, ϛ, N
0, ϛ → R, ϛ, N
$, ϛ → N, ϛ, N
#, ϛ → R, ϛ, N
#, ϛ → N, ϛ, N
$, ϛ → L, ϛ, N
1, ϛ → L, ϛ, N
0, ϛ → L, ϛ, N
#, ϛ → L, ϛ, N
ϛ, ϛ → R, ϛ, R
1, ␣ → R, a, R
0, ␣ → R, a, R
0, ␣ → R, a, N
1, ␣ → R, a, N
#, ␣ → R, a, N
1, a → R, a, N
0, a → R, a, N
1, a → R, a, N
0, a → R, a, N
#, a → R, ␣, L
1, a → R, ␣, L
0, a → R, ␣, L
#, a → R, ␣, L
1, a → R, ␣, L
0, a → R, ␣, L
0, ϛ → N, ϛ, N
1, ϛ → N, ϛ, N
1, ϛ → N, ϛ, N
$, a → N, a, N
$, ϛ → N, ϛ, N
0, ϛ → N, ϛ, N
$, a → N, a, N
$, a → N, ϛ, N
q0
q1
qreject
qguess
p0
p1
p#
s0
s1
qaccept

Рис. 4.12.

- Во-вторых, в случае НКА у всех конфигураций дерева вычислений, находящихся на одной и той же глубине,[25] головка установлена на одну и ту же позицию ленты. Это свойство отсутствует в случае НМТ.

Теорема 4.27. *Пусть M – некоторая* НМТ. *Тогда существует некоторая* МТ A, *такая что*

(a) $L(M) = L(A)$,
(b) если у машины M нет бесконечных вычислений над словами языка $(L(M))^{\complement}$, *то A всегда останавливается.*

Доказательство. Согласно лемме 4.13, нам достаточно привести описание 2-ленточной МТ A, обладающей свойствами (a) и (b). Мы лишь объясним работу машины A – без её строгого формального построения. Стратегией работы машины A является поиск в ширину в деревьях вычислений машины M.

Вход: Слово w.
Этап 1. A копирует инициальную конфигурацию $q_0¢w$ машины M, работающей над входом над w, на свою первую рабочую ленту.
Этап 2. A проверяет, содержит ли первая лента некоторую допускающую конфигурацию[26] машины M. Если содержит, то A принимает w. Иначе A переходит на этап 3 и продолжает вычисления.
Этап 3. Для каждой конфигурации, записанной на первой рабочей ленте, A последовательно записывает все конфигурации, являющиеся её непосредственными потомками, на свою вторую рабочую ленту. Заметим, что у любой конфигурации имеется конечное число потомков (т. к. для любого набора аргументов множество возможных действий машины M конечно) – поэтому A может выполнить этот процесс за конечное время. Если ни у какой конфигурации, записанной на первой ленте, нет ни одного потомка (т. е. вторая рабочая лента остаётся пустой) – то A останавливается в состоянии q_{reject}. Иначе A продолжает вычисления, переходя на этап 4.
Этап 4. A стирает содержимое первой рабочей ленты и копирует содержимое второй рабочей ленты на первую ленту. После этого A стирает содержимое второй рабочей ленты и продолжает вычисления, переходя на этап 2.

После i-го выполнения этапов 3 и 4 первая рабочая лента машины A содержит все такие конфигурации дерева вычислений $T_{M,w}$, у которых расстояние от корня дерева равно i, – т. е. все конфигурации машины M, которые достижимы за i шагов вычисления.

Если $w \in L(M)$, то существует некоторое допускающее вычисление C машины M над w. А поскольку любое допускающее вычисление конечно, мы можем считать, что для некоторого $j \in \mathbb{N}$ вычисление C состоит из j шагов. Поэтому после того, как этапы 3 и 4 выполнятся ровно j раз, машина A найдёт некоторую допускающую конфигурация на шаге 2 – и примет слово w.

Если же $w \notin L(M)$, то очевидно, что A не принимает w. Более того, поскольку дерево $T_{M,x}$ конечное, A достигает состояния q_{reject}. □

[25] Т. е. на одном и том же расстоянии от корня.
[26] Для этого достаточно определить, записано ли допускающее состояние q_{accept} в некоторой ячейке первой рабочей ленты.

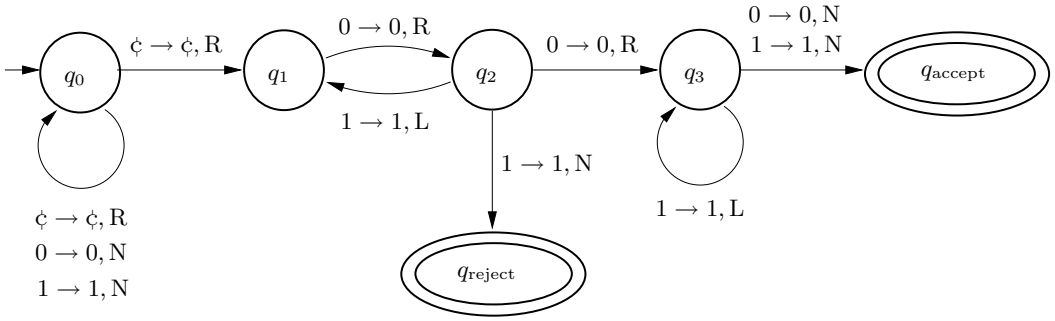


Рис. 4.13.

Упражнение 4.28. Пусть A – НМТ, приведённая на рис. 4.13.

- Опишите первые 6 уровней дерева вычислений $T_A(x)$ для входных слов $x = 01$ и $x = 0010$ – т. е. приведите все конфигурации, включающие по крайней мере 5 шагов вычислений машины M.
- Опишите язык $L(A)$.

4.5 Кодирование машин Тьюринга

У каждой программы имеется двоичное представление, определяемое её машинным кодом. Компьютером выполняется работа, заключающаяся в трансляции программы из слова над алфавитом Σ_{keyboard} в её машинный код – слово над алфавитом Σ_{bool}. Данная глава преследует похожую цель для машин Тьюринга: мы приведём возможный вариант их простого двоичного кодирования (двоичного представления). Начнём с описания кодов машин Тьюринга над алфавитом $\{0, 1, \#\}$.

Пусть $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ – некоторая МТ, где

$$Q = \{q_0, q_1, \ldots, q_m, q_{\text{accept}}, q_{\text{reject}}\} \quad \text{и} \quad \Gamma = \{A_1, A_2, \ldots, A_r\}.$$

Во-первых, определим коды специальных символов следующим образом:

$$\begin{aligned}
\text{Code}(q_i) &= 10^{i+1}1 \text{ для } i = 0, 1, \ldots, m,\\
\text{Code}(q_{\text{accept}}) &= 10^{m+2}1,\\
\text{Code}(q_{\text{reject}}) &= 10^{m+3}1,\\
\text{Code}(A_j) &= 110^j11 \text{ для } j = 1, \ldots, r,\\
\text{Code}(\text{N}) &= 1110111,\\
\text{Code}(\text{R}) &= 1110^2111,\\
\text{Code}(\text{L}) &= 1110^3111.
\end{aligned}$$

Коды этих символов используются для кодирования каждого конкретного перехода:

$$\begin{aligned}
&\text{Code}(\delta(p, A_l) = (q, A_m, \alpha))\\
&= \#\text{Code}(p)\text{Code}(A_l)\text{Code}(q)\text{Code}(A_m)\text{Code}(\alpha)\#
\end{aligned}$$

для любого перехода $\delta(p, A_l) = (q, A_m, \alpha)$, где $p \in \{q_0, q_1, \ldots, q_m\}$, $q \in Q$, $l, m \in \{1, \ldots, r\}$, $\alpha \in \{\mathrm{N}, \mathrm{L}, \mathrm{R}\}$.

Код машины Тьюринга M начинается с общей информации – а именно, с числа состояний ($|Q|$) и числа символов рабочего алфавита ($|\Gamma|$) машины M – после чего следует список всех переходов. Таким образом,

$$\mathrm{Code}(M) = \#0^{m+3}\#0^r\#\#\mathrm{Code}(\mathit{Transition}_1)\#\mathrm{Code}(\mathit{Transition}_2)\#\ldots.$$

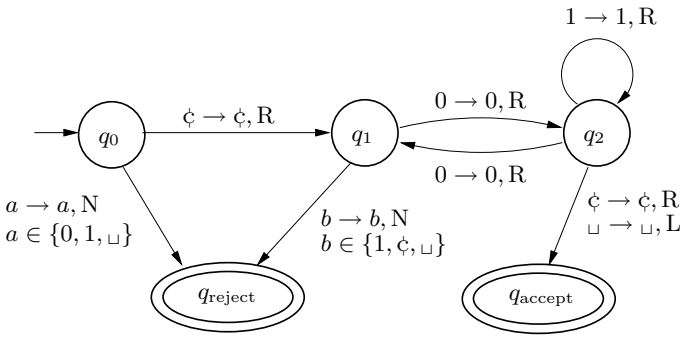


Рис. 4.14.

Упражнение 4.29. Пусть M – машина Тьюринга, заданная на рис. 4.14.

- Приведите код $\mathrm{Code}(M)$ машины M. Сделайте это путём описания каждой части $\mathrm{Code}(M)$ между двумя символами #.
- Определите язык $L(M)$.
- Является ли язык $L(M)$ регулярным? (Обоснуйте свой ответ.)

Чтобы получить код над алфавитом Σ_{bool}, можно использовать гомоморфизм

$$h : \{0, 1, \#\}^* \to (\Sigma_{\mathrm{bool}})^*,$$

в котором

$$h(\#) = 01, \quad h(0) = 00, \quad h(1) = 11.$$

Определение 4.30. *Для каждой* МТ M

$$\mathbf{Kod}(\boldsymbol{M}) = h(\mathrm{Code}(M))$$

есть **код машины Тьюринга** $\boldsymbol{M}$.

$$\mathbf{KodMT} = \{\mathrm{Kod}(M) \mid M \text{ – } \textit{некоторая}\ \mathrm{MT}\}$$

обозначает множество кодов всех машин Тьюринга.

Очевидно, что отображение, ставящее в соответствие слово $\mathrm{Kod}(M)$ произвольной машине Тьюринга M, является инъективным – поэтому $\mathrm{Kod}(M)$ однозначно определяет M.

Упражнение 4.31. Разработайте программу, которая вычисляет слово $\mathrm{Code}(M)$ для любого формального описания машины Тьюринга M – согласно определению 4.30.

Упражнение 4.32. Разработайте программу, отвечающую для любого $x \in \{0,1\}^*$ на вопрос, верно ли, что $x \in \mathrm{KodMT}$ – т. е. является ли x кодом некоторой машины Тьюринга.

Далее запись $\boldsymbol{A_{\mathrm{ver}}}$ обозначает алгоритм (машину Тьюринга), решающий проблему принадлежности $(\Sigma_{\mathrm{bool}}, \mathrm{KodMT})$ – т. е. для любого заданного слова $x \in (\Sigma_{\mathrm{bool}})^*$ определяющий, является ли это слово x кодом некоторой МТ.

Очень важно отметить, что путём фиксации конкретного варианта двоичного представления (двоичных кодов) машин Тьюринга мы получаем на всём таком множестве некоторый линейный порядок.

Определение 4.33. *Пусть $x \in (\Sigma_{\mathrm{bool}})^*$. Для каждого натурального i будем говорить, что x является* **кодом i-й МТ**, *если:*

- $x = \mathrm{Kod}(M)$ *для некоторой машины Тьюринга M,*
- *множество $\{y \in (\Sigma_{\mathrm{bool}})^* \mid y$ предшествует x относительно канонического порядка$\}$ содержит в точности $i-1$ слов, являющихся кодами машин Тьюринга.*

Если $x = \mathrm{Kod}(M)$ – код i-й МТ, *то будем говорить, что M является* **i-й машиной Тьюринга M_i**. *Число i при этом называется* **порядком МТ M_i**.

Заметим, что для любого заданного натурального i не представляет сложности вычислить код i-й машины Тьюринга $\mathrm{Kod}(M_i)$. Итак, пусть Gen – функция, действующая из $\mathbb{N}$ в $(\Sigma_{\mathrm{bool}})^*$, определённая равенством $Gen(i) = \mathrm{Kod}(M_i)$.

Лемма 4.34. *Функция Gen является рекурсивной – т. е. существует некоторый алгоритм (некоторая машина Тьюринга), который вычисляет $\mathrm{Kod}(M_i)$ для любого заданного натурального i.*

Доказательство. Программа, вычисляющая Gen, может быть следующей.

Вход: $i \in \mathbb{N}$.
Этап 1.
  $x := 1$; {x является словом над алфавитом $(\Sigma_{\mathrm{bool}})^*$}
  $I := 0$;
Этап 2.
  `while` $I < i$ `do begin`
    выполнить A_{ver} для проверки условия $x \in \mathrm{KodMT}$;
    `if` $x \in \mathrm{KodMT}$ `then begin`
      $I := I + 1$;
      $y := x$
    `end`;
    $x :=$ следующее после x слово над алфавитом $(\Sigma_{\mathrm{bool}})^*$
        относительно канонического порядка
  `end`
Этап 3.
  `output`(y).

Упражнение 4.35. Напишите программу, которая для кода $\mathrm{Kod}(M) \in (\Sigma_{\mathrm{bool}})^*$ некоторой машины Тьюринга M вычисляет порядок M.

4.6 Заключение

Машина Тьюринга является абстрактной моделью вычислений; при этом её вычислительная мощность равна вычислительной мощности реальных компьютеров. Компоненты машины Тьюринга – бесконечная лента, конечное устройство управления и читающе-пишущая головка. Лента состоит из ячеек, каждая из которых содержит некоторый символ рабочего алфавита. Таким образом, ячейка соответствует регистру реального компьютера, а символы рабочего алфавита соответствуют всем возможным компьютерным словам (возможному содержанию регистра). Лента рассматривается не только как вход, но и как память. Инструкции машины (элементарные действия, команды) называются переходами. Аргументами перехода являются текущее состояние конечного устройства управления и символ, читаемый головкой ленты. В процессе выполнения инструкции машина Тьюринга может изменить своё состояние, заменить читаемый символ некоторым другим, а также переместить свою головку на одну ячейку налево или направо. Вычисление определяется как последовательность таких элементарных инструкций. Машина принимает [отклоняет] слово x, если она заканчивает вычисление над x в специальном состоянии q_{accept} [q_{reject}] – и не принимает слово x, если либо она отклоняет x, либо вычисление над x бесконечно. Для некоторой заданной машины M язык $L(M)$ есть множество всех слов, принимаемых M. Язык называется рекурсивно перечислимым, если $L = L(M)$ для некоторой машины Тьюринга M. Язык L называется рекурсивным, если $L = L(M)$ для некоторой МТ M, не имеющей бесконечных вычислений – т. е. если все вычисления машины M заканчиваются в одном из состояний q_{accept} или q_{reject}.

В модели многоленточной машины Тьюринга вместо одной бесконечной ленты (используемой как для входа, так и в качестве памяти) применяются:

- конечная лента для представления входа;[27]
- конечное число бесконечных рабочих лент в качестве памяти.

Вычислительные модели машин Тьюринга и многоленточных машин Тьюринга эквивалентны между собой – в том смысле, что каждая МТ может быть смоделирована с помощью ММТ, и наоборот. Эти модели машины Тьюринга эквивалентны программам на любом обычном языке программирования.

Тезис Чёрча–Тьюринга говорит, что машина Тьюринга без бесконечных вычислений (т. е. машина Тьюринга, которая всегда останавливается) является формализацией интуитивного понятия «алгоритм». Следовательно, все проблемы, разрешимые с помощью машин Тьюринга, являются алгоритмически (автоматически) разрешимыми – и наоборот, все проблемы, *не*разрешимые с помощью машин Тьюринга, являются алгоритмически неразрешимыми. Тезис Чёрча–Тьюринга – единственная собственная аксиома информатики, и, как и любая другая аксиома, никогда не может быть доказана. Единственная открытая возможность состоит в том, чтобы *пересмотреть* её – но это будет необходимо только в том случае, если кто-нибудь опишет более сильную (более мощную), однако при этом реалистическую модель алгоритмов.

Недетерминизм может быть введён для машин Тьюринга так же, как это было сделано в предыдущей главе для конечных автоматов. Для одного и того же входного слова недетерминированная машина Тьюринга может иметь несколько различных вариантов вычислений.[28] Вход x считается принятым, если существует по крайней мере

[27] Аналогично конечному автомату.

[28] Возможно даже бесконечное множество вычислений.

одно вычисление над x, заканчивающееся в состоянии q_{accept}. Любая недетерминированная машина Тьюринга может быть смоделирована (детерминированной) машиной Тьюринга. Аналогично случаю конечных автоматов, моделирование основано на поиске в ширину в деревьях вычислений НМТ.

Машины Тьюринга могут быть однозначно закодированы как слова над алфавитом $\{0,1\}$ – подобно тому, как каждая программа имеет свой двоичный машинный код. Слова над алфавитом $\{0,1\}$ можно считать линейно упорядоченными (на основе канонического порядка) – поэтому мы можем, используя этот порядок, получить линейный порядок и для всех машин Тьюринга. Для любого заданного натурального i можно вычислить код i-й МТ – и наоборот, для любой заданной машины Тьюринга можно вычислить её номер.

Введение формальной модели алгоритмов было первым шагом, который привёл к основанию теоретической информатики. Этот процесс был инициирован оригинальной работой Гёделя [20]. Упомянутая статья представляет самое первое доказательство существования математических проблем, которые не могут быть решены алгоритмически – т. е. никаким методом.[29] Именно результат Гёделя послужил мотивацией для разработки Чёрчем [12], Клини [38], Постом [51] и Тьюрингом [68] формальных моделей интуитивного понятия «алгоритм». Все эти модели – а также многие другие – оказались, как обнаружилось впоследствии, эквивалентными между собой. Итак, результат этого опыта разработки формальной модели алгоритмов – тезис Чёрча–Тьюринга.

Модель машины Тьюринга [68] стала основной моделью алгоритмов (компьютеров) в теоретической информатике – хотя оригинальное толкование этой модели с компьютерами связано не было, цель Тьюринга состояла в формализации методов (алгоритмов), необходимых для *символьной манипуляции*.[30] Вместо того, чтобы думать о компьютере, он воображал человека (человека-вычислителя, в том числе математика), выполняющего некоторое вычисление с помощью карандаша и бумаги. Выбор одномерной (линейной) ленты машины Тьюринга был мотивирован тем, что на бумаге мы тоже пишем последовательно, строка за строкой. Конечное число используемых символов определяет рабочий алфавит. Для систематизации всех этих идей Тьюринг и разделил ленту на ячейки – так, что каждая из них может содержать ровно один символ. При этом размер ленты (т. е. листа) считается неограниченным. Тьюринг предполагал, что поскольку человеческий мозг конечен, он может находиться только в одном из конечного числа состояний: конечное множество состояний машины и является следствием этого предположения. Тьюринг использовал аналогичные аргументы для обоснования ещё одного предположения – о том, что одно действие человека-вычислителя (математика) может повлиять только на часть ленты, размер которой ограничен некоторой константой – и эта константа не должна зависеть от длины всего содержимого ленты (записанного на ней слова). Так как любое подобное действие может быть представлено в виде последовательного выполнения элементарных инструкций, каждая из которых действует только на один символ, Тьюринг решил в качестве базовых инструкций использовать переходы – в том виде, в котором они были определены в разделе 4.2.

[29] Сегодня мы сказали бы «никаким алгоритмом».

[30] Термин «символьные вычисления» широко применяется и в современной информатике. *(Прим. перев.)*

Многоленточные машины Тьюринга, ставшие впоследствии основной вычислительной моделью в теории сложности, были введены Хартманисом и Стирнсом в [25]. Очень интересное обсуждение темы этой главы приведено Харелом в [23].

Тысячи талантов рассказывают о том,
чем обладает эпоха –
но лишь гений делает то,
чего ей не хватает.
Э. Гейбель

5 Теория вычислимости

5.1 Цели и задачи главы

Теория вычислимости – это первая теория информатики. В ней были разработаны методы классификации задач, деления их на алгоритмически разрешимые и неразрешимые. Это означает, что теория вычислимости даёт нам методы доказательства *отсутствия* алгоритмов решения некоторых конкретных задач. И основная цель данной главы – изучение таких методов.

В этой главе мы сосредоточим наше внимание на проблемах распознавания. Наша первая цель – показать, что существуют такие языки, которые не могут быть приняты никакой машиной Тьюринга. Это утверждение получается как очевидное следствие другого: число языков больше, чем число машин Тьюринга. Но число языков, как и число машин, бесконечно – поэтому нам необходимо научиться доказывать, что «одно из этих чисел больше другого». Для этого в разделе 5.2 мы познакомимся с одним из методов теории множеств – т. н. методом диагонализации. Более того, этот метод позволит нам доказать, что некоторый конкретный язык, называемый языком диагонализации, не принадлежит множеству рекурсивно перечислимых языков, т. е. $\mathcal{L}_{\mathrm{RE}}$.

Наша вторая цель – определить и более-менее подробно рассмотреть т. н. метод сводимости (сведéния, редукции). Этот метод и является главным инструментом доказательства неразрешимости различных проблем. Он позволяет нам доказать нерекурсивность многих конкретных языков – при условии, что мы уже имеем по крайней мере один язык, не принадлежащий $\mathcal{L}_{\mathrm{RE}}$. В разделе 5.3 мы используем этот метод для доказательства неразрешимости некоторых проблем принадлежности, связанных с машинами Тьюринга (программами для них). Таким образом мы покажем, что правильно поставленные задачи проверки корректности программ, вообще говоря, алгоритмически неразрешимы.

Раздел 5.4 посвящён теореме Райса – в которой говорится, что каждая нетривиальная задача является алгоритмически неразрешимой. В разделе 5.5 мы покажем, что метод сводимости может быть использован и для доказательства неразрешимости некоторых других задач – причём иначе, чем для проблем принадлежности в случае машин Тьюринга. В качестве примера такой задачи мы приведём проблему соответствий Поста – которую можно рассматривать в качестве специального варианта игры с фишками домино.

В разделе 5.6 представлен ещё один метод доказательства неразрешимости конкретных задач. Этот метод основан на сложности по Колмогорову – и может рассмат-

риваться как альтернатива методу диагонализации: можно ввести теорию вычислимости и доказать *все* необходимые для неё результаты, используя сначала сложность по Колмогорову, а потом – метод сводимости.

5.2 Метод диагонализации

Наша первая цель – показать, что существуют языки, не являющиеся рекурсивно перечислимыми. Чтобы сделать это, будем использовать следующий «вычислительный» аргумент.[1] Покажем, что

> *мощность* $|\mathrm{KodMT}|$ *множества машин Тьюринга меньше, чем мощность множества языков над алфавитом* Σ_{bool}.

Напомним, что KodMT – это множество двоичных представлений всех машин Тьюринга, описанное в разделе 4.5.

Число машин Тьюринга бесконечно; оно может быть ограничено сверху[2] мощностью $|(\Sigma_{\mathrm{bool}})^*|$ – поскольку $\mathrm{KodMT} \subseteq (\Sigma_{\mathrm{bool}})^*$. Мощность всех языков над алфавитом Σ_{bool} – это $|\mathcal{P}((\Sigma_{\mathrm{bool}})^*)|$; последнее значение тоже является т. н. трансфинитным. Чтобы доказать, что

$$\left|(\Sigma_{\mathrm{bool}})^*\right| < \left|\mathcal{P}\left((\Sigma_{\mathrm{bool}})^*\right)\right|,$$

нам необходим метод сравнения двух трансфинитных чисел (размеров двух бесконечных множеств).

Описанная далее теория Кантора, предназначенная для сравнения мощностей двух бесконечных множеств, затрагивает философские и аксиоматические (т. е. не требующие доказательства) основы математики и закладывает основные принципы современной теории множеств.

Определение 5.1 (концепция Кантора). *Пусть A и B – два множества. Будем говорить, что*

$$|\boldsymbol{A}| \leq |\boldsymbol{B}|,$$

если существует инъективная функция[3] *f, действующая из A в B. Будем говорить, что*

$$|\boldsymbol{A}| = |\boldsymbol{B}|,$$

если $|A| \leq |B|$ и $|B| \leq |A|$ (т. е. существует инъективная и сюръективная функция[4]*, действующая из A в B). Будем также говорить, что*

$$|\boldsymbol{A}| < |\boldsymbol{B}|,$$

если $|A| \leq |B|$, и при этом не существует взаимно-однозначного соответствия множеств A и B.

---

[1] Ср. доказательство леммы 2.52. *(Прим. перев.)*

[2] С точки зрения теории множеств, см. далее. *(Прим. перев.)*

[3] Функция f из A в B является инъективной (или просто инъекцией), если для всех $a, b \in A$ условие $a \neq b$ влечёт $f(a) \neq f(b)$.

[4] Функция f из A в B является сюръективной (или просто сюръекцией), если для каждого $b \in B$ существует $a \in A$, такое что $f(a) = b$. Функция, являющаяся одновременно инъективной и сюръективной, называется биективной – либо взаимно-однозначной.

Покажем неформально, что концепция, применяемая на основе определения 5.1, для двух *конечных* множеств A и B согласуется с нашим интуитивным пониманием сравнения мощности двух множеств (рис. 5.1). Если для всех $x, y \in A$ условие $x \neq y$ влечёт $f(x) \neq f(y)$ – то множество B должно содержать по крайней мере столько же элементов, сколько их в A. Если f – инъекция, и, кроме того,

$$\{f(x) \mid x \in A\} = B,$$

то f определяет разбиение элементов множеств A и B на пары $(x, f(x))$ – и, следовательно,

$$|A| = |B|.$$

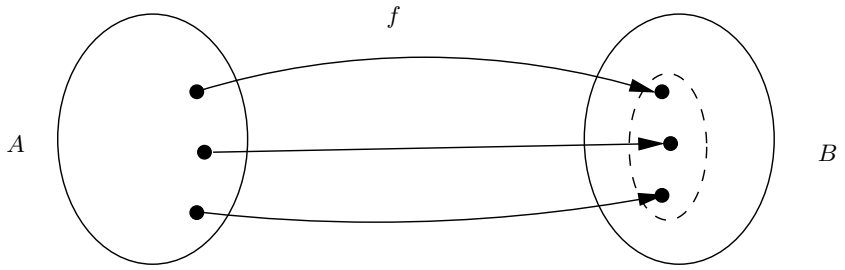


Рис. 5.1.

Теперь перейдём к бесконечным множествам, которые и нужны для наших целей. Согласно определению 5.1, нам достаточно доказать, что не существует взаимнооднозначного соответствия множества языков над алфавитом Σ_{bool} и множества машин Тьюринга. Следствием этого факта будет то, что *не* существует и взаимнооднозначного соответствия множеств $\mathcal{P}((\Sigma_{\text{bool}})^*)$ и KodMT – которое ставило бы в соответствие машине M язык L таким образом, что $L = L(M)$.[5]

Упражнение 5.2. Пусть A, B и C – некоторые множества. Докажите, что условия

$$|B| \leq |A| \quad \text{и} \quad |C| \leq |B| \quad \text{влекут} \quad |C| \leq |A|.$$

Упражнение 5.3. Пусть A и B – некоторые множества. Докажите, что $A \subseteq B$ влечёт $|A| \leq |B|$.

Определение 5.1 содержит выводы, которые могут показаться парадоксальными в «конечном мире». Допустим, что

$$\mathbb{N}_{\text{even}} = \{2i \mid i \in \mathbb{N}_0\}.$$

Из концепции Кантора, приведённой в определении 5.1, следует, что

$$|\mathbb{N}_0| = |\mathbb{N}_{\text{even}}|$$

– потому что функция $f : \mathbb{N}_0 \to \mathbb{N}_{\text{even}}$, определённая равенством

[5] Несуществование следует из того, что такое соответсвие должно быть инъективным.

$$f(i) = 2i$$

для всех $i \in \mathbb{N}$, является инъекцией[6] из $\mathbb{N}_0$ в $\mathbb{N}_{\text{even}}$. Всё это парадоксально для конечного мира, в нём такое просто невозможно: опыт конечного мира утверждает, что любой объект всегда больше, чем некоторая его часть – поэтому собственное подмножество B конечного множества A не может иметь такую же мощность, какую имеет само A. Но это не повод, чтобы утверждать, что концепция Кантора сравнения мощностей двух множеств содержит ошибку. Эта концепция потверждает лишь то, что мир бесконечных множеств и трансфинитных чисел подчиняется законам, не соответствующим нашему опыту, приобретённому в конечном мире. Результат – т. е. равенство $|\mathbb{N}_0| = |\mathbb{N}_{\text{even}}|$ – в мире бесконечных объектов оказывается правильным: взаимно-однозначное отображение $f(i) = 2i$ определяет пару $(i, 2i)$ элементов из $\mathbb{N}_0$ и $\mathbb{N}_{\text{even}}$ – вследствие чего можно утверждать, что оба эти множества имеют одинаковые мощности.

В тоже время нужно отметить, что концепция Кантора относится к аксиоматическому уровню математики. Следовательно, просто невозможно доказать, что эта концепция даёт возможность правильно (корректно) сравнивать размеры бесконечных множеств. И определение 5.1 – это только конкретная попытка формализовать интуитивное понимание сравнения мощностей в математике; нельзя исключать того, что существует другая, более удобная формализация.[7] Но, независимо от этого, значимость концепции Кантора заключается в возможности её использования для доказательства существования языков, не являющихся рекурсивно перечислимыми.

В дальнейшем мы рассматриваем $\mathbb{N}_0$ (или просто $\mathbb{N}$) в качестве «самого маленького» бесконечного множества. Это порождает вопросы типа – какие бесконечные множества имеют такую же мощность, как $\mathbb{N}$? существует ли бесконечно большое множество A, такое что $|A| > |\mathbb{N}|$?

Определение 5.4. *Множество A называется* **исчислимым (счётным)**, *если либо A конечное, либо $|A| = |\mathbb{N}_0|$.*[8]

Интуитивный смысл счётности множества A состоит в том, что элементы A можно расположить в некотором порядке (счётно) – первый, второй, третий, ... Это – естественное требование, поскольку однозначное отображение $f : A \to \mathbb{N}_0$ определяет линейный порядок на A – который и является перечислением.[9] Поэтому не удивительно, что множества $(\Sigma_{\text{bool}})^*$ и KodMT являются счётными.

Лемма 5.5. *Пусть Σ – конечный алфавит. Тогда множество Σ^* счётное.*

[6] Более того – биекцией.

[7] Эти аргументы похожи на приведённые выше рассуждения о тезисе Чёрча–Тьюринга.

[8] Приведём эквивалентное определение исчислимости (счётности) множества.

$$A \text{ счётное} \iff \text{существует биекция } f : A \to \mathbb{N}_0.$$

Это означает, что не существует бесконечно большого множества B, такого что $|B| < |\mathbb{N}_0|$ – т. е. $\mathbb{N}_0$ является одним из самых маленьких бесконечно больших множеств. Мы опускаем доказательство этого факта.

[9] Счётность множества A определяет линейный порядок со следующим свойством: между любыми двумя элементами A существует конечное множество элементов этого же множества.

Доказательство. Пусть $\Sigma = \{a_1, \ldots, a_m\}$ – непустое конечное множество. Установим на Σ линейный порядок $a_1 < a_2 < \cdots < a_m$. Этот линейный порядок порождает канонический порядок на Σ^* – согласно определению 2.16. Данный канонический порядок на Σ^* и есть пересчёт элементов множества Σ^* – следовательно, он определяет взаимно-однозначную функцию, действующую из Σ^* в $\mathbb{N}_0$. □

Теорема 5.6. *Множество* KodMT *кодов машин Тьюринга является счётным.*

Доказательство. Теорема 5.6 является прямым следствием леммы 5.5 и того факта, что $\mathrm{KodMT} \subseteq (\Sigma_{\mathrm{bool}})^*$. □

Упражнение 5.7. Опишите подробно пересчёт элементов множества $(\Sigma_{\mathrm{bool}})^*$, который согласуется с каноническим порядком на $(\Sigma_{\mathrm{bool}})^*$.

Упражнение 5.8. Докажите, что множество $\mathbb{Z}$ – счётное.

Упражнение 5.9. Пусть A – счётное множество, а a – некоторый элемент, не принадлежащий A. Докажите, что множество $A \cup \{a\}$ также является счётным.

Упражнение 5.10. Пусть A и B – счётные множества. Докажите, что множество $A \cup B$ также счётно.

А следующий вывод может показаться немного неожиданным – по крайней мере, с первого взгляда. Мы покажем, что мощность множества $\mathbb{Q}^+$ положительных рациональных чисел эквивалентна мощности $\mathbb{N}_0$. Удивительным при этом является то, что рациональные числа имеют высокую плотность на действительной оси – а именно, между двумя любыми различными рациональными числами существует бесконечно много различных рациональных чисел. А натуральные числа лежат на этой же оси с расстоянием в 1 – и поэтому всегда существует лишь конечное множество натуральных чисел между двумя выбранными натуральными числами. Так как каждое положительное рациональное число может быть представленно как p/q для некоторых $p, q \in \mathbb{N}$, мы можем предположить, что $|\mathbb{Q}^+|$ приблизительно равно $|\mathbb{N}_0 \times \mathbb{N}_0|$ – а последняя запись выглядит подобно «бесконечности, помноженной на бесконечность». В конечном мире можно говорить о сравнении n^2 и n. Однако $|\mathbb{Q}^+| = |\mathbb{N}_0|$ – потому что *можно* пересчитать элементы $\mathbb{Q}^+$.[10]

Итак, следующий метод доказательства счётности рациональных чисел, использующийся и в теории вычислимости, достаточно прост.

Лемма 5.11. $\mathbb{N} \times \mathbb{N}$ *является счётным.*

Доказательство. Рассмотрим бесконечно большую матрицу $M_{\mathbb{N}\times\mathbb{N}}$, приведённую на рис. 5.2. Матрица $M_{\mathbb{N}\times\mathbb{N}}$ имеет бесконечно много строк и бесконечно много столбцов, они помечены положительными целыми числами $1, 2, 3, \ldots$ Элемент $(i, j) \in \mathbb{N} \times \mathbb{N}$ лежит на пересечении i-ой строки и j-ого столбца. Очевидно, что $M_{\mathbb{N}\times\mathbb{N}}$ содержит все элементы множества $\mathbb{N} \times \mathbb{N}$.

Неудачной является попытка упорядочить элементы $\mathbb{N} \times \mathbb{N}$ путём выбора сначала элементов первой строки, затем второй и т.д. – потому что первая строка бесконечно

[10] Не обращая при этом внимания на их значения.

	1	2	3	4	5	6	⋯
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	⋯
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	⋯
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	⋯
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	⋯
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	⋯
6	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Рис. 5.2.

большая, и, следовательно, пересчёт её элементов никогда не закончится – т. е. пересчёт элементов второй строки никогда не начнётся. Удобная возможность пересчёта элементов матрицы $M_{\mathbb{N}\times\mathbb{N}}$ – использование зигзагообразного способа действий, согласно рис. 5.3. В этом случае мы берём одну за другой конечные диагонали – начиная с элемента $(1,1)$ в вернем левом углу. Итоговый пересчёт –

$$a_1 = (1,1),\ a_2 = (2,1),\ a_3 = (1,2),\ a_4 = (3,1),$$
$$a_5 = (2,2),\ a_6 = (1,3),\ a_7 = (4,1),\ \ldots\ .$$

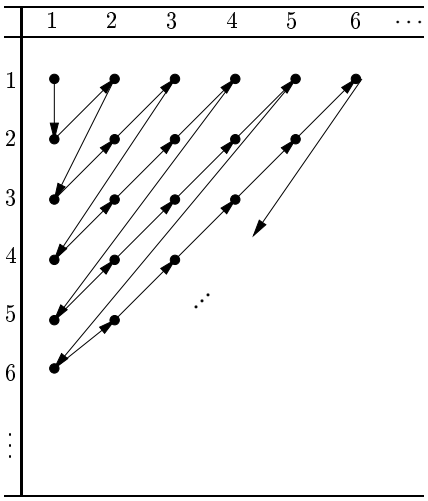


Рис. 5.3.

Формально мы определяем следующий линейный порядок на множестве $\mathbb{N}\times\mathbb{N}$:

$$(a,b) < (c,d) \iff a+b < c+d \text{ или } (a+b = c+d \text{ и } b < d).$$

Соответствующий пересчёт f можно описать как функцию

$$f(a,b) = \binom{a+b-1}{2} + b,$$

поскольку элемент (a, b) – это b-й элемент среди стоящих на $(a+b-1)$-й диагонали, а число элементов в первых $a+b-2$ диагоналях равно

$$\sum_{i=1}^{a+b-2} i = \frac{(a+b-2)\cdot(1+a+b-2)}{2} = \binom{a+b-1}{2}.$$

Очевидно, что f – это биекция из $\mathbb{N}\times\mathbb{N}$ в $\mathbb{N}_0$. □

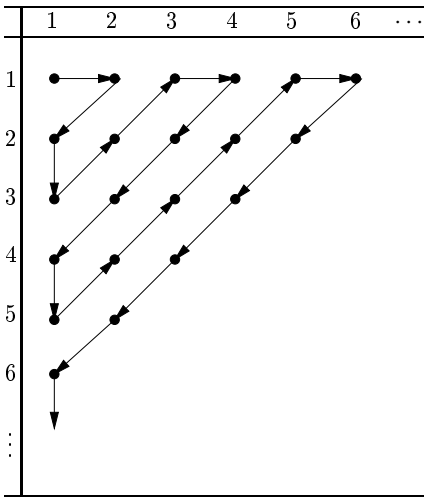


Рис. 5.4.

Упражнение 5.12. Чтобы доказать счётность множества $\mathbb{N}\times\mathbb{N}$, можно использовать пересчёт, определённый согласно рис. 5.4. Приведите описание соответствующей биекции из $\mathbb{N}\times\mathbb{N}$ в $\mathbb{N}_0$.

Упражнение 5.13. Сколько существует различных инъективных функций, действующих из $\mathbb{N}\times\mathbb{N}$ в $\mathbb{N}_0$?

Теорема 5.14. $\mathbb{Q}^+$ *счётно.*

Доказательство. Пусть h – следующая функция, действующая из $\mathbb{Q}^+$ в $\mathbb{N}\times\mathbb{N}$:

$$h\left(\frac{p}{q}\right) = (p, q)$$

для всех взаимно простых p и q.

Очевидно, что h – однозначное соответствие. Так как $\mathbb{N}\times\mathbb{N}$ счётно (лемма 5.11), то $\mathbb{Q}^+$ тоже счётно. □

Упражнение 5.15. Докажите, что множество $\mathbb{N}_0\times\mathbb{N}_0\times\mathbb{N}_0$ – счётное.

Упражнение 5.16. Пусть A и B – некоторые счётные множества. Докажите, что $A\times B$ также является счётным.

Итак, несмотря на плотность действительной оси, множество положительных рациональных чисел счётно. Согласно одному из утверждений упражнения 5.16, множество $(\mathbb{Q}^+)^i$ счётно для всех натуральных i. Обобщая все эти факты, можно было бы далее предположить, что *все* бесконечно большие множества счетны. Но в дальнейшем мы покажем, что множество действительных чисел, т. е. $\mathbb{R}$, несчётно. Следовательно, $\mathbb{R}$ – это «другая бесконечность», чем $\mathbb{N}_0$ и $\mathbb{Q}^+$.

Теорема 5.17. *Множество* $[0,1]$ *несчётно.*

Доказательство. Нам достаточно показать, что не существует инъективной функции (однозначного соответствия), действующей из $[0,1]$ в $\mathbb{N}$. Докажем этот факт от противного. Предположим, что множество $[0,1]$ счётно – т. е. пусть однозначное соответствие f, действующее из $[0,1]$ в $\mathbb{N}$, существует. Функция f определяет пересчёт действительных чисел из $[0,1]$ – в порядке, приведённом на рис. 5.5. При этом i-е число отрезка $[0,1]$ есть

$$a_i = 0.a_{i1}a_{i2}a_{i3}a_{i4}a_{i5}\ldots$$

(т. е. $f(a_i) = i$), где $a_{ij} \in \{0,1,2,\ldots,9\}$ для $j = 1,2,\ldots$

$f(x)$	$x \in [0,1]$						
1	0.	$\boxed{a_{11}}$	a_{12}	a_{13}	a_{14}	$\ldots$	
2	0.	a_{21}	$\boxed{a_{22}}$	a_{23}	a_{24}	$\ldots$	
3	0.	a_{31}	a_{32}	$\boxed{a_{33}}$	a_{34}	$\ldots$	
4	0.	a_{41}	a_{42}	a_{43}	$\boxed{a_{44}}$	$\ldots$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\ldots$	
i	0.	a_{i1}	a_{i2}	a_{i3}	a_{i4}	$\ldots$	$\boxed{a_{ii}}$ $\ldots$
$\vdots$	$\vdots$						

Рис. 5.5.

Далее воспользуемся т. н. **методом диагонализации** – чтобы показать, что по крайней мере одно действительное число из $[0,1]$ в таблице 5.5 пропущено – и, следовательно f не является инъективным отображением из $[0,1]$ в $\mathbb{N}$, т. е. f не обеспечивает счётность всех действительных чисел отрезка $[0,1]$. Действительные числа, показанные на рис. 5.5, могут быть представлены в виде бесконечно большой матрицы

$$M = [a_{ij}]_{i=1,\ldots,\infty,j=1,\ldots,\infty}.$$

Наша цель – сконструировать (построить) некоторое действительное число

$$c = 0.c_1c_2c_3c_4c_5\ldots,$$

которое отличается от любой из строк матрицы M (т. е. от всех действительных чисел рассматриваемого отрезка – при пересчёте этих чисел с помощью функции f).

Идея построения заключается в выборе числа c таким образом, чтобы для каждого $i \in \mathbb{N}$ оно отличалось от i-й строки таблицы M в i-м десятичном знаке:

$$c_i \neq a_{ii} \quad \text{и} \quad c_i \notin \{0,9\}.$$

Для этой цели мы рассмотрим диагональ $a_{11}a_{22}a_{33}\ldots$ матрицы M – и выберем для каждого натурального i какую-нибудь цифру из множества

$$c_i \in \{1,2,3,4,5,6,7,8\} - \{a_{ii}\}.$$

При таком выборе представление числа c отличается от представления любого числа a_i, приведённого на рис. 5.5, по крайней мере в i-м десятичном разряде a_{ii}. А поскольку десятичное представление числа c не содержит цифр 0 и 9, оно является единственным[11] для c – и, следовательно, для любого натурального i выполнено неравенство

$$c \neq a_i.$$

Поэтому выбранное нами число c *не* представлено на рис. 5.5, а функция f *не* является пересчётом действительных чисел из $[0,1]$. □

Мы показали, что множество машин Тьюринга (алгоритмов) является счётным. Поэтому для доказательства существования задач, являющихся алгоритмически неразрешимыми, нам достаточно доказать несчётность множества всех языков над алфавитом $\{0,1\}$; отметим, что множество этих языков можно рассматривать как множество проблем принадлежности.

Мы сделаем это двумя различными способами. Во-первых, покажем, что

$$|[0,1]| \leq |\mathcal{P}((\Sigma_{\text{bool}})^*)|.$$

А во-вторых, используя метод диагонализации мы непосредственно укажем язык, который не принимается никакой машиной Тьюринга.

Теорема 5.18. *Множество $\mathcal{P}((\Sigma_{\text{bool}})^*)$ не является счётным.*

Доказательство. Так как согласно теореме 5.17 отрезок $[0,1]$ является несчётным, нам достаточно показать, что

$$|\mathcal{P}((\Sigma_{\text{bool}})^*)| \geq |[0,1]| \tag{5.1}$$

с помощью описания взаимно-однозначного соответствия из $[0,1]$ в $\mathcal{P}((\Sigma_{\text{bool}})^*)$.

Каждое действительное число отрезка $[0,1]$ может быть представлено в двоичном виде следующим образом: представление

$$b = 0.b_1b_2b_3\ldots$$

с $b_i \in \Sigma_{\text{bool}}$ для $i = 1,2,3,\ldots$ кодирует действительное число

$$Number(b) = \sum_{i=1}^{\infty} a_i 2^{-i}.$$

Будем использовать это двоичное представление действительных чисел для определения функции $f : [0,1] \to \mathcal{P}((\Sigma_{\text{bool}})^*)$. Пусть

$$a = 0.a_1a_2a_3a_4a_5a_6a_7\ldots$$

[11] Заметим, что $1.00\overline{0}$ и $0.99\overline{9}$ суть два разных представления одного и того же числа 1. Аналогично, $0.142\overline{9}$ и $0.143\overline{0}$ представляют одно и то же число 0.143.

– двоичное представление некоторого действительного числа из интервала $[0,1]$. Определим

$$f(a) = \{a_1, a_2a_3, a_4a_5a_6, \ldots, a_{\binom{n}{2}+1}a_{\binom{n}{2}+2}\ldots a_{\binom{n+1}{2}}, \ldots\}.$$

Отметим, что $f(a)$ является языком над алфавитом Σ_{bool}, содержащим для любого натурального n ровно одно слово длины n. Поэтому любое отличие в каком-нибудь бите между двумя бинарными представлениями чисел b и c означает, что

$$f(b) \neq f(c).$$

Следовательно, f – однозначное соответствие, неравенство (5.1) выполнено, т. е. $\mathcal{P}((\Sigma_{\text{bool}})^*)$ не является счётным. □

Следствие 5.19. $|\text{KodMT}| < |\mathcal{P}((\Sigma_{\text{bool}})^*)|$ – *следовательно, существует бесконечное множество языков над алфавитом* Σ_{bool}*, не являющихся рекурсивно перечислимыми.*

Упражнение 5.20. Докажите, что $|[0,1]| = |\mathcal{P}((\Sigma_{\text{bool}})^*)|$.

	w_1	w_2	w_3	…	w_i	…
M_1	$\boxed{d_{11}}$	d_{12}	d_{13}	…	d_{1i}	…
M_2	d_{21}	$\boxed{d_{22}}$	d_{23}	…	d_{2i}	…
M_3	d_{31}	d_{32}	$\boxed{d_{33}}$	…	d_{3i}	…
⋮	⋮	⋮	⋮		⋮	
M_i	d_{i1}	d_{i2}	d_{i3}	…	$\boxed{d_{ii}}$	…
⋮	⋮	⋮	⋮		⋮	

Рис. 5.6.

Теперь применим метод диагонализации для иной цели – а именно, для описания конкретного языка, не являющегося рекурсивно перечислимым. Пусть

$$w_1,\ w_2,\ w_3,\ w_4,\ w_5,\ \ldots$$

– канонический порядок слов над алфавитом Σ_{bool}, а

$$M_1,\ M_2,\ M_3,\ M_4,\ M_5\ \ldots$$

– последовательность всех машин Тьюринга.[12] Определим бесконечно большую Булеву матрицу (рис. 5.6)

$$A = [d_{ij}]_{i,j=1,\ldots,\infty}$$

следующим образом:

$$d_{ij} = 1 \quad \Leftrightarrow \quad M_i \text{ принимает } j\text{-е слово } w_j.$$

В этом представлении i-я строка

[12] Как обычно, M_i обозначает i-ю машину Тьюринга.

$$d_{i1}d_{i2}d_{i3}d_{i4}d_{i5}\ldots$$

матрицы A определяет язык

$$L(M_i) = \{w_j \mid d_{ij} = 1 \text{ для всех } j \in \mathbb{N}\}.$$

Аналогично созданию конкретного действительного числа, которое оказалось не включённым в гипотетический пересчёт всех действительных чисел на рис. 5.5 (см. доказательство теоремы 5.17), мы создадим **язык диагонализации** L_{diag}, который не совпадает ни с каким языком $L(M_i)$. Для этого определим

$$\begin{aligned}\boldsymbol{L_{\text{diag}}} &= \{w \in (\Sigma_{\text{bool}})^* \mid w = w_i \text{ для } i \in \mathbb{N} \text{ и} \\ &\qquad M_i \text{ не принимает } w_i\} \\ &= \{w \in (\Sigma_{\text{bool}})^* \mid w = w_i \text{ для } i \in \mathbb{N} \text{ и } d_{ii} = 0\}.\end{aligned}$$

Теорема 5.21.

$$L_{\text{diag}} \notin \mathcal{L}_{\text{RE}} \ .$$

Доказательство. Докажем условие $L_{\text{diag}} \notin \mathcal{L}_{\text{RE}}$ от противного. Для этого предположим, что $L_{\text{diag}} \in \mathcal{L}_{\text{RE}}$. Тогда $L_{\text{diag}} = L(M)$ для некоторой машины Тьюринга M. Так как M является одной из перечисленных нами машин Тьюринга, существует некоторое натуральное число i, такое что $M = M_i$. Но язык L_{diag} не может совпадать с языком $L(M_i)$ – потому что

$$w_i \in L_{\text{diag}} \quad \Leftrightarrow \quad d_{ii} = 0 \quad \Leftrightarrow \quad w_i \notin L(M_i).$$

□

Упражнение 5.22. Рассмотрите язык

$$\begin{aligned}L_{2\text{diag}} = \{w \in (\Sigma_{\text{bool}})^* \mid w = w_{2i} \text{ для некоторого } i \in \mathbb{N} \text{ и} \\ M_i \text{ не принимает } w_{2i} \ (d_{i,2i} = 0)\}.\end{aligned}$$

Докажите, что $L_{2\text{diag}} \notin L_{\text{RE}}$.

Упражнение 5.23. Пусть для каждого $k \in \mathbb{N}_0$

$$\begin{aligned}L_{k,\text{diag}} = \{w \in (\Sigma_{\text{bool}})^* \mid w = w_{i+k} \text{ для некоторого } i \in \mathbb{N} \text{ и} \\ M_i \text{ не допускает } w_{i+k} \ (d_{i,i+k} = 0)\}.\end{aligned}$$

Докажите, что для каждого $k \in \mathbb{N}$ выполнено условие $L_{k,\text{diag}} \notin L_{\text{RE}}$.

5.3 Метод сводимости

Метод сводимости является наиболее общим методом классификации проблем принадлежности относительно их алгоритмической разрешимости. Смысл метода очень прост. Пусть A – некоторая проблема, для которой мы хотим показать алгоритмическую неразрешимость. Если мы укажем проблему B, такую что:

- уже известно, что B алгоритмически неразрешима;
- алгоритмическая разрешимость A означала бы алгоритмическую разрешимость B

– то можно заключить, что и проблема A алгоритмически неразрешима. Такой способ доказательства алгоритмической неразрешимости A называется сводимостью (сведéнием) проблемы A к B.

Определение 5.24. *Пусть $L_1 \subseteq \Sigma_1^*$ и $L_2 \subseteq \Sigma_2^*$ – некоторые языки над алфавитами Σ_1 и Σ_2. Будем говорить, что* $\boldsymbol{L_1}$ **(рекурсивно) сводим к** $\boldsymbol{L_2}$, *и писать при этом* $\boldsymbol{L_1 \leq_R L_2}$, *если*

$$L_2 \in \mathcal{L}_R \quad \Rightarrow \quad L_1 \in \mathcal{L}_R.$$

Запись

$$L_1 \leq_R L_2$$

соответствует нашему интуитивному представлению о том, что

> *относительно алгоритмической разрешимости язык L_2 является по крайней мере таким же трудным, как и L_1*

– поскольку если L_2 алгоритмически разрешим (т. е. $L_2 = L(A)$ для некоторого алгоритма A), то и L_1 должен быть алгоритмически разрешимым (т. е. $L_1 = L(B)$ для некоторого алгоритма B).

Мы уже знаем, что язык диагонализации L_{diag} является примером языка, который не принадлежит $\mathcal{L}_{RE}$ – и, следовательно, не принадлежит $\mathcal{L}_R$. Чтобы получить какие-либо иные нерекурсивные языки, нам необходимы конкретные методы проверки результатов типа $L_1 \leq_R L_2$. Мы познакомимся с двумя такими методами – соответствующими понятию рекурсивной сводимости.

Первый метод[13] относится к прямой интепретации термина «сводимость проблемы P_1 к проблеме P_2»: решение для P_2 может быть напрямую использованно как решение для P_1. Одним из названий соответствующего метода является «сводимость входа» – потому что в данном случае весь процесс сводимости представляет собой простое преобразование входной информации P_1 в другой вариант входной информации, P_2. Формально же мы строим некоторую МТ (алгоритм) M – которая для любого входа x проблемы принадлежности (Σ_1, L_1) вычисляет такой вход y проблемы принадлежности (Σ_2, L_2), что решения для входных примеров x (проблемы (Σ_1, L_1)) и y (проблемы (Σ_2, L_2)) одинаковы. Это означает, что если имеется алгоритм A для проблемы (Σ_2, L_2), то конкатенация (последовательное выполнение) алгоритмов M и A (рис. 5.7) представляет собой алгоритм решения проблемы (Σ_1, L_1).

Определение 5.25. *Пусть $L_1 \subseteq \Sigma_1^*$ и $L_2 \subseteq \Sigma_2^*$ – языки над алфавитами Σ_1 и Σ_2. Будем говорить, что язык L_1* **сводим с помощью отображения к языку** $\boldsymbol{L_2}$,[14] *и писать при этом* $\boldsymbol{L_1 \leq_m L_2}$, *если существует некоторая* МТ M, *которая для каждого $x \in \Sigma_1^*$ вычисляет отображение*

[13] Забегая вперёд, отметим следующее. В дальнейшем тексте автор явно не укажет на *второй* метод. Имеется в виду указание *конкретного* языка L, для которого выполнено условие $L \notin \mathcal{L}_R$ – и такие примеры будут приведены далее. *(Прим. перев.)*

[14] В литературе на русском языке употребляются также термины «сводимость соответствия», «неоднозначная сводимость» и др. Английская терминология тоже не отличается однозначностью: «mapping reducibility», «many-one-reducibility» – возможно, и др. *(Прим. перев.)*

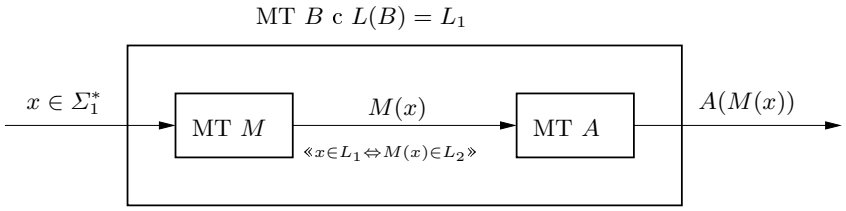


Рис. 5.7.

$$f_M : \Sigma_1^* \to \Sigma_2^*$$

со следующим свойством (рис. 5.7) :

$$x \in L_1 \quad \Leftrightarrow \quad f_M(x) \in L_2 \,.$$

При этом функция f_M называется **функцией сводимости** *языка L_1 к L_2; мы также говорим, что* $\boldsymbol{M}$ **сводит язык $\boldsymbol{L_1}$ к языку $\boldsymbol{L_2}$**.

Следующая лемма показывает, что отношение $\leq_{\mathrm{m}}$ является специальным случаем отношения $\leq_{\mathrm{R}}$. Например, для доказательства того, что $L_1 \leq_{\mathrm{R}} L_2$, достаточно доказать, что $L_1 \leq_{\mathrm{m}} L_2$.

Лемма 5.26. *Пусть $L_1 \subseteq \Sigma_1^*$ и $L_2 \subseteq \Sigma_2^*$ – языки над алфавитами Σ_1 и Σ_2. Если $L_1 \leq_{\mathrm{m}} L_2$, то $L_1 \leq_{\mathrm{R}} L_2$.*

Доказательство. Пусть $L_1 \leq_{\mathrm{m}} L_2$. Чтобы доказать, что $L_1 \leq_{\mathrm{R}} L_2$, достаточно показать существование алгоритма A (т. е. машины Тьюринга A, которая всегда завершает работу), решающего проблему L_2 ($L_2 \in \mathcal{L}_{\mathrm{R}}$) – согласно предположению, что существует алгоритм B, решающий проблему L_1.

Пусть A – некоторая МТ, которая всегда останавливается, причём $L(A) = L_2$. Предположив, что $L_1 \leq_{\mathrm{m}} L_2$, мы получаем отсюда существование машины Тьюринга M, которая для каждого $x \in \Sigma_1^*$ вычисляет слово $M(x) \in \Sigma_2^*$, такое что

$$x \in L_1 \quad \Leftrightarrow \quad M(x) \in L_2.$$

На основе A и M опишем машину Тьюринга B, которая всегда останавливается, и при этом принимает язык L_1 (рис. 5.7). B работает над входом $x \in \Sigma_1^*$ следующим образом:

- Сначала B моделирует работу M над входом x – до того момента, пока на ленте не будет записано слово $M(x)$.
- Далее B моделирует работу A над $M(x)$.
  Если A принимает $M(x)$, то B принимает x.
  Если A отклоняет $M(x)$, то B отклоняет x.

Очевидно, что $L(B) = L_1$. Поскольку A всегда завершает работу,[15] то B тоже завершает работу для всех возможных входных данных. Следовательно $L_1 \in \mathcal{L}_{\mathrm{R}}$. □

[15] Заметим, что M вычисляет $M(x)$ для всех $x \in \Sigma_1^*$ – поэтому M также завершает работу для любого входа.

Упражнение 5.27. Докажите, что $\leq_{\mathrm{m}}$ является транзитивным отношением, т. е. что

$$(L_1 \leq_{\mathrm{m}} L_2 \text{ и } L_2 \leq_{\mathrm{m}} L_3) \Rightarrow L_1 \leq_{\mathrm{m}} L_3.$$

Отметим, что машина B с $L(B) = L_1$ использует машину A в качестве подпрограммы. Эта подпрограмма A выполняется в точности один раз (рис. 5.7), и мы считаем, что выходами B являются выходы A. Однако последнее не является существенным ограничением: можно построить машину B таким образом, что B выполнит работу A для нескольких различных входов, после чего использует выходные данные A для окончательного ответа на вопрос, входит ли x в язык L_1 (рис. 5.8).

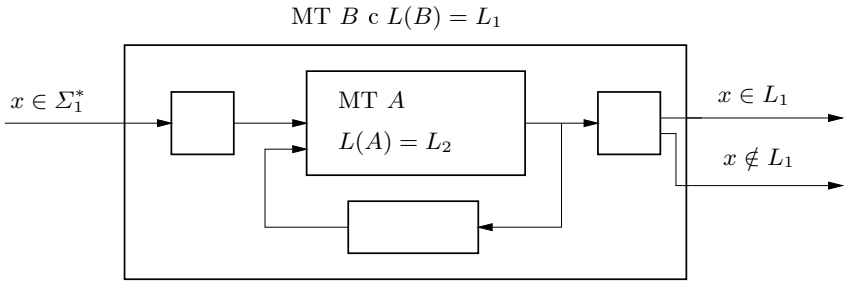


Рис. 5.8.

Теперь получим важные результаты, являющиеся следствием свойства $L \notin \mathcal{L}_{\mathrm{R}}$ для конкретного языка L. Начнём с общего утверждения – о том, что некоторый язык принадлежит классу $\mathcal{L}_{\mathrm{R}}$ тогда и только тогда, когда и его дополнение принадлежит $\mathcal{L}_{\mathrm{R}}$.

Лемма 5.28. *Пусть Σ – некоторый алфавит. Для любого языка $L \subseteq \Sigma^*$ выполнено следующее:*

$$L \leq_{\mathrm{R}} L^{\complement} \quad \textit{и} \quad L^{\complement} \leq_{\mathrm{R}} L.$$

Доказательство. Нам достаточно доказать, что $L^{\complement} \leq_{\mathrm{R}} L$ для каждого языка L: поскольку выполнено равенство $(L^{\complement})^{\complement} = L$, то из $L^{\complement} \leq_{\mathrm{R}} L$ следует, что $(L^{\complement})^{\complement} \leq_{\mathrm{R}} L^{\complement}$ (в условии $L^{\complement} \leq_{\mathrm{R}} L$ мы заменяем язык $L^{\complement}$ на L).

Пусть A – некоторый алгоритм, решающий проблему принадлежности (L, Σ). Алгоритм B, решающий аналогичную проблему для $L^{\complement}$, приведён на рис. 5.9: B просто передаёт свои входные данные $x \in \Sigma^*$ для подпрограммы A, а потом, после работы A, заменяет её ответ на обратный.

Приведём ещё одно доказательство того, что $L^{\complement} \leq_{\mathrm{R}} L$ – используя формализм машин Тьюринга. Пусть $A = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{accept}}, q_{\mathrm{reject}})$ – некоторая МТ, которая всегда завершает работу и принимает язык L_1. Рассмотрим машину $B = (Q, \Sigma, \Gamma, \delta, q_0, q'_{\mathrm{accept}}, q'_{\mathrm{reject}})$, у которой

$$q'_{\mathrm{accept}} = q_{\mathrm{reject}} \quad \text{и} \quad q'_{\mathrm{reject}} = q_{\mathrm{accept}};$$

эта машина и является искомой – благодаря изменению ролей принятия и отклонения состояний машины A. Поскольку A всегда останавливается, B также всегда останавливается – следовательно, $L(B) = (L(A))^{\complement}$. А поэтому существование алгоритма для языка L означает существование алгоритма для языка $L^{\complement}$. □

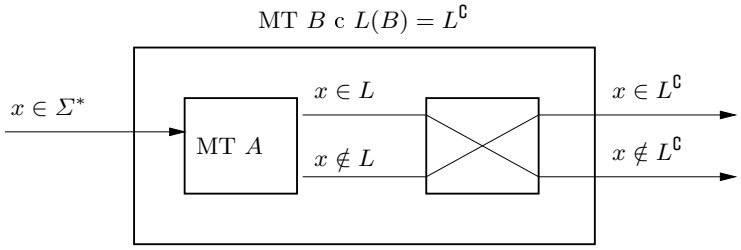


Рис. 5.9.

Следствие 5.29. $(L_{\text{diag}})^{\complement} \notin \mathcal{L}_{\text{R}}$.

Доказательство. В теореме 5.18 мы показали, что $L_{\text{diag}} \notin \mathcal{L}_{\text{RE}}$ – следовательно, $L_{\text{diag}} \notin \mathcal{L}_{\text{R}}$. Вследствие леммы 5.28,

$$L_{\text{diag}} \leq_{\text{R}} (L_{\text{diag}})^{\complement}.$$

А согласно определению отношения $\leq_{\text{R}}$ мы получаем, что

$$(L_{\text{diag}})^{\complement} \in \mathcal{L}_{\text{R}} \text{ влечёт } L_{\text{diag}} \in \mathcal{L}_{\text{R}}.$$

Отсюда $(L_{\text{diag}})^{\complement} \notin \mathcal{L}_{\text{R}}$. □

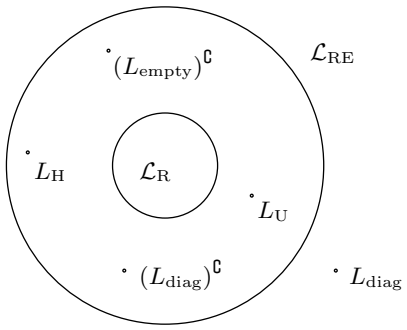


Рис. 5.10.

Однако на основе последнего утверждения нельзя заключить, что $(L_{\text{diag}})^{\complement}$ не принадлежит множеству $\mathcal{L}_{\text{RE}}$. Более того, истинным является противоположное утверждение. Доказав, что $(L_{\text{diag}})^{\complement} \in \mathcal{L}_{\text{RE}}$, мы отсюда получаем, что $\mathcal{L}_{\text{R}} \subsetneq \mathcal{L}_{\text{RE}}$ – см. рис. 5.10.

Лемма 5.30. $(L_{\text{diag}})^{\complement} \in \mathcal{L}_{\text{RE}}$.

Доказательство. Согласно определению языка L_{diag}, мы получаем, что

$$(L_{\text{diag}})^{\complement} = \{x \in (\Sigma_{\text{bool}})^* \mid \text{если } x = w_i \text{ для некоторого } i \in \mathbb{N}, \\ \text{то } M_i \text{ принимает } w_i\}.$$

Машину Тьюринга D, допускающую язык $(L_{\text{diag}})^{\complement}$, можно описать следующим образом:

Вход: $x \in (\Sigma_{\text{bool}})^*$.
Этап 1. Вычисляется натуральное i – такое, что x является i-м словом w_i относительно канонического порядка над алфавитом Σ_{bool}.
Этап 2. Генерируется код $\text{Kod}(M_i)$ i-й машины Тьюринга M_i.
Этап 3. Моделируется вычисление M_i над словом $w_i = x$.
Если M_i принимает w_i, то D также принимает x.
Если M_i отклоняет[16] w_i, то D также отклоняет $x = w_i$.
Если вычисление машины M_i над w_i бесконечно, (т. е. $w_i \notin L(M_i)$), то D моделирует эти бесконечные вычисления. Следовательно, D не останавливается на входе x, и, таким образом, $x \notin L(D)$.

Очевидно, что $L(D) = (L_{\text{diag}})^{\complement}$. □

Следствие 5.31. $(L_{\text{diag}})^{\complement} \in \mathcal{L}_{\text{RE}} - \mathcal{L}_{\text{R}}$, *поэтому*

$$\mathcal{L}_{\text{R}} \subsetneq \mathcal{L}_{\text{RE}}.$$

В дальнейшем мы познакомим с языками, которые не являются рекурсивными, но принадлежат классу $\mathcal{L}_{\text{RE}}$ (рис. 5.10).

Определение 5.32. Универсальный язык – *это язык*

$$\boldsymbol{L_{\text{U}}} = \{\text{Kod}(M)\#w \mid w \in (\Sigma_{\text{bool}})^* \quad \textit{и} \quad M \textit{ принимает } w\}.$$

Теорема 5.33. *Существует машина Тьюринга* U, *называемая* **универсальной МТ**, *такая что*

$$L(U) = L_{\text{U}}.$$

Следовательно, $L_{\text{U}} \in \mathcal{L}_{\text{RE}}$.

Доказательство. Достаточно описать 2-ММТ U, допускающую язык $L(U) = L_{\text{U}}$. Машина U работает следующим образом.

Вход: $z \in \{0, 1, \#\}^*$.
Этап 1. U проверяет, содержит ли z ровно один символ $\#$. Если да, то U переходит на этап 2. Если нет, то U отклоняет свой вход z.
Этап 2. Пусть $z = y\#x$, где $y, z \in (\Sigma_{\text{bool}})^*$. U проверяет, является ли y кодом некоторой машины Тьюринга. Если y не представляет собой код какой-либо МТ, то U отклоняет свой вход $z = y\#x$. Если же y кодирует некоторую МТ, то U переходит на этап 3.
Этап 3. Итак, $y = \text{Kod}(M)$ для некоторой машины Тьюринга M. U записывает инициальную конфигурацию этой машины M над словом x на свою первую рабочую ленту, после чего переходит на этап 4.
Этап 4. Машина U пошагово моделирует вычисления машины M над словом x следующим образом:

`while` состояние конфигурации M на первой рабочей ленте отличается от q_{accept} и q_{reject} машины M

[16] Напомним, что отклонение входа означает останов работы машины в состоянии q_{reject}.

```
    do моделировать один шаг вычислений машины M
    {машина U делает это путём чтения кода Kod(M)
    на своей входной ленте}
  if состояние машины M есть q_accept
    then U принимает z = Kod(M)#x
    else U отклоняет z = Kod(M)#x
```

Заметим, что бесконечное вычисление M над x является причиной бесконечного вычисления[17] машины U над словом $\mathrm{Kod}(M)\#x$. Таким образом, в этом случае U не принимает $\mathrm{Kod}(M)\#x$ – и, следовательно, $L(U) = L_{\mathrm{U}}$. □

Что означает условие $L_{\mathrm{U}} \in \mathcal{L}_{\mathrm{RE}}$? Он даёт гарантию существования некоторой машины Тьюринга (программы), которая может моделировать работу произвольной МТ (программы) над любым входом – но, вообще говоря, не гарантирует завершения такой работы. Это – естественное свойство машин Тьюрига, которое является необходимым для любого формального определения алгоритмов.[18] «В мире языков программирования» универсальная машина Тьюринга является интерпретатором. Не существует вычислительной системы, которая *не* может выполнять синтаксически правильные программы для данного входа – конечно, при условии, что соответствующий язык программирования[19] является *частью* этой системы.

В дальнейшем мы покажем, что $L_{\mathrm{U}} \notin \mathcal{L}_{\mathrm{R}}$. Следствием последнего результата является тот факт, что единственно возможная общая стратегия (алгоритм) определения результата вычисления машины M над входом x состоит в моделировании вычисления M над x. Но если вычисление M над x является бесконечным, то мы на этапе моделирования этого вычисления не сможем определить, является ли вычисление M над x бесконечным – или же M завершит работу после следующего шага вычисления и выведет результат. Следовательно, для заданных M и x, вообще говоря, невозможно за конечное время ответить на вопрос, принадлежит ли x языку $L(M)$.

На методе сводимости основано несколько следующих доказательств. Большинство утверждений будут доказаны дважды: сначала приводится очевидное доказательство «на уровне алгоритмов» – как программ на некотором языке программирования с использованием общего метода сводимости, рис. 5.8; а затем приводится строгое доказательство с использованием формализма машин Тьюринга – методом сводимости входа, рис. 5.11.

Теорема 5.34. $L_{\mathrm{U}} \notin \mathcal{L}_{\mathrm{R}}$.

Доказательство. Достаточно показать, что $(L_{\mathrm{diag}})^{\mathtt{C}} \leq_{\mathrm{R}} L_{\mathrm{U}}$, поскольку, согласно следствию 5.29, мы получаем, что $(L_{\mathrm{diag}})^{\mathtt{C}} \notin \mathcal{L}_{\mathrm{R}}$ – и поэтому $L_{\mathrm{U}} \notin \mathcal{L}_{\mathrm{R}}$.

Пусть A – некоторый алгоритм (программа), который решает проблему L_{U}. Опишем алгоритм B, который использует A как подпрограмму для решения проблемы $(L_{\mathrm{diag}})^{\mathtt{C}}$. Структура алгоритма B приведена на рис. 5.11. Для любого входа $x \in (\Sigma_{\mathrm{bool}})^*$ подпрограмма C сначала вычисляет некоторое число i, такое что $x = w_i$. Затем C вычисляет код $\mathrm{Kod}(M_i)$ i-й машины Тьюринга. C выдаёт свои выходные данные, т. е.

[17] В данном случае можно сказать – моделирования этого вычисления.

[18] Существует система аксиом, которая должна удовлетворять вычислительной модели. И существование универсальной машины является одной из таких аксиом.

[19] Напомним, что любой язык программирования – это формальная модель алгоритмов.

w_i и $\mathrm{Kod}(M_i)$, в качестве входных данных для подпрограммы A.[20] Окончательно B «наследует» выходную информацию, «принятую» или «отклонённую» A, в качестве своего собственного выхода. Очевидно, что $L(B) = (L_{\mathrm{diag}})^{\complement}$ – и, кроме того, B всегда останавливается, поскольку всегда останавливаются как C, так и A. Следовательно, $(L_{\mathrm{diag}})^{\complement} \leq_{\mathrm{R}} L_{\mathrm{U}}$, что и завершает доказательсто теоремы.

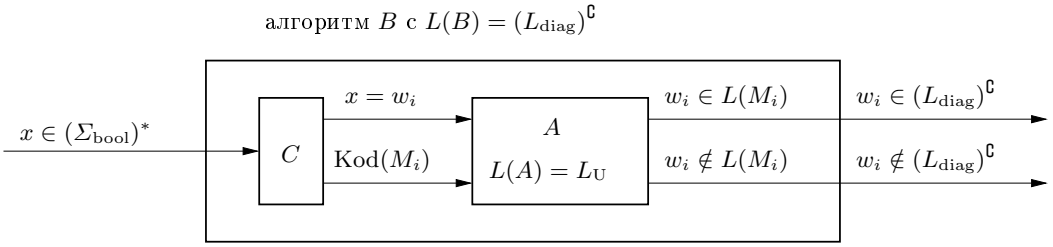


Рис. 5.11.

Теперь приведём альтернативное доказательство – в рамках формализма машин Тьюринга. Мы докажем, что

$$(L_{\mathrm{diag}})^{\complement} \leq_{\mathrm{m}} L_{\mathrm{U}}.$$

Для этого опишем машину M, которая вычисляет отображение f_M из $(\Sigma_{\mathrm{bool}})^*$ в $\{0,1,\#\}^*$, такое что

$$x \in (L_{\mathrm{diag}})^{\complement} \Leftrightarrow f_M(x) \in L_{\mathrm{U}}.$$

Машина M работает следующим образом. Для каждого входа x она сначала вычисляет некоторое i, такое что $x = w_i$. После этого она вычисляет код $\mathrm{Kod}(M_i)$ i-й МТ. M останавливается с содержимым ленты $\mathrm{Kod}(M_i)\#x$. Поскольку $x = w_i$, мы получаем согласно определению языка $(L_{\mathrm{diag}})^{\complement}$:

$$\begin{aligned} x = w_i \in (L_{\mathrm{diag}})^{\complement} &\Leftrightarrow M_i \text{ принимает } w_i \\ &\Leftrightarrow w_i \in L(M_i) \\ &\Leftrightarrow f_M(x) = \mathrm{Kod}(M_i)\#x \in L_U. \end{aligned}$$

Упражнение 5.35. Покажите, что следующий язык

$$\begin{aligned} \{\mathrm{Kod}(M)\#x\#0^i \mid\ & x \in \{0,1\}^*,\ i \in \mathbb{N}_0, \\ & M \text{ имеет по крайней мере } i+1 \text{ состояний}, \\ & \text{и в течение вычисления над словом } x \\ & \text{по крайней мере 1 раз достигает } i\text{-го состояния}\} \end{aligned}$$

не является рекурсивным.

Мы видим, что основные задачи теории вычислимости относятся к проблеме останова машин Тьюринга – т. е. конечности её вычислений. Для языка $(L_{\mathrm{diag}})^{\complement}$ и L_{U} у нас есть машины Тьюринга (программы), которые *принимают* эти языки – но нет машин Тьюринга, которые *решают* эти языки (т. е. принимают эти языки и не имеют бесконечных вычислений). Итак, рассмотрим следующую очень важную проблему.

[20] Отметим, что формально входом машины A является слово $\mathrm{Kod}(M_i)\#w_i$.

Определение 5.36. Проблема останова *является проблемой принадлежности* $(\{0,1,\#\}^*, L_{\mathrm{H}})$, *где*

$$L_{\mathrm{H}} = \{\mathrm{Kod}(M)\#x \mid x \in \{0,1\}^* \text{ и } M \text{ останавливается на } x\}.$$

Упражнение 5.37. Докажите, что $L_{\mathrm{H}} \in \mathcal{L}_{\mathrm{RE}}$.

Следующий результат показывает, что не существует алгоритма, проверяющего, завершает ли некоторая программа свою работу.

Теорема 5.38. $L_{\mathrm{H}} \notin \mathcal{L}_{\mathrm{R}}$.

Доказательство. Сначала приведём доказательство на программном уровне, опишем применение метода сводимости согласно рис. 5.8. Мы покажем, что

$$L_{\mathrm{U}} \leq_{\mathrm{R}} L_{\mathrm{H}}.$$

Предположим, что $L_{\mathrm{H}} \in \mathcal{L}_{\mathrm{R}}$ – т. е. пусть существует алгоритм A, решающий проблему L_{H}. При таком предположении опишем алгоритм B (рис. 5.12) для определения универсального языка L_{U} – использующий алгоритм A в качестве подпрограммы. А именно, для любого входа w алгоритм B сначала использует подпрограмму C для проверки, имеет ли w вид $y\#x$ – где $y = \mathrm{Kod}(M)$ для некоторой машины Тьюринга M, а $x \in (\Sigma_{\mathrm{bool}})^*$.

Если y не представим в таком виде, то B отклоняет w.

Если же $y = \mathrm{Kod}(M)\#x$, то B передаёт y в качестве входа для подпрограммы A.

Если выходом алгоритма A является «M не останавливается на x», то B знает, что $x \notin L(M)$ – и, следовательно, может немедленно отвергнуть свой вход $w = \mathrm{Kod}(M)\#x$.

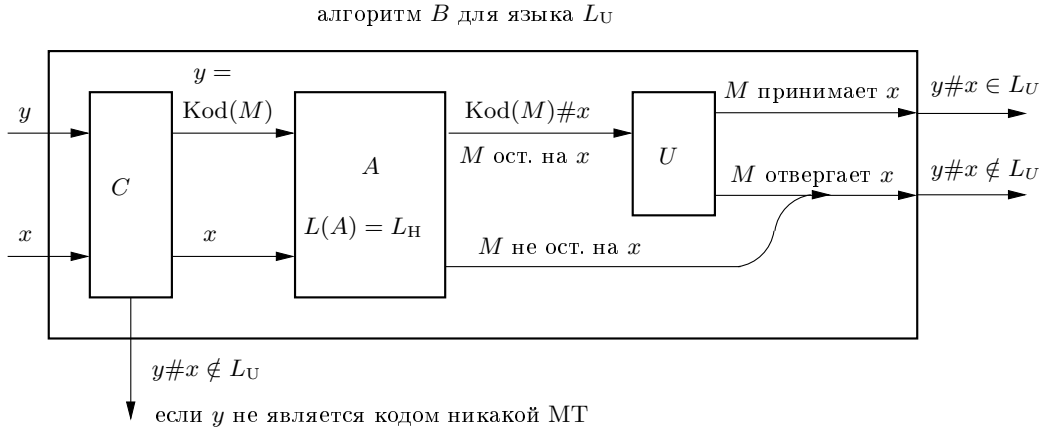


Рис. 5.12.

Если выходом алгоритма A является «M останавливается на x», то B моделирует работу M над входом x с помощью подпрограммы U. Поскольку вычисления машины M над x конечны, U выполняет это моделирование за конечное время.

Если при таком моделировании вычислений ответом U является «M принимает x», то B принимает вход $y\#x = \mathrm{Kod}(M)\#x$. Если же ответом U является «M отклоняет x», то B отклоняет этот вход.

Очевидно, что, $L(B) = L_{\mathrm{U}}$, и при этом B всегда останавливается – что и завершает доказательство.

Теперь докажем тот же факт, т. е. что

$$L_{\mathrm{U}} \leq_{\mathrm{R}} L_{\mathrm{H}},$$

используя формализм машин Тьюринга. Для этого опишем некоторую машину M, которая сводит язык L_{U} к языку L_{H}. Для любого входа w машина M работает следующим образом. Во-первых, она проверяет, имеет ли w вид

$$w = \mathrm{Kod}(\overline{M})\#x$$

для машины $\overline{M}$ и слова $x \in (\Sigma_{\mathrm{bool}})^*$.

- Если w не является словом такого вида, то M генерирует код $\mathrm{Kod}(M_1)$ некоторой машины Тьюринга M_1, которая для любого входа попадает в цикл[21] – $\delta(q_0, ¢) = (q_0, ¢, \mathrm{N})$. При этом M завершает работу с содержимым ленты

$$M(w) = \mathrm{Kod}(M_1)\#x.$$

- Если $w = \mathrm{Kod}(\overline{M})\#x$, то M изменяет код машины $\overline{M}$ – получая при этом машину M_2 с языком $L(M_2) = L(\overline{M})$. Машина M_2 работает в точности так, как $\overline{M}$ – со следующим исключением: все переходы, ведущие в состояние q_{reject} машины $\overline{M}$, заменены на переходы в новое состояние p, такое что $\delta(p, a) = (p, a, \mathrm{N})$ для всех $a \in \Sigma$. Следовательно, M_2 никогда не отклоняет входные данные – и для всех входов y, таких что $y \notin L(\overline{M}) = L(M_2)$, будет бесконечно выполнять вычисления. M заканчивает работу с содержимым ленты

$$M(w) = \mathrm{Kod}(M_2)\#x.$$

Теперь докажем, что для всех $w \in \{0, 1, \#\}^*$ выполнено следующее:

$$w \in L_{\mathrm{U}} \Leftrightarrow M(w) \in L_{\mathrm{H}}.$$

Пусть $w \in L_{\mathrm{U}}$. Тогда $w = \mathrm{Kod}(\overline{M})\#x$ для некоторой машины Тьюринга $\overline{M}$, поэтому $x \in \{0, 1\}^*$, а отсюда $x \in L(\overline{M})$. Поскольку $L(M_2) = L(\overline{M})$, машина M останавливается на входе x и принимает его. Следовательно, слово $M(w) = \mathrm{Kod}(M_2)\#x$ принадлежит языку L_{H}.

Теперь пусть $w \notin L_{\mathrm{U}}$; при этом будем различать следующие две возможности. Если слово w не представимо в виде $\mathrm{Kod}(\overline{M})\#x$ для некоторой машины Тьюринга $\overline{M}$, то положим $M(w) = \mathrm{Kod}(M_1)\#x$, где M_1 не завершает работу ни на каком входе. Следовательно, M_1 также не завершает работу на x, и поэтому $M(w)$ не принадлежит языку L_{H}. Если же w имеет вид $\mathrm{Kod}(\overline{M})\#x$ для некоторой машины $\overline{M}$, и $\mathrm{Kod}(\overline{M})\#x \notin L_{\mathrm{U}}$, то $x \notin L(\overline{M})$. В этом случае $M(w) = \mathrm{Kod}(M_2)\#x$, причём машина M_2 не завершает работу ни для какого входа $(\Sigma_{\mathrm{bool}})^* - L(\overline{M})$. Так как $x \notin L(\overline{M})$, машина $\overline{M}$ не заканчивает работу на входе x, и поэтому $\mathrm{Kod}(\overline{M})\#x$ не принадлежит языку L_{H}. □

[21] Т. е. M_1 не останавливается ни на каком входе.

Упражнение 5.39. Докажите следующие утверждения:

- $L_{\mathrm{U}} \leq_{\mathrm{R}} (L_{\mathrm{diag}})^{\complement}$;
- $L_{\mathrm{H}} \leq_{\mathrm{R}} (L_{\mathrm{diag}})^{\complement}$;
- $L_{\mathrm{diag}} \leq_{\mathrm{R}} L_{\mathrm{U}}$;
- $(L_{\mathrm{diag}})^{\complement} \leq_{\mathrm{R}} L_{\mathrm{H}}$;
- $L_{\mathrm{H}} \leq_{\mathrm{R}} L_{\mathrm{U}}$.

Теперь рассмотрим следующий язык

$$\boldsymbol{L_{\mathbf{empty}}} = \{\mathrm{Kod}(M) \mid L(M) = \emptyset\}.$$

Он включает коды машин Тьюринга, принимающих т. н. пустой язык – т. е., иными словами, не принимающих никакого входа. Очевидно, что

$$(L_{\mathrm{empty}})^{\complement} = \{x \in (\Sigma_{\mathrm{bool}})^* \mid x \neq \mathrm{Kod}(\overline{M}) \text{ для всех МТ } \overline{M}, \text{ либо } x = \mathrm{Kod}(M) \text{ и } L(M) \neq \emptyset\}.$$

Лемма 5.40. $(L_{\mathrm{empty}})^{\complement} \in \mathcal{L}_{\mathrm{RE}}$.

Доказательство. Приведём два разных доказательства того факта, что $(L_{\mathrm{empty}})^{\complement} \in \mathcal{L}_{\mathrm{RE}}$. Первое доказательство показывает, что действительно полезно иметь модель недетерминированной машины Тьюринга. А второе доказательство показывает, как можно применять метод, использовавшийся в теории множеств, в лемме 5.11, – для доказательства того, что $|\mathbb{N}_0| = |\mathbb{Q}^+|$.

Так как для любой НМТ M_1 существует МТ M_2 такая, что $L(M_1) = L(M_2)$, нам достаточно показать, что существует НМТ M_1 с языком $L(M_1) = (L_{\mathrm{empty}})^{\complement}$. Опишем такую машину – которая работает над входом x следующим образом.

Этап 1. M_1 детерминированно проверяет, выполнено ли для некоторой МТ M условие $x = \mathrm{Kod}(M)$. Если x не кодирует никакую машину Тьюринга, то M_1 допускает слово x.

Этап 2. Если $x = \mathrm{Kod}(M)$ для некоторой МТ M, то M недетерминированно генерирует слово $y \in (\Sigma_{\mathrm{bool}})^*$ и детерминированно моделирует вычисление M над этим словом y.

Этап 3. Если M принимает y (т. е. если $L(M) \neq \emptyset$), то M_1 допускает свой вход $x = \mathrm{Kod}(M)$.

Если M отклоняет y, то M_1 не принимает x в этом вычислении.

Если вычисление M над y бесконечно, то и M не завершает работу над словом x – и, следовательно, в этом вычислении слово $x = \mathrm{Kod}(M)$ не принимается.

Итак, согласно описанию этапа 1, M_1 принимает все слова, которые не кодируют машины Тьюринга.

Если $x = \mathrm{Kod}(M)$ для некоторой МТ M, и при этом $L(M) \neq \emptyset$, то существует слово $y \in L(M)$. Следовательно, существует допускающее вычисление машины M_1 над $x = \mathrm{Kod}(M)$.

Если же $x \in L_{\mathrm{empty}}$, то существует допускающее вычисление машины M_1 над словом $x = \mathrm{Kod}(M)$ – и, следовательно

$$L(M_1) = (L_{\mathrm{empty}})^{\complement}.$$

Теперь приведём другое доказательство того, что $(L_{\text{empty}})^{\complement} \in \mathcal{L}_{\text{RE}}$. Для этого мы непосредственно опишем детерминированную машину Тьюринга A, которая допускает язык $(L_{\text{empty}})^{\complement}$. Над каждым входом w машина A работает следующим образом.

Этап 1′. Если w не является кодом никакой МТ, то A принимает w.

Этап 2′. Если же $w = \text{Kod}(M)$ для некоторой МТ M, то машина A работает следующим образом.

A последовательно генерирует все пары $(i, j) \in \mathbb{N} \times \mathbb{N}$ – например, в порядке, приведённом на рис. 5.3 или 5.4.

Для каждой такой пары (i, j) машина A генерирует i-е слово w_i относительно канонического порядка над входным алфавитом машины M, после чего моделирует j шагов вычисления машины M над словом w_i.

Если для некоторой пары (k, l) машина M допускает слово w_k за l шагов, то A допускает вход $w = \text{Kod}(M)$. Иначе A работает бесконечно – и, следовательно, не принимает w.

Идея описанного моделирования заключалась в том, что если существует $y \in L(M)$, то $y = w_k$ для некоторого натурального k – следовательно, допускающее вычисление машины M над словом y имеет некоторую конечную длину l. Поэтому полный перебор всех вариантов – т. е. всех пар (i, j) на вышеописанном этапе $2'$ машины A – гарантирует, что A примет слово w. □

Теперь покажем, что $(L_{\text{empty}})^{\complement} \notin \mathcal{L}_{\text{R}}$. Это соответствует доказательству *не*существования алгоритма, который для некоторой заданной машины проверял бы, допускает ли она пустой язык. Следствием последнего будет такой факт: правильность программ алгоритмически проверить невозможно. Более того, соответствующая проверка невозможна даже для таких тривиальных задач, как вычисление функции-константы.

Лемма 5.41. $(L_{\text{empty}})^{\complement} \notin \mathcal{L}_{\text{R}}$.

Доказательство. Покажем, что

$$L_{\text{U}} \leq_{\text{m}} (L_{\text{empty}})^{\complement}.$$

Для этого опишем машину Тьюринга A, которая сводит язык L_{U} к языку $(L_{\text{empty}})^{\complement}$ (рис. 5.13). Для каждого входа $x \in \{0, 1, \#\}$ машина A работает следующим образом.

- Если x не представим в виде $\text{Kod}(M)\#w$ ни для каких машины M и слова $w \in (\Sigma_{\text{bool}})^*$, то A пишет на ленту выход $A(x) = \text{Kod}(B_x)$; здесь B_x – некоторая конкретная машина Тьюринга, которая работает над алфавитом Σ_{bool} и принимает пустой язык – т. е. $L(B_x) = \emptyset$.[22]
- Если же $x = \text{Kod}(M)\#w$ для некоторых машины M и слова $w \in (\Sigma_{\text{bool}})^*$, то A генерирует код $\text{Kod}(B_x)$ машины B_x иным способом. А именно, для любого своего входа y машина B_x работает независимо от этого входа[23] – следующим образом.
  - B_x генерирует на своей ленте $x = \text{Kod}(M)\#w$.
    {Слово x конечное, поэтому оно может быть описано с помощью состояний и функции переходов машины B_x.}

[22] Машина A может создать B_x путём установки такого значения функции переходов δ машины B_x: $\delta(q_0, ¢) = (q_{\text{reject}}, ¢, \text{N})$.

[23] Это означает, что B_x допускает либо $\emptyset$, либо $(\Sigma_{\text{bool}})^*$ – см. подробнее ниже.

– B_x пошагово моделирует работу машины M над входом w.
Если M допускает w, то и B_x допускает свой вход y.
Если M отклоняет w, то и B_x отклоняет свой вход y.
Если же M не останавливается на входе w, то и B_x не останавливается на своём входе y – таким образом, B_x не допускает y.

Теперь покажем, что для всех $x \in \{0,1,\#\}^*$

$$x \in L_{\mathrm{U}} \Leftrightarrow A(x) = \mathrm{Kod}(B_x) \in (L_{\mathrm{empty}})^{\complement}.$$

Пусть $x \in L_{\mathrm{U}}$. Тогда $x = \mathrm{Kod}(M)\#w$ для некоторых машины M и слова $w \in L(M)$. В этом случае $L(B_x) = (\Sigma_{\mathrm{bool}})^* \neq \emptyset$, поэтому $\mathrm{Kod}(B_x) \in (L_{\mathrm{empty}})^{\complement}$.

Пусть $x \notin L_{\mathrm{U}}$. Тогда либо x не представимо в виде $\mathrm{Kod}(M')\#z$ ни для каких машины M' и слова $z \in \{0,1\}^*$, либо $x = \mathrm{Kod}(M)\#w$ для некоторых машины M и слова $w \notin L(M)$. В обоих случаях $L(B_x) = \emptyset$ – следовательно, $\mathrm{Kod}(B_x) \notin (L_{\mathrm{empty}})^{\complement}$. □

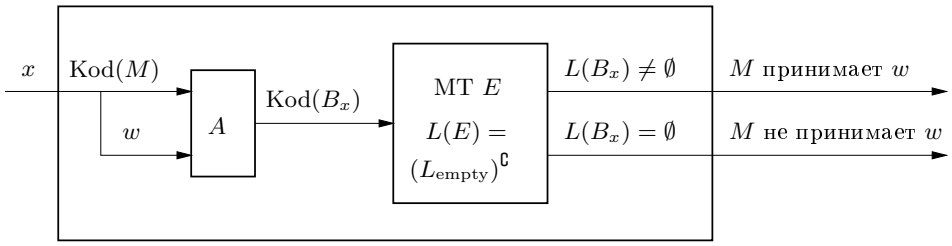


Рис. 5.13.

Следствие 5.42. $L_{\mathrm{empty}} \notin \mathcal{L}_{\mathrm{R}}$.

Доказательство. На основе леммы 5.28 мы получаем, что условие $L_{\mathrm{empty}} \in \mathcal{L}_{\mathrm{R}}$ влечёт $(L_{\mathrm{empty}})^{\complement} \in \mathcal{L}_{\mathrm{R}}$. □

Упражнение 5.43.* Докажите, что следующие языки не принадлежат классу $\mathcal{L}_{\mathrm{RE}}$:

- L_{empty};
- $(L_{\mathrm{H}})^{\complement}$;
- $(L_{\mathrm{U}})^{\complement}$.

Приведённое далее следствие леммы 5.41 формулирует неразрешимость проблемы эквивалентности для машин Тьюринга. Она утверждает, что невозможно создать программу, определяющую, решают ли две данные программы одну и ту же проблему.

Следствие 5.44. *Язык* $L_{\mathrm{EQ}} = \{\mathrm{Kod}(M)\#\mathrm{Kod}(\overline{M}) \mid L(M) = L(\overline{M})\}$ *является неразрешимым – т. е.* $L_{\mathrm{EQ}} \notin \mathcal{L}_{\mathrm{R}}$.

Доказательство. Смысл доказательства прост – поскольку язык L_{empty} можно рассматривать в качестве частного случая языка L_{EQ}. Формально необходимо показать, что

$$L_{\text{empty}} \leq_{\text{m}} L_{\text{EQ}}.$$

Несложно описать такую машину Тьюринга A, которая для данного входа $\text{Kod}(M)$ создаёт выход

$$\text{Kod}(M)\#\text{Kod}(C)$$

– где C является заранее зафиксированной тривиальной машиной Тьюринга с языком $L(C) = \emptyset$. Очевидно, что

$$\text{Kod}(M)\#\text{Kod}(C) \in L_{\text{EQ}} \Leftrightarrow L(M) = L(C) = \emptyset \Leftrightarrow \text{Kod}(M) \in L_{\text{empty}},$$

что завершает доказательство. □

Упражнение 5.45. Докажите, что язык

$$\{\text{Kod}(M)\#\text{Kod}(\overline{M}) \mid L(M) \subseteq L(\overline{M})\}$$

не принадлежит классу $\mathcal{L}_{\text{R}}$.

5.4 Теорема Райса

В предыдущем разделе мы узнали, что проверка программ является труднорешаемой задачей.[24] Для данных программы A и входного слова x неразрешим вопрос, завешает ли A работу над x. Поэтому нельзя проверить, останавливается ли заданная программа для любых входных данных – т. е., согласно введённой выше терминологии, является ли программа алгоритмом. Самая тривиальная из проблем принадлежности – принимает ли данная программа хоть какой-либо вход (т. е. верно ли, что $L(M) \neq \emptyset$ для заданной машины Тьюринга M) – также является неразрешимой.

Всё это приводит нас к предположению о том, что существует очень небольшое число разрешимых проблем, связанных с программами (алгоритмами, машинами Тьюринга). Цель данного раздела – показать, что *все* (в особом, специальном смысле) нетривиальные проблемы о программах являются *неразрешимыми*. В следующем определении объясняется, что именно обозначает термин «нетривиальный» в этом контексте.

Определение 5.46. *Язык*

$$L \subseteq \{\text{Kod}(M) \mid M \text{ – } \textit{некоторая } \text{МТ}\}$$

является **семантически нетривиальной проблемой принадлежности о машинах Тьюринга**, *если выполнены следующие условия:*

(a) Существует МТ M_1, *такая что* $\text{Kod}(M_1) \in L$ – *т. е.* $L \neq \emptyset$.
(b) Существует МТ M_2, *такая что* $\text{Kod}(M_2) \notin L$ – *т. е.* L *не содержит кодов всех машин Тьюринга.*
(c) Для любых машин Тьюринга A *и* B *из равенства* $L(A) = L(B)$ *следует, что*

$$\text{Kod}(A) \in L \iff \text{Kod}(B) \in L.$$

[24] В оригинале – «hard problem». Это сочетание можно считать термином – который будет определён (точнее – описан) в следующей главе. Однако в данном случае более важным является другое утверждение – см. далее. *(Прим. перев.)*

Прежде чем приступить к доказательству неразрешимости семантически нетривиальных проблем принадлежности, нам необходимо рассмотреть следующий язык

$$L_{\mathrm{H},\lambda} = \{\mathrm{Kod}(M) \mid M \text{ завершает работу на } \lambda\}$$

в качестве специального варианта проблемы останова.

Лемма 5.47.

$$L_{\mathrm{H},\lambda} \notin \mathcal{L}_{\mathrm{R}}.$$

Доказательство. Покажем, что

$$L_{\mathrm{H}} \leq_{\mathrm{m}} L_{\mathrm{H},\lambda}.$$

Некоторая машина Тьюринга A может свести L_{H} к $L_{\mathrm{H},\lambda}$ следующим образом. Для каждого входа x, который не представим в виде $\mathrm{Kod}(M)\#w$, A генерирует МТ H_x, которая не заканчивает работу ни для какого входа.

Если же $x = \mathrm{Kod}(M)\#w$ для некоторых машины Тьюринга M и входного слова w, то A генерирует код $\mathrm{Kod}(H_x)$ машины H_x, которая работает следующим образом:

1. Независимо от своего собственного входа МТ H_x генерирует на ленте слово $x = \mathrm{Kod}(M)\#w$.
2. H_x моделирует работу машины M над словом w. Если при этом M останавливается, то H_x также останавливается и принимает свой вход. Если же при этом вычисление M над w бесконечно, то и H_x не останавливается.[25]

Очевидно, что для каждого $x \in \{0,1,\#\}^*$ выполнено следующее:

$$\begin{aligned} x \in L_{\mathrm{H}} &\Leftrightarrow x = \mathrm{Kod}(M)\#w \text{ и } M \text{ завершает работу на } w \\ &\Leftrightarrow H_x \text{ всегда останавливается (для всех возможных входов)} \\ &\Leftrightarrow H_x \text{ останавливается на } \lambda \\ &\Leftrightarrow \mathrm{Kod}(H_x) \in L_{\mathrm{H},\lambda} \end{aligned}$$

□

Теорема 5.48 (Райса).* *Каждая семантически нетривиальная проблема принадлежности о машинах Тьюринга является неразрешимой.*

Доказательство. Пусть L – произвольная семантически нетривиальная проблема принадлежности о машинах Тьюринга. Покажем, что

$$\text{либо } \; L_{\mathrm{H},\lambda} \leq_{\mathrm{m}} L, \;\; \text{либо } \; L_{\mathrm{H},\lambda} \leq_{\mathrm{m}} L^{\complement}.$$

Пусть $M_\emptyset$ – некоторая МТ, такая что $L(M_\emptyset) = \emptyset$. Будем различать две возможности – в зависимости от того, является ли $\mathrm{Kod}(M_\emptyset)$ словом языка L.

[25] В этом случае либо H_x всегда останавливается – если машина M останавливается на входе w, либо H_x не останавливается ни для какого входа – если M не останавливается на входе w.

I. Пусть $\mathrm{Kod}(M_\emptyset) \in L$. В этом случае покажем, что

$$L_{\mathrm{H},\lambda} \leq_{\mathrm{m}} L^{\complement}.$$

Согласно определению 5.46 (b), существует некоторая машина Тьюринга $\overline{M}$, такая что $\mathrm{Kod}(\overline{M}) \notin L$.

Опишем работу машины S (рис. 5.14) которая сводит $L_{\mathrm{H},\lambda}$ к $L^{\complement}$. Для каждого входа $x \in (\Sigma_{\mathrm{bool}})^*$ машина S вычисляет:

- либо $S(x) = \mathrm{Kod}(M')$ с соответствующим языком $L(M') = L(M_\emptyset) = \emptyset$ (т. е. $\mathrm{Kod}(M') \notin L^{\complement}$) – в случае $x \notin L_{\mathrm{H},\lambda}$;
- либо $S(x) = \mathrm{Kod}(M')$ с соответствующим языком $L(M') = L(\overline{M})$ (т. е. $\mathrm{Kod}(M') \in L^{\complement}$) – в случае $x \in L_{\mathrm{H},\lambda}$.

Итак, мы видим, что идея метода сводимости в значительной степени основана на семантической нетривиальности L. Машина S выполняет вычисления следующим образом (рис. 5.14):

Вход: Некоторое слово $x \in (\Sigma_{\mathrm{bool}})^*$.

Этап 1. S проверяет, верно ли, что $x = \mathrm{Kod}(M)$ для некоторой МТ M. Если x не является кодом никакой машины Тьюринга, то машина S в качестве своего выхода записывает на ленту слово $S(x) = \mathrm{Kod}(M_\emptyset)$.

Этап 2. Если же $x = \mathrm{Kod}(M)$ для некоторой МТ M, то S генерирует код $\mathrm{Kod}(M')$ машины M', которая работает следующим образом.

(a) Входным алфавитом машины M' является $\Sigma_{\overline{M}}$ (входной алфавит машины $\overline{M}$, у которой $\mathrm{Kod}(\overline{M}) \notin L$, т. е. $\mathrm{Kod}(\overline{M}) \in L^{\complement}$).

(b) Для каждого входа $y \in (\Sigma_{\overline{M}})^*$ машина M' генерирует на ленте слово $x = \mathrm{Kod}(M)$ справа от y (т. е. слово y при этом не перезаписывается) – после чего моделирует работу машины M над входом λ.

Если M не останавливается на слове λ (т. е. $\mathrm{Kod}(M) \notin L_{\mathrm{H},\lambda}$), то M' также не останавливается на y, поэтому $y \notin L(M')$.

{Поскольку моделирование вычисления машины M над словом λ с помощью M' выполняется независимо от входа y машины M', т. е. $L(M') = \emptyset = L(M_\emptyset)$, то выполнено условие $\mathrm{Kod}(M') \in L$ (т. е. $\mathrm{Kod}(M') \notin L^{\complement}$).}

Если M завершает свою работу над словом λ (т. е. если $\mathrm{Kod}(M) \in L_{\mathrm{H},\lambda}$), то M' генерирует на своей ленте код $\mathrm{Kod}(\overline{M})$ машины $\overline{M}$. После этого M' моделирует работу машины $\overline{M}$ над своим собственным входом $y \in (\Sigma_{\overline{M}})^*$. M' допускает y тогда и только тогда, когда $\overline{M}$ допускает y.

{Следовательно, $L(M') = L(\overline{M})$, поэтому $\mathrm{Kod}(M') \notin L$ – т. е. $\mathrm{Kod}(M') \in L^{\complement}$.}

Мы видим, что

$$x \in L_{\mathrm{H},\lambda} \Leftrightarrow S(x) \in L^{\complement}$$

для всех $x \in (\Sigma_{\mathrm{bool}})^*$ – следовательно, $L_{\mathrm{H},\lambda} \leq_{\mathrm{m}} L^{\complement}$.

II. Пусть $\mathrm{Kod}(M_\emptyset) \notin L$.

Согласно определению 5.46 (a), существует некоторая машина Тьюринга $\tilde{M}$, такая что $\mathrm{Kod}(\tilde{M}) \in L$. На основе этого можно доказать, что $L_{\mathrm{H},\lambda} \leq_{\mathrm{m}} L$ – тем же способом, которым в части I мы доказали, что $L_{\mathrm{H},\lambda} \leq_{\mathrm{m}} L^{\complement}$. В этом доказательстве машина $\tilde{M}$ выполняет роль машины $\overline{M}$. □

Упражнение 5.49. Приведите полное доказательство того, что $L_{\mathrm{H},\lambda} \leq_{\mathrm{m}} L$ – для случая II, в котором $\mathrm{Kod}(M_\emptyset) \notin L$.

У теоремы Райса имеется такое следствие. Пусть L – произвольный рекурсивный язык, и пусть

$$\mathrm{Kod}_L = \{\mathrm{Kod}(M) \mid M - \text{некоторая МТ, такая что } L(M) = L\}$$

– язык, содержащий коды всех машин Тьюринга, которые допускают язык L. Поскольку L является рекурсивным, $\mathrm{Kod}_L \neq \emptyset$. Очевидно, что существуют машины Тьюринга, коды которых не принадлежат множеству Kod_L (т. е. выполнено определение 5.46 (b)), кроме того, Kod_L удовлетворяет определению 5.46 (c). Итак, Kod_L является семантически нетривиальной проблемой о машинах Тьюринга – и согласно теореме Райса мы получаем, что $\mathrm{Kod}_L \notin \mathcal{L}_\mathrm{R}$.

Интерпретация последнего результата заключается в том, что невозможно проверить, является ли данная программа корректным алгоритмическим решением заданной проблемы. Итак, проверка правильности программы – труднорешаемая задача, а поэтому хорошо структурированное, модульное проектирование программ имеет огромное значение для разработки надежных программных систем.

Упражнение 5.50. Докажите, что для любых алфавита Σ и языка $L \subseteq \Sigma^*$ выполнено следующее:

$$L, L^\complement \in \mathcal{L}_\mathrm{RE} \Leftrightarrow L \in \mathcal{L}_\mathrm{R}.$$

5.5 Проблема соответствий Поста

В предыдущих разделах мы показали, что практически все проблемы о машинах Тьюринга (о программах) являются неразрешимыми. Поэтому возникает интересный вопрос – существуют ли неразрешимые проблемы, не относящиеся к машинам Тьюринга. Ответ на этот вопрос положителен, и цель данного раздела – показать, как можно применить метод сводимости для преобразования результатов о неразрешимости машин Тьюринга к миру игр.[26]

Рассмотрим игру в домино со следующими правилами. Имеется набор фишек домино конечного числа типов (рис. 5.15) – каждый из типов представляет собой пару (x, y) слов x и y над фиксированным алфавитом Σ. Для каждого типа (x, y) имеется бесконечное множество фишек (x, y). Возникает вопрос: можно ли поместить некоторое количество фишек рядом – таким образом, чтобы верхний текст (верхнее слово, оно определяется как конкатенация первых элементов фишек домино) был бы таким же, как и нижний (определяемый аналогично по вторым элементам фишек). Проиллюстрируем это на следующем примере.

Пусть

$$s_1 = (1, \#0),\ s_2 = (0, 01),\ s_3 = (\#0, 0),\ s_4 = (01\#, \#)$$

– допустимые типы фишек домино над алфавитом $\{0, 1, \#\}$. Графическое представление фишек s_1, s_2, s_3 и s_4 приведено на рис. 5.16.

Эта игра имеет решение, которым является последовательность s_2, s_1, s_3, s_2, s_4. Эта последовательность определяет слово 01#0001#, описанное на рисунках 5.17 и 5.18 двумя разными способами.

[26] По-видимому, здесь и всюду в дальнейшем вместо термина «игры» точнее употреблять «головоломки» – однако мы при переводе всюду оставляем вариант автора. *(Прим. перев.)*

Однако для игры

$$s_1 = (00, 001),\ s_2 = (0, 001),\ s_3 = (1, 11)$$

решения не существует – потому что все вторые элементы (слова) фишек домино длиннее, чем соответствующие им первые элементы.

Такая игра называется проблемой соответствий Поста.

Определение 5.51. *Пусть Σ – некоторый алфавит.* **Частный случай**[27] **проблемы соответствий Поста** *над алфавитом Σ – это пара (A, B), где*

$$A = w_1,\ \ldots,\ w_k\,,\quad B = x_1,\ \ldots,\ x_k$$

для некоторых натурального k и слов $w_i, x_i \in \Sigma^*$ при $i = 1, \ldots, k$. Для каждого $i \in \{1, \ldots, k\}$ пара (w_i, x_i) называется **фишкой домино**.

Будем говорить, что частный случай (A, B) проблемы соответствия Поста имеет **решение**, *если существуют некоторые натуральное k и натуральные $i_1, i_2, \ldots, i_m$, такие что*

$$w_{i_1} w_{i_2} \ldots w_{i_m} = x_{i_1} x_{i_2} \ldots x_{i_m}.$$

Проблема соответствий Поста (ПСП[28]**)** *заключается в том, чтобы определить, имеет ли решение данный частный случай ПСП.*

Лемма 5.52. *Если некоторый частный случай ПСП имеет решение, то он имеет бесконечно много решений.*

Доказательство. Если

$$i_1,\ i_2,\ \ldots,\ i_k$$

– некоторое решение частного случая ПСП (A, B), то

$$(i_1,\ i_2,\ \ldots,\ i_k)^j$$

– решение проблемы (A, B) для каждого натурального j. □

Упражнение 5.53. Определите ПСП как проблему принадлежности $(L_{\mathrm{PCP}}, \Sigma_{\mathrm{bool}})$ для языка $L_{\mathrm{PCP}} \subseteq (\Sigma_{\mathrm{bool}})^*$. Докажите, что $L_{\mathrm{PCP}} \in \mathcal{L}_{\mathrm{RE}}$.

Упражнение 5.54. Докажите, что частный случай ПСП $((10, 011, 101), (101, 11, 011))$ не имеет решений.

Упражнение 5.55. Имеет ли решение частный случай ПСП $((1, 1110111, 101), (111, 1110, 01))$?

---

[27] В русской литературе иногда применяется термин «пример» (проблемы соответствий Поста). Употребляемый нами термин более удачен – поскольку согласуется с остальной терминологией этой книги. *(Прим. перев.)*

[28] Английская аббревиатура – PCP. Мы будем употреблять оба варианта, при этом английский – в формулах.

Сто́ит также отметить, что в теоретической информатике эта аббревиатура (PCP) употребляется для двух различных понятий – данного, а также «probabilistically checkable proofs». Однако при чтении этой книги проблем, связанных с этой неоднозначностью, возникнуть не может. *(Прим. перев.)*

Наша цель – показать, что ПСП неразрешима. Важное значение этого доказательства о неразрешимости состоит в том, что описанный нами вариант игры в домино является мощным инструментом – достаточным для моделирования вычислений машины Тьюринга. Однако в связи с техническими ограничениями мы введём изменённую (упрощённую) версию ПСП – такую, в которой первая фишка домино задана заранее.

Определение 5.56. *Пусть Σ – некоторый алфавит.* **Частный случай модифицированной проблемы соответствий Поста (МПСП[29])** *над алфавитом Σ – это пара (C, D), где*

$$C = u_1, \ \ldots, \ u_k\,, \quad D = v_1, \ \ldots, \ v_k$$

для некоторых натурального k и слов $u_i, v_i \in \Sigma^*$ $(i = 1, \ldots, k)$.

Будем говорить, что частный случай (C, D) модифицированной проблемы соответствий Поста имеет решение, если существуют некоторое натуральное m и m натуральных $j_1, j_2, \ldots, j_m$, такие что

$$u_1 u_{j_1} u_{j_2} \ldots u_{j_m} = v_1 v_{j_1} v_{j_2} \ldots v_{j_m}.$$

Модифицированная проблема соответствий Поста *заключается в определении того, имеет ли решение некоторый заданный частный случай МПСП над алфавитом Σ.*

Рассмотрим следующий частный случай ПСП (A, B): пусть

$$A = 0, 11, 1, \quad B = 001, 1, 11,$$

$$\text{т. е.} \quad s_1 = (0, 001)\,,\ s_2 = (11, 1)\,,\ s_3 = (1, 11).$$

Очевидно, что $s_2 s_3$ – решение этого частного случая ПСП. Но если мы рассмотрим (A, B) в качестве частного случая МПСП, то получим, что (A, B) не имеет решений. Таким образом, возможна разница между парой (A, B), рассматриваемой как частный случай ПСП и как частный случай МПСП.

Упражнение 5.57. Докажите, что частный случай МПСП $((0, 11, 1), (001, 1, 11))$ не имеет решений.

Лемма 5.58. *Если ПСП разрешима, то и МПСП также разрешима.*

Доказательство. Докажем это утверждение с помощью метода сводимости. Пусть (A, B) – частный случай МПСП. Опишем конкретный частный случай ПСП (C, D) – такой что

$$\text{МПСП } (A, B) \text{ имеет решение} \iff \text{ПСП } (C, D) \text{ имеет решение.}$$

Пусть

$$A = w_1, \ \ldots, \ w_k, \quad B = x_1, \ \ldots, \ x_k,$$

где для алфавита Σ и натурального k слова $w_i, x_i \in \Sigma^*$ при $i = 1, \ldots, k$. Пусть $\$, ¢ \notin \Sigma$. Сконструируем частный случай ПСП (C, D) над алфавитом $\Sigma_1 = \Sigma \cup \{¢, \$\}$.

[29] Английская аббревиатура – MPCP.

Во-первых, следующим образом определим два гомоморфизма, h_{L} и h_{R}, действующих из Σ^* в Σ_1^*. Для каждого символа $a \in \Sigma$ положим

$$h_{\mathrm{L}}(a) = ¢a \quad \text{и} \quad h_{\mathrm{R}}(a) = a¢.$$

Мы видим, что h_{L} вставляет символ ¢ слева от любого символа, а h_{R} вставляет этот же символ ¢ справа от любого символа. Например, для слова 0110 мы получаем

$$h_{\mathrm{L}}(0110) = ¢0¢1¢1¢0 \quad \text{и} \quad h_{\mathrm{R}}(0110) = 0¢1¢1¢0¢.$$

Положим

$$C = y_1,\, y_2,\, \ldots,\, y_{k+2} \quad \text{и} \quad D = z_1,\, z_2,\, \ldots,\, z_{k+2},$$

где

$$\begin{array}{rr}
y_1 = ¢h_{\mathrm{R}}(w_1) & z_1 = h_{\mathrm{L}}(x_1) \\
y_2 = h_{\mathrm{R}}(w_1) & z_2 = h_{\mathrm{L}}(x_1) \\
y_3 = h_{\mathrm{R}}(w_2) & z_3 = h_{\mathrm{L}}(x_2) \\
\vdots & \vdots \\
y_{i+1} = h_{\mathrm{R}}(w_i) & z_{i+1} = h_{\mathrm{L}}(x_i) \\
\vdots & \vdots \\
y_{k+1} = h_{\mathrm{R}}(w_k) & z_{k+1} = h_{\mathrm{L}}(x_k) \\
y_{k+2} = \$ & z_{k+2} = ¢\$.
\end{array}$$

Например, для частного случая МПСП

$$((0, 11, 1),\, (001, 1, 11))$$

мы создаём такой частный случай ПСП:

$$((¢0¢, 0¢, 1¢1¢, 1¢, \$),\, (¢0¢0¢1, ¢0¢0¢1, ¢1, ¢1¢1, ¢\$)).$$

Очевидно, что существует некоторый алгоритм (некоторая МТ), который вышеописанным способом строит пару (C, D) на основе любой заданной пары (A, B). Остаётся показать, что либо обе эти задачи, МПСП (A, B) и ПСП (C, D), имеют решение – либо наоборот, ни одна из них решения не имеет.

- Сначала докажем, что каждое решение МПСП (A, B) определяет решение ПСП (C, D). Пусть $i_1, i_2, \ldots, i_m$ – некоторое решение МПСП (A, B). Тогда

  $$u = w_1 w_{i_1} w_{i_2} \ldots w_{i_m} = x_1 x_{i_1} x_{i_2} \ldots x_{i_m}.$$

  При этом последовательность индексов

  $$2,\, i_1 + 1,\, i_2 + 1,\, \ldots,\, i_m + 1,$$

  для ПСП(C, D) соответствует используемым морфизмам: h_{R} для записи $w_1 w_{i_1} w_{i_2} \ldots w_{i_m}$ и h_{L} для $x_1 x_{i_1} x_{i_2} \ldots x_{i_m}$. Поэтому

  $$¢h_{\mathrm{R}}(u) = ¢y_2 y_{i_1+1} \ldots y_{i_m+1} = z_2 z_{i_1+1} \ldots z_{i_m+1}¢ = h_{\mathrm{L}}(u)¢.$$

  Следовательно, разница между

$$h_{\mathrm{R}}(u) = y_2 y_{i_1+1} \dots y_{i_m+1} \quad \text{и} \quad h_{\mathrm{L}}(u) = z_2 z_{i_1+1} \dots z_{i_m+1}$$

заключается только в первом и последнем символах ¢.[30] Согласно методу построения пары (C, D), мы получаем $y_1 = ¢y_2$ и $z_1 = z_2$. Итак, мы можем заменить первый индекс 2 на 1. Таким образом мы получаем последовательность

$$1,\ i_1 + 1,\ i_2 + 1,\ \dots,\ i_m + 1,$$

для которой

$$y_1 y_{i_1+1} y_{i_2+1} \dots y_{i_m+1} = z_1 z_{i_1+1} z_{i_2+1} \dots z_{i_m+1} ¢.$$

Теперь единственным отличием между сформированными левой и правой частями является дополнительный символ ¢ в конце вышеприведённого текста. И чтобы получить решение для ПСП (C, D), мы просто добавляем $(k+2)$-ю фишку $(¢, ¢\$)$. Итак,

$$y_1 y_{i_1+1} y_{i_2+1} \dots y_{i_m+1} y_{k+2} = z_1 z_{i_1+1} z_{i_2+1} \dots z_{i_m+1} z_{k+2},$$

и в результате последовательность

$$1,\ i_1 + 1,\ i_2 + 1,\ \dots,\ i_m + 1,\ k + 2$$

определяет решение ПСП (C, D).

- Теперь мы должны показать, что если существует решение проблемы ПСП (C, D), то существует решение и для исходного частного случая МПСП (A, B). Для начала отметим, что все слова z_i начинаются символом ¢, а y_1 – единственное слово, начинающееся символом ¢, среди всех y_i. Следовательно, каждое решение для ПСП (C, D) должно начинаться с первой фишки. С другой стороны, только та фишка, у которой оба слова заканчиваются одним и тем же символом, является фишкой $(k+2)$-го типа. Следовательно, каждое решение для пары ПСП (C, D) должно кончаться индексом $k + 2$.
  Таким образом, пусть

  $$1,\ j_1,\ j_2,\ \dots,\ j_m,\ k + 2$$

  – некоторое решение для ПСП (C, D). Мы утверждаем, что

  $$1, j_1 - 1, j_2 - 1, \dots, j_m - 1$$

  – решение для МПСП (C, D). Последнее выполняется вследствие того, что:
  (a) удаление символов ¢ и \$ из слова $y_1 y_{j_1} y_{j_2} \dots y_{j_m} y_{k+2}$ даёт слово $w_1 w_{j_1-1} w_{j_2-1} \dots w_{j_m-1}$,
  (b) удаление символов ¢ и \$ из слова $z_1 z_{j_1} z_{j_2} \dots z_{j_m} z_{k+2}$ даёт слово $x_1 x_{j_1-1} x_{j_2-1} \dots x_{j_m-1} x_{k+2}$.
  Поскольку $1, j_1, \dots j_m, k+2$ – решение ПСП (C, D),

  $$y_1, y_{j_1} y_{j_2} \dots y_{j_m} y_{k+2} = z_1 z_{j_1} z_{j_2} \dots z_{j_m} z_{k+2}.$$

  Вместе с условиями (a) и (b) последнее означает, что

  $$w_1 w_{j_1-1} w_{j_2-1} \dots w_{j_m-1} = x_1 x_{j_1-1} x_{j_2-1} \dots x_{j_m-1}.$$

  Следовательно, последовательность $1, j_1 - 1, j_2 - 1, \dots, j_m - 1$ определяет решение для МПСП (A, B).

[30] Первый символ входит в $h_{\mathrm{L}}(u)$, но не входит в $h_{\mathrm{R}}(u)$; последний – наоборот.

□

Упражнение 5.59. Докажите, что разрешимость МПСП влечёт разрешимость ПСП.

Теперь мы покажем неразрешимость МПСП – а для этого продемонстрируем, как рассматриваемый нами вариант игры в домино может быть использован для моделирования вычислений машины Тьюринга.

Лемма 5.60.* *Разрешимость МПСП влечёт разрешимость* L_{U}.

Доказательство. Пусть $x \in \{0,1,\#\}^*$. Создадим частный случай МПСП (A,B), такой что

$$x \in L_U \iff \text{МПСП}(A,B) \text{ имеет решение.}$$

Если слово x не представимо в виде $\mathrm{Kod}(M)\#w$ ни для каких МТ M и слова $w \in \{0,1\}^*$ (т. е. $x \notin L_{\mathrm{U}}$), то мы полагаем $A = 0$ и $B = 1$. Очевидно, что игра в домино с фишками единственного типа $(0,1)$ не имеет решения.

Далее, пусть $x = \mathrm{Kod}(M)\#w$ для некоторых машины Тьюринга $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{accept}}, q_{\mathrm{reject}})$ и слова $w \in \Sigma^*$. Без ограничения общности мы можем предполагать, что M перемещает свою головку на каждом шаге вычисления. Опишем МПСП (C,D) в виде следующих четырёх шагов, каждый из которых будет определять группу фишек домино со специальным значением.

Идея нашего построения состоит в использовании:

- B-слов (нижней части) для моделирования вычислений M на слове w;
- A-слов (верхней части) – для того, чтобы следовать этому моделированию с задержкой на одну конфигурацию.

Благодаря такой задержке можно использовать фишки так, что верхняя часть при этом соответствует аргументам (состоянию и читаемому символу), а нижняя часть – результатам, полученным для этих аргументов после применения функции переходов δ моделируемой машины. Используя описываемые далее специальные фишки, «задерживающаяся» вверхняя часть догоняет нижнюю тогда и только тогда, когда вычисление заканчивается в состоянии q_{accept}.

Пусть $\#$ – некоторый новый символ, не содержащийся в алфавите Γ.

1. Первая группа содержит только один тип фишек:

   $$(\#, \#q_0 ¢ w \#).$$

   Эта фишка начинает моделирование вычислений машины M над словом w. Нижняя часть соответствует инициальной конфигурации M на w, вследствие чего у верхней части образуется «задержка» $q_0 ¢ w \#$.
2. Вторая группа содержит следующие типы фишек:

   $$(X, X) \text{ для всех } X \in \Gamma;$$
   $$(\#, \#).$$

   Эта группа (состоящая из $|\Gamma| + 1$ типов фишек) используется для посимвольного копирования из нижнего текста в верхний соответствующих частей конфигурации – при этом верхний текст не изменяется на следующем шаге вычисления.

3. Третья группа используется для воспроизведения шага вычислений машины M. Для всех $q \in Q - \{q_{\text{accept}}\}$, $p \in Q$, $X, Y, Z \in \Gamma$ мы выбираем следующие типы фишек:

$$\begin{aligned} (qX, Yp), &\quad \text{если } \delta(q, X) = (p, Y, \text{R}); \\ (ZqX, pZY), &\quad \text{если } \delta(q, X) = (p, Y, \text{L}); \\ (q\#, Yp\#), &\quad \text{если } \delta(q, \sqcup) = (p, Y, \text{R}); \\ (Zq\#, pZY\#), &\quad \text{если } \delta(q, \sqcup) = (p, Y, \text{L}). \end{aligned}$$

   Эти фишки используются для посимвольного копирования конфигураций из нижнего текста в верхний, а также генерацию следующей конфигурации в нижней части.

4. С помощью фишек четвёртой группы верхний текст достигает размера нижнего – если вычисление машины M над словом w заканчивается в состоянии q_{accept}. Используя следующие фишки, состояние q_{accept} может «стереть» символы множества $\Gamma \cup \{¢\}$. Итак, для всех $X, Y \in \Gamma \cup \{¢\}$ мы выбираем следующие типы фишек:

$$(Xq_{\text{accept}}Y, q_{\text{accept}});$$
$$(Xq_{\text{accept}}, q_{\text{accept}});$$
$$(q_{\text{accept}}Y, q_{\text{accept}}).$$

   Если состояние q_{accept} в нижнем тексте «поглощает» все символы ленты, то задержка верхней части «уменьшается» до слова $q_{\text{accept}}\#$.
   А следующие фишки

$$(q_{\text{accept}}\#\#, \#)$$

   позволяют нам окончательно сбалансировать длины верхней и нижней частей.

Проиллюстрируем создание частного случая МПСП (A, B) для машины Тьюринга

$$M = (\{q_0, q_1, q_{\text{accept}}, q_{\text{reject}}\}, \{0, 1\}, \{¢, 0, 1, \sqcup\}, \delta_M, q_{\text{accept}}, q_{\text{reject}}),$$

представленной на рис. 5.19. Для входного слова $w = 01$ мы создаём следующие типы фишек домино.

1. Первая группа:

$$(\#, \#q_0¢01\#).$$

2. Вторая группа:

$$(0, 0), (1, 1), (¢, ¢), (\$, \$), (\#, \#).$$

3. Третья группа:

$$(q_0 1, 1q_0), (q_0 0, 0q_1), (q_0¢, ¢q_1),$$
$$(1q_0\#, q_{\text{accept}}1\#), (0q_0\#, q_{\text{accept}}0\#)$$
$$(q_1 1, 1q_1), (q_1 0, 0q_0), (1q_1\#, q_{\text{reject}}1\#), (0q_1\#, q_{\text{reject}}0\#).$$

4. Четвёртая группа:

$$(0q_{\text{accept}}0, q_{\text{accept}}), (1q_{\text{accept}}1, q_{\text{accept}}), (1q_{\text{accept}}0, q_{\text{accept}}),$$
$$(0q_{\text{accept}}1, q_{\text{accept}}), (0q_{\text{accept}}, q_{\text{accept}}), (1q_{\text{accept}}, q_{\text{accept}}),$$
$$(q_{\text{accept}}0, q_{\text{accept}}), (q_{\text{accept}}1, q_{\text{accept}}),$$
$$(¢q_{\text{accept}}, q_{\text{accept}}), (q_{\text{accept}}¢, q_{\text{accept}}),$$
$$(¢q_{\text{accept}}a, q_{\text{accept}}), (aq_{\text{accept}}¢, q_{\text{accept}}) \text{ для всех } a \in \{0, 1\}$$
$$(q_{\text{accept}}\#\#, \#).$$

Начало моделирования вычисления машины M над словом 01 представлено на рис. 5.20.

Рис. 5.21 показывает этап сокращения – в котором верхний текст «догоняет» нижний.

Очевидно, что конкретный частный случай МПСП (A, B) может быть алгоритмически создан на основе заданного кода машины Тьюринга M. Нам осталось доказать, что

$$M \text{ допускает } w \iff \text{МПСП}(A, B) \text{ имеет решение.}$$

Не вдаваясь в излишние технические детали, приведём только неформальные аргументы, обосновывающие эту эквивалентность.

Если $w \in L(M)$, то существует допускающее вычисление машины M над словом w – будем использовать это вычисление для построения решения частного случая МПСП (A, B). Решение начинается с фишки первой группы $(\#, \#q_0 ¢ w \#)$. Затем мы используем фишки второй группы для копирования символов нижней части в верхнюю – в том случае, если эти символы остаются неизменными на следующем шаге вычисления. Фишки третьей группы используются для установления следующей конфигурации нижней части. После каждого символа # текст внизу всегда на одну конфигурацию длиннее, чем наверху.[31] Если нижний текст содержит завершённое вычисление (заканчивающееся в состянии q_{accept}) – то фишки четвёртой группы позволяют верхнему тексту «догнать» нижний. Таким образом мы получаем решение для МПСП (A, B).

Пусть $w \notin L(M)$. Каждое потенциальное решение МПСП (A, B) должно начинаться с фишки первой группы $(\#, \#q_0 ¢ w \#)$. Поскольку фишки второй и третьей групп допускают только такие изменения первой конфигурации, которые относятся к некоторому шагу вычислений машины M, символ q_{accept} никогда не встретится в нижней части. Поэтому нижний текст всегда остаётся длиннее верхнего, и поэтому МПСП (A, B) не имеет решений. □

Упражнение 5.61. Напишите МПСП (A, B) для машины Тьюринга, приведённой на рис. 4.6.

Упражнение 5.62. Пусть (A, B) является частным случаем МПСП для некоторых машины Тьюринга M и слова w – согласно построению, представленному в доказательстве леммы 5.60. Докажите с помощью индукции сформулированные далее утверждения.

Если $w \in L(M)$, то существует некоторая последовательность индексов, такая что:

- нижний текст содержит полное вычисление машины M над словом w – где соседние конфигурации разделены символом #;
- верхний текст, являющийся собственным префиксом нижнего, содержит все вычисления кроме последней конфигурации.

Теорема 5.63. *ПСП неразрешима.*

Доказательство. Согласно лемме 5.60 получаем, что МПСП неразрешима, а лемма 5.58 утверждает, что ПСП по крайней мере столь же сложна, как МПСП. □

Упражнение 5.64.* Рассмотрим упрощённую версию ПСП – в которой все фишки домино заданы над однобуквенным алфавитом $\{0\}$. Разрешима ли такая проблема?

[31] Напомним, что текст верхней части всегда является префиксом текста нижней части.

5.6 Метод сложности по Колмогорову

В разделе 5.2 мы использовали метод диагонализации для получения первой алгоритмически неразрешимой проблемы L_{diag}. Доказав нерекурсивность L_{diag}, мы получили базу для построения теории вычислимости. Мы использовали эту базу вместе с методом сводимости в разделах 5.3, 5.4 и 5.5 – для доказательства неразрешимости некоторых проблем принадлежности. Цель этого раздела – предложить альтернативный способ построения теории вычислимости. Не предполагая существования неразрешимых проблем (т. е. без использования каких-либо результатов о неразрешимости, доказанных в предыдущих разделах), мы используем теорию сложности по Колмогорову – для доказательства несуществования алгоритма, вычисляющего сложность по Колмогорову $K(x)$ некоторого заданного слова $x \in (\Sigma_{\text{bool}})^*$. Получив этот результ о неразрешимости в качестве альтернативной базы[32], мы снова демонстрируем использование метода сводимости – доказывая с его помощью неразрешимость проблемы останова. Получив этот результат, мы можем пойти тем же путём, что и в разделах 5.3 и 5.4 – для доказательства неразрешимости некоторых конкретных проблем принадлежности.

Теорема 5.65. *Проблема вычисления сложности по Колмогорову $K(x)$ для любого заданного слова $x \in (\Sigma_{\text{bool}})^*$ алгоритмически неразрешима.*[33]

Доказательство. Докажем теорему 5.65 методом от противного. Пусть A – некоторый алгоритм, вычисляющий $K(x)$ для некоторого заданного слова $x \in (\Sigma_{\text{bool}})^*$. Пусть x_n – первое слово относительно канонического порядка над алфавитом Σ_{bool}, такое что

$$K(x_n) \geq n.$$

На основе алгоритма A для любого натурального n опишем алгоритм B_n, который:

- использует A как подпрограмму;
- вычисляет x_n для входа, являющегося пустым словом λ.

B_n работает следующим образом.

```
B_n:    begin x := λ;
          вычислить K(x) с помощью алгоритма A;
          while K(x) < n do
            begin
              x := следующее слово после x
                     относительно канонического порядка;
              вычислить K(x) с помощью алгоритма A
            end;
          output(x)
        end
```

Очевидно, что для каждого натурального n алгоритм B_n вычисляет слово x_n. Отметим, что все алгоритмы B_n идентичны – за исключением значения n. Пусть c –

[32] Вместо языка диагонализации.

[33] Если рассматривать K как функцию, действующую из $(\Sigma_{\text{bool}})^*$ в $(\Sigma_{\text{bool}})^*$ – вместо функции, действующей из $(\Sigma_{\text{bool}})^*$ в $\mathbb{N}_0$ (т. е. если $K(x)$ будет двоичным представлением сложности по Колмогорову слова x) – то утверждение теоремы 5.65 можно переформулировать следующим образом: функция K является нерекурсивной.

длина машиного кода программы B_n (не включая n). Тогда для всех $n \in \mathbb{N}_0$ двоичная длина программы B_n не превосходит значения

$$\lceil \log_2(n+1) \rceil + c$$

– для некоторой константы c, не зависящей от n.[34] Поскольку программа B_n генерирует слово x_n, то для всех натуральных n выполнено неравенство

$$K(x_n) \leq \lceil \log_2(n+1] + c.$$

Но, согласно определению слов x_n, для всех $n \in \mathbb{N}$ выполнено неравенство

$$K(x_n) \geq n.$$

При этом неравенство

$$\lceil \log_2(n+1) \rceil + c \geq K(x_n) \geq n$$

может выполняться только для конечного множества натуральных чисел – следовательно, мы получаем противоречие со сделанным предположением о существовании алгоритма A, вычисляющего значение $K(x)$ для любого слова x. □

Упражнение 5.66. Докажите, что для любого заданного натурального n задача вычисления первого слова x_n, такого что $K(x_n) \geq n$, алгоритмически неразрешима.

Теорему 5.65 можно усилить – доказав при этом неразрешимость основных используемых нами языков, таких как L_{U}, L_{H}, L_{empty} и др. Для этого мы сводим проблему вычисления функции K к проблеме останова.

А следующая лемма представляет собой альтернативное доказательство факта $L_{\mathrm{H}} \notin \mathcal{L}_{\mathrm{R}}$ из теоремы 5.38.

Лемма 5.67. *Если $L_{\mathrm{H}} \in \mathcal{L}_{\mathrm{R}}$, то существует алгоритм, вычисляющий сложность по Колмогорову $K(x)$ для любого заданного слова $x \in (\Sigma_{\mathrm{bool}})^*$.*

Доказательство. Пусть $L_{\mathrm{H}} \in \mathcal{L}_{\mathrm{R}}$, и пусть H – некоторый алгоритм, решающий проблему L_{H}. Следующий алгоритм A (рис. 5.22) вычисляет $K(x)$ для каждого слова $x \in (\Sigma_{\mathrm{bool}})^*$.

A генерирует слова $w_1, w_2, w_3, \ldots$ в каноническом порядке. Для каждого натурального i алгоритм A проверяет, является ли w_i двоичным кодом некоторой программы на Паскале. Если w_i не является кодом никакой программы, то A продолжает вычисления с w_{i+1}. А если $w_i = \mathrm{Kod}(M)$ для некоторой программы на Паскале M, то A применяет H для определения того, останавливается ли M на слове λ.

Если M останавливается на входе λ, то A моделирует работу машины M над входом λ.

Если M останавливается с выходом $u = M(\lambda)$, то A проверяет, выполнено ли равенство $u = x$. Если $u = x$, то выходом алгоритма A является

$$K(x) = |w_i|.$$

Если же $u \neq x$, то A продолжает вычисления со словом w_{i+1}.

[34] Заметим ещё, что двоичная длина алгоритма A также является костантой, не зависящей от n.

Если $|w_i|$ – выход машины A для входа x, то w_i – самое короткое слово, обладающее свойством

$$w_i = \mathrm{Kod}(M)$$

для программы на Паскале M, и M генерирует слово x. Таким образом, алгоритм A вычисляет сложность по Колмогорову $K(x)$ для любого заданного входа $x \in (\Sigma_{\mathrm{bool}})^*$. □

Упражнение 5.68. Докажите следующие утверждения:

- если $L_{\mathrm{U}} \in \mathcal{L}_{\mathrm{R}}$, то существует алгоритм, который вычисляет значение $K(x)$ для любого слова $x \in (\Sigma_{\mathrm{bool}})^*$;
- если $L_{\mathrm{empty}} \in \mathcal{L}_{\mathrm{R}}$, то существует алгоритм, который вычисляет значение $K(x)$ для любого слова $x \in (\Sigma_{\mathrm{bool}})^*$.

5.7 Заключение

Основные методы доказательств[35], применяемых в теории вычислимости, относятся к теории множеств. Самые важные из них таковы:

- сравнение бесконечных чисел – точнее, мощностей бесконечных множеств;
- метод диагонализации;
- метод сводимости.

Множество A имеет мощность, которая является по крайней мере столь же большой, как и мощность множества B, если существует однозначное отображение, действующее из B в A. Для любого бесконечного множества A выполнено равенство $|A| = |A \times A|$ – следовательно, $|\mathbb{N}_0| = |\mathbb{Q}^+|$. Наименьшее бесконечное множество – $\mathbb{N}_0$, и каждое множество C с мощностью $|C| \leq |\mathbb{N}_0|$ называется счётным. Каждое множество B, имеющее мощность $|B| > |\mathbb{N}_0|$, называется несчётным.

Множество всех машин Тьюринга (программ) счётно, а множество всех языков (задач) над алфавитом $\{0,1\}$ несчётно. Поэтому большинство вычислительных задач неразрешимо.

Если $|A| < |B|$ для двух бесконечных множеств A и B, то метод диагонализации (и ему подобные технологии) позволяет сконструировать элемент множества $B - A$.

Язык L, допускаемый некоторой машиной Тьюринга, называется рекурсивно перечислимым. Если при этом существует машина Тьюринга M, останавливающаяся для каждого входа в одном из состояний q_{accept} либо q_{reject}, то язык L называется рекурсивным (или разрешимым). Метод диагонализации позволяет создать конкретный язык, не являющийся рекурсивно перечислимым – т. н. язык диагонализации L_{diag}.

Применяя метод сводимости, можно показать, что некоторые проблемы являются по крайней мере такими же трудными[36], как L_{diag}, – и, следовательно, соответствующие языки нерекурсивны. Наиболее важные примеры неразрешимых проблем принадлежности – это проблема универсального языка и проблема останова. Универсальный язык содержит все слова, состоящие из следующих двух частей: первая кодирует машину Тьюринга M, а вторая является некоторым словом w, таким что $w \in L(M)$.

[35] Можно сказать – инструменты.

[36] Относительно алгоритмической разрешимости.

Проблема заключается в том, чтобы для заданных машины M и входного слова w решить, принимает ли M это слово. А проблема останова заключается в получении ответа на вопрос, останавливается ли заданная машина Тьюринга M на заданном входе w.

Теорема Райса утверждает, что неразрешимой является каждая нетривиальная проблема принадлежности о машинах Тьюринга (программах). Следовательно, не существует алгоритмов как для тестирования правильности программы, так и для ответа на вопрос, останавливается ли программа. Поэтому, нельзя алгоритмически решить, является ли данная машина Тьюринга (программа) алгоритмом[37] для решения некоторой заданной проблемы – даже если сама проблема тривиальна, вроде вычисления функции-константы. Используя метод сводимости, можно применять доказательство неразрешимости и в более сложных случаях – за пределами задач о машинах Тьюринга (программах). Хорошим примером такой задачи является проблема соответствий Поста, которая может рассматриваться как специальный вариант игры в домино.

Первые аргументы необходимости изучения проблем (не)разрешимости математических задач появились на основе результатов известного математика Давида Гильберта. В начале 20-го века он сформулировал иследовательский проект по математике, целью которого была разработка формализма (математической теории), в рамках которого можно было бы решить *все* математические проблемы. Курт Гёдель в 1931 г. доказал нереальность стремлений Гилберта: согласно его теореме, каждая нетривиальная математическая теория[38] неразрешима – т. е. на языке любой такой теории можно сформулировать проблемы, которые не могут быть решены в её рамках. Работа [20], в которой приведено конструктивное доказательство теоремы Гёделя, привела к формализации понятия «алгоритм» – и, следовательно, к построению основ теории вычислимости.

Неразрешимость универсального языка была установленна Тьюрингом [68]. В 1953 году Райс [57] опубликовал результат, известный в настоящее время как теорема Райса. Неразрешимость проблемы соответствий Поста была доказана Постом в [52].[39]

Используя метод сводимости для определения эквивалентности задач относительно рекурсивности, мы разделяем множество *не*рекурсивных языков на бесконечное множество классов $\mathcal{L}_i$ ($i \in \mathbb{N}$). Значение фразы «язык L принадлежит классу $\mathcal{L}_i$» таково: L являлся *бы* неразрешимым даже при дополнительном предположении о разрешимости всех языков классов $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_{i-1}$. Очень интересный и важный момент здесь заключается в следующем: существуют практически значимые задачи, принадлежащие к высшим классам этой иерархии.

Для более всестороннего и детального изучения основ теории вычислимости мы настоятельно рекомендуем соответствующие главы учебников [27, 28, 65]. А для углубления этих знаний – классические учебники Трахтенброта [67] и Роджерса [59].

---

[37] Напомним, что алгоритм завершает свою работу (останавливается, не зацикливается) для каждого корректного входа.

[38] Нетривиальность теории в данном случае означает, что она включает в себя по крайней мере формальную арифметику.

[39] К настоящему времени опубликовано несколько доказательств неразрешимости этой проблемы. Наиболее интересным (и применимым в самых разных областях теоретической информатики) является, по-видимому, доказательство, приведённое в книге *[С.С.Марченков. «Замкнутые классы булевых функций», М., Физматлит, 2000]. (Прим. перев.)*

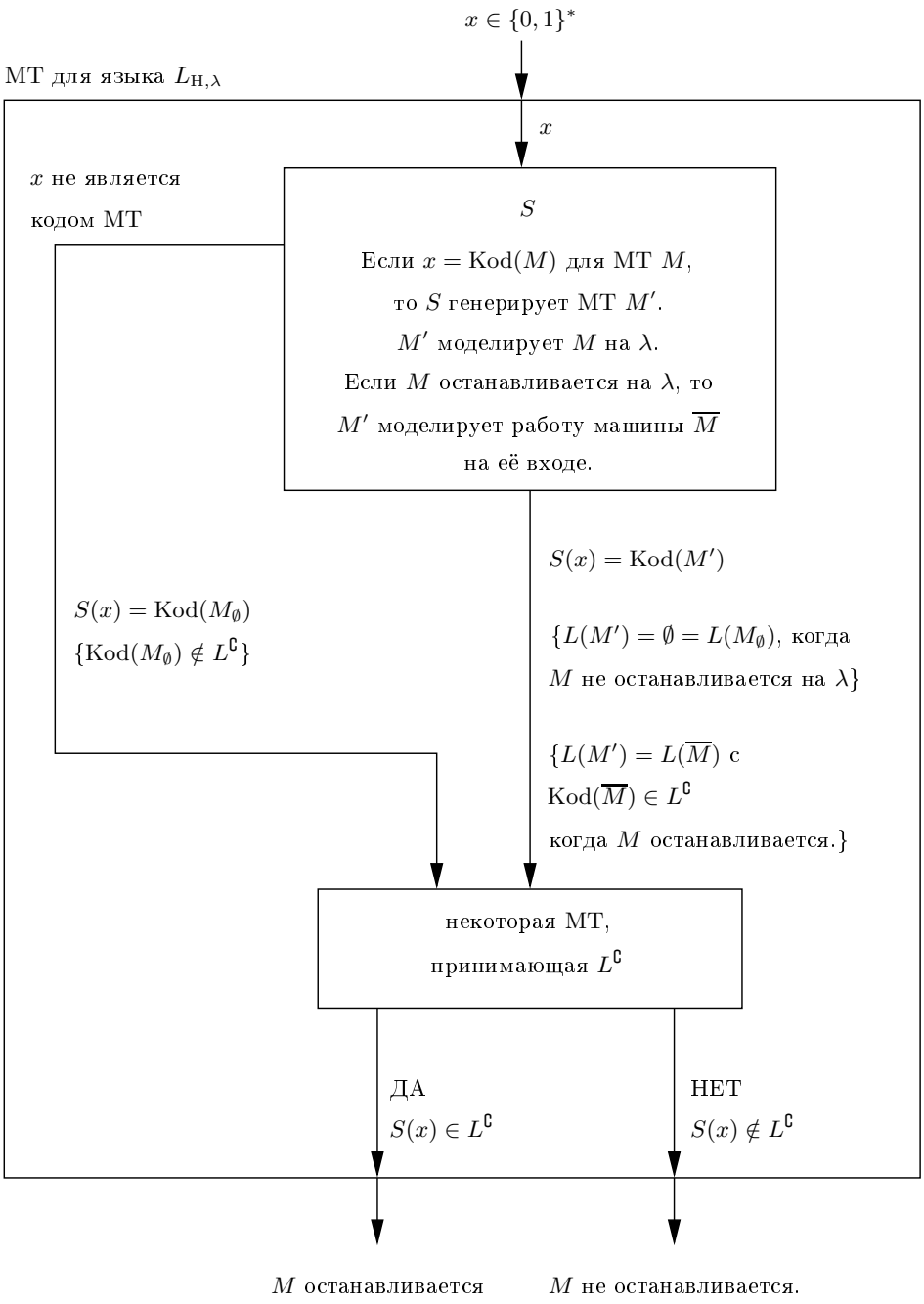

$x \in \{0,1\}^*$
МТ для языка $L_{\text{н},\lambda}$
x
x не является кодом МТ
S
Если $x = \text{Kod}(M)$ для МТ M, то S генерирует МТ M'.
M' моделирует M на λ.
Если M останавливается на λ, то M' моделирует работу машины $\overline{M}$ на её входе.
$S(x) = \text{Kod}(M')$
$\{L(M') = \emptyset = L(M_\emptyset)$, когда M не останавливается на $\lambda\}$
$\{L(M') = L(\overline{M})$ с $\text{Kod}(\overline{M}) \in L^{\complement}$ когда M останавливается.$\}$
$S(x) = \text{Kod}(M_\emptyset)$
$\{\text{Kod}(M_\emptyset) \notin L^{\complement}\}$
некоторая МТ, принимающая $L^{\complement}$
ДА
$S(x) \in L^{\complement}$
НЕТ
$S(x) \notin L^{\complement}$
M останавливается
M не останавливается.

Рис. 5.14.

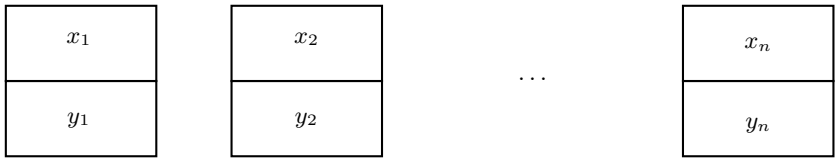


Рис. 5.15.

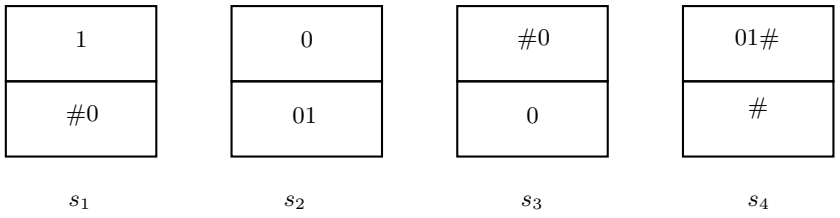


Рис. 5.16.

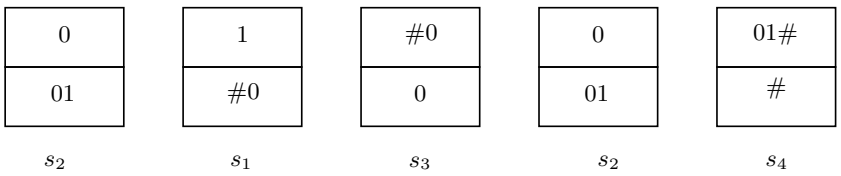


Рис. 5.17.

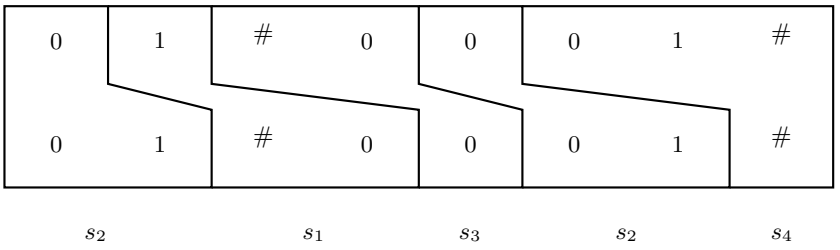


Рис. 5.18.

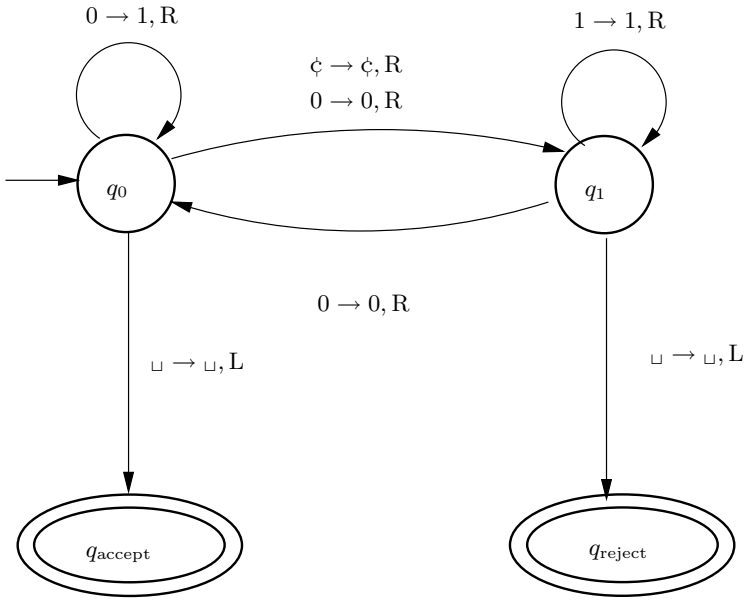


Рис. 5.19.

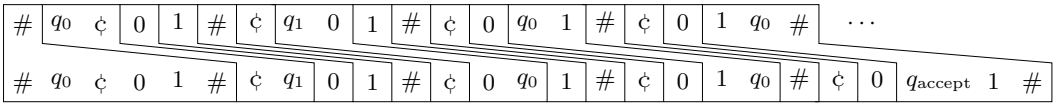


Рис. 5.20.

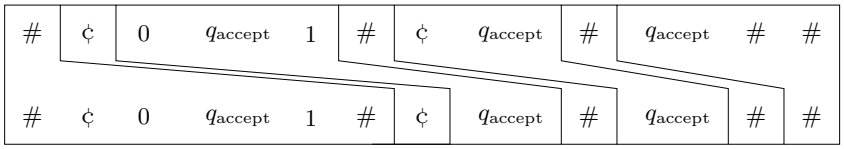


Рис. 5.21.

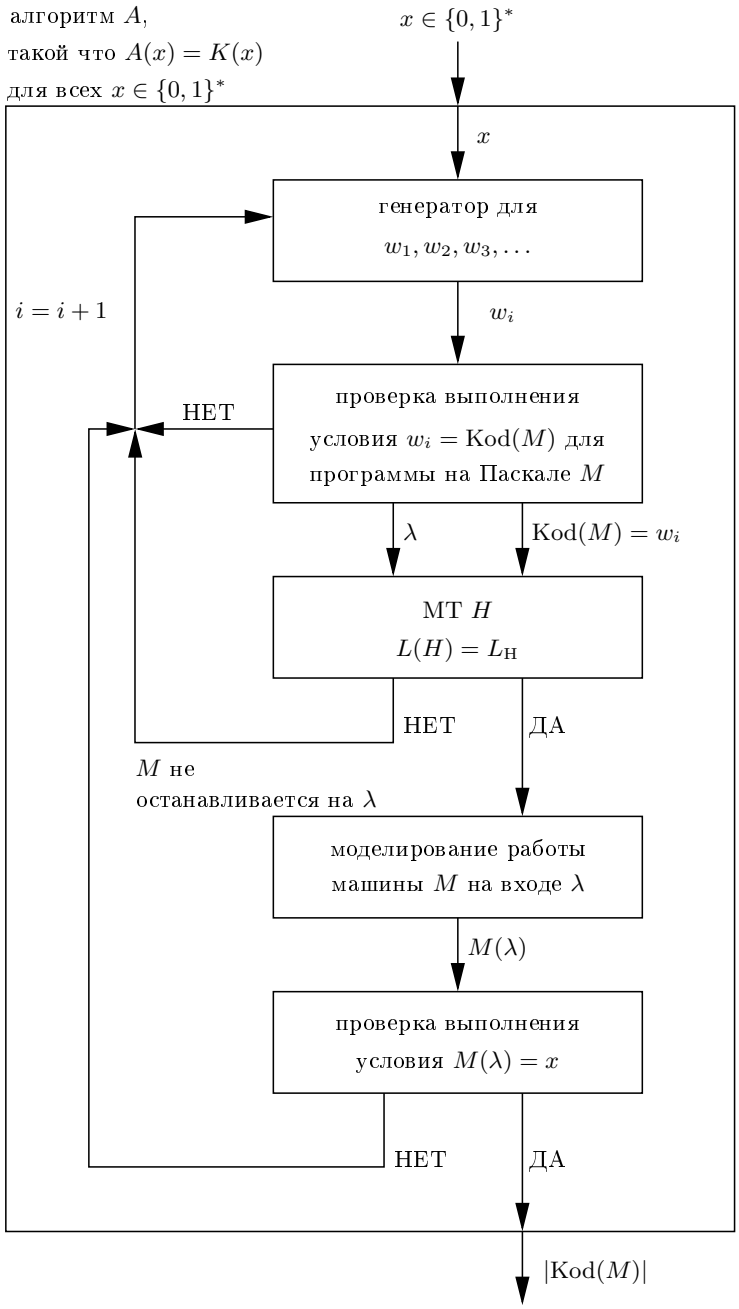

алгоритм A,
такой что $A(x) = K(x)$
для всех $x \in \{0,1\}^*$
$x \in \{0,1\}^*$
x
генератор для
$w_1, w_2, w_3, \ldots$
w_i
$i = i + 1$
проверка выполнения
условия $w_i = \mathrm{Kod}(M)$ для
программы на Паскале M
НЕТ
λ
$\mathrm{Kod}(M) = w_i$
МТ H
$L(H) = L_{\mathrm{H}}$
НЕТ
ДА
M не
останавливается на λ
моделирование работы
машины M на входе λ
$M(\lambda)$
проверка выполнения
условия $M(\lambda) = x$
НЕТ
ДА
$|\mathrm{Kod}(M)|$

Рис. 5.22.

Нет большей потери,
чем потерянное время.
Микельанджело

6 Теория сложности

6.1 Цели и задачи главы

Теория вычислимости даёт методы классификации задач относительно их алгоритмической разрешимости: задача алгоритмически разрешима, если существует алгоритм, решающий её – и алгоритмически неразрешима, если такого алгоритма не существует. А в качестве продолжения теории вычислимости можно рассматривать *теорию сложности*, в том смысле, что последняя пытается разделить класс алгоритмически разрешимых задач на ряд подклассов – согласно достижимой эффективности их решения.

С 1960-х годов, когда компьютеры начали использоваться уже не только в нескольких научно-исследовательских институтах, многие программисты-практики поняли, что *существования алгоритма* для некоторой задачи недостаточно для её *решения* с помощью компьютера. Было описано множество важных для практического использования алгоритмически разрешимых задач, для которых все реализованные на компьютерах алгоритмы работали так долго, что компьютеры ломались до получения какого-либо результата. В результате возник вопрос: являются ли все эти трудности результатом нашей неспособности найти эффективное алгоритмическое решение проблемы – или результатом особого свойства этой проблемы, просто не предусматривающей никакого эффективного алгоритмического решения?

Все эти рассуждения приводят нас к идее измерения сложности вычислительных задач относительно количества компьютерных ресурсов, необходимых и достаточных для их вычисления. А эта идея, в свою очередь, приводит нас к классификации алгоритмически разрешимых задач согласно степени их вычислительной сложности.

Таким образом, теория сложности – это теория количественных законов и ограничений алгоритмической обработки информации. У этой теории есть даже своя единица измерения. Например, можно определять алгоритмически разрешимую задачу как неразрешимую на практике (труднорешаемую) – если для программного выполнения любого алгоритма, решающего эту задачу с реальными входными данными, необходимо больше мощности, чем мощность известной нам части вселенной.

Основные цели теории сложности таковы.

- Создать методы оценки вычислительной сложости, необходимой и достаточной для решения конкретных алгоритмических задач. При этом вычислительная сложность

– это либо сложность по времени (временна́я сложность), либо сложность по памяти.[1]

- Определить понятие класса «практически разрешимых» задач и разработать методы их классификации – на «практически разрешимые» (легко, эффективно разрешимые) и «практически неразрешимые» (трудно разрешимые).[2]
- Сравнить эффективность (вычислительную мощность) детерминированных, недетерминированных и рандомизированных алгоритмов.

Поскольку в этой главе происходит только наше первое знакомство с теорией сложности, мы ограничимся следующими целями.

В разделе 6.2 мы научимся измерять вычислительную сложность машин Тьюринга и программ. Мы также рассмотрим некоторые свойства этих единиц измерения.

В разделе 6.3 обсуждается задача определения сложности алгоритмических проблем. Основные классы сложности проблем принадлежности для языков определены как классы языков, которые разрешимы в рамках рассматриваемой сложности. Мы также обсудим, как определить класс разрешимых на практике (легко разрешимых) проблем принадлежности, и приводим аргументы в пользу того, чтобы «в качестве первого приближения» этого класса выбрать класс задач, решаемых за время, полиномиально зависящее от длины входных данных.

В разделе 6.4 мы покажем, как измерять сложность недетерминированных машин Тьюринга, а также введём основные классы недетерминированной сложности.

Раздел 6.5 посвящён сравнению эффективности детерминированных и недетерминированных вычислений. Это сравнение затрагивает философские основы математики. Мы покажем, что временна́я сложность недетерминированных вычислений соответствует сложности детерминированной проверке корректности данного математического доказательства некоторого утверждения – в то время как детерминированная временна́я сложность соответствует сложности создания (поиска) математического доказательства такого утверждения. Поэтому вопрос «могут ли недетерминированные алгоритмы быть более эффективными, чем детерминированные» равносилен вопросу «проще ли проверка корректности математических доказательств, чем автоматическое (алгоритмическое) создание таких доказательств».

Раздел 6.6 знакомит с понятием NP-полноты. Это понятие в настоящее время является основным инструментом для доказательства того факта, что некоторые конкретные проблемы являются трудными – в том смысле, что они не принадлежат классу легко разрешимых задач. Это – первый пример, показывающий, что такое нереалистическое понятие, как недетерминированное вычисление, может быть удачно исполь-

[1] В отличие от размера необходимой памяти компьютера, число необходимых компьютерных команд, по-видимому, *уже сейчас* требует небольших пояснений. Во-первых, мы имеем в виду не число команд для представления программы в памяти компьютера, а число команд, выполняемых в процессе работы алгоритма. А во-вторых, мы для упрощения считаем, что все компьютерные команды выполняются за одинаковое время – однако стоит отметить, что это упрощение практически не влияет на получаемые нами оценки вычислительной сложости; подробнее см. следующие разделы. *(Прим. перев.)*

[2] В литературе на русском языке вместо «разрешимые» во всех этих случаях употребляется также слово «решаемые». Например, название книги [19] в русском переводе звучит как «Вычислительные машины и труднорешаемые задачи». Согласно устоявшейся русской терминологии, названия «труднорешаемые» и «легкорешаемые» будем писать в одно слово. *(Прим. перев.)*

зовано в качестве полноценного инструмента для детерминированных вычислений – например, в научных исследованиях.

6.2 Меры сложности

Для определения основной единицы измерения сложности мы используем модель многоленточных машин Тьюринга. Главными причинами, послужившими для выбора этой модели, являются, с одной стороны, её достаточная простота, а с другой стороны – то, что она соответствует фундаментальной структуре, используемой в фон-неймановской модели вычислительной машины. Позже мы увидим, что модель многоленточных машин Тьюринга достаточно надёжна – это проявляется в том, что определённые с помощью ММТ основные результаты, касающиеся сложности алгоритмов, сохраняются и для сложности компьютерных программ, написанных на произвольном языке программирования. При этом наиболее обоснованными являются результаты классификации, связанные с легко разрешимыми проблемами для машин Тьюринга.

Итак, определим единицы измерения для двух основных величин: временно́й сложности (сложности по времени) и сложности по памяти. Временна́я сложность вычислений – это число элементарных команд (шагов машины Тьюринга), выполняемых при этом вычислении. Следовательно, существует линейное соотношение между временно́й сложностью вычислений и «вычислительной мощностью», необходимой для выполнения этих вычислений. Сложность по памяти – это используемый в вычислениях объём памяти машины, который измеряется количеством необходимых машинных слов. Для модели машины Тьюринга размер машинных слов определяется с помощью используемого рабочего алфавита – поскольку его символы описывают допустимое содержание машинных слов (регистров).

Определение 6.1. *Пусть M – некоторая* МТ *или* ММТ*, которая всегда останавливается, Σ – входной алфавит этой машины, $x \in \Sigma^*$, а $D = C_1, C_2, \ldots, C_k$ – некоторое вычисление машины M над словом x.* **Временна́я сложность (сложность по времени) $\mathbf{Time}_M(x)$ вычисления машины M над x** *определяется как*

$$\mathbf{Time}_M(x) = k - 1,$$

т. е. как число шагов вычисления D.

Временна́я сложность машины M *– это функция* $\mathrm{Time}_M : \mathbb{N}_0 \to \mathbb{N}_0$*, определённая как*

$$\mathbf{Time}_M(n) = \max\{\mathrm{Time}_M(x) \mid x \in \Sigma^n\}.$$

Отметим, что $\mathrm{Time}_M(n)$ определяется так, что каждое входное слово длины n (т. е. каждый вход, принадлежащий множеству Σ^n) распознаётся машиной M за время, не превышающее $\mathrm{Time}_M(n)$ – причём существует именно такой вход x длины n, что $\mathrm{Time}_A(x) = \mathrm{Time}_A(n)$. Иными словами, $\mathrm{Time}_M(n)$ – это временна́я сложность самых долгих вычислений с каким-либо из входов длины n. Поэтому этот вариант измерения сложности называется сложностью в худшем случае. Сложность в худшем случае не может быть признана удачной мерой измерения – это хорошо видно на примере какой-нибудь конкретной имашины Тьюринга, имеющей для входов одной и той же длины весьма разные значения временно́й сложности. В подобных случаях мы можем

вместо сложности в худшем случае рассматривать «средний» вариант – в нём сложность является усреднённой величиной соответствующих значений, полученных для всех входов с одной и той же длиной n.

Однако сложность в худшем случае всё же обычно предпочтительнее – по следующим двум важным причинам. Во-первых, определение средней временно́й сложности алгоритма обычно является более трудной задачей, чем определение $\mathrm{Time}_M(n)$: часто даже невозможно произвести точное вычисление этого среднего значения. Во-вторых, люди имеют склонность к гарантиям – а измерение сложности в худшем случае убеждает, что соответствующий алгоритм может решать любой частный случай проблемы x длины n за время $\mathrm{Time}_M(n)$. Итак, в этой книге мы заострим наше внимание на варианте вычисления сложности в худшем случае.

Определение 6.2. *Пусть $k \in \mathbb{N}$, M – некоторая k-ленточная*-МТ*, которая всегда останавливается,*

$$C = (q, x, i, \alpha_1, i_1, \alpha_2, i_2, \ldots, \alpha_k, i_k),$$
$$\text{где } 0 \le i \le |x| + 1 \text{ и } 0 \le i_j \le |\alpha_j| + 1 \text{ для } j = 1, \ldots, k$$

– некоторая конфигурация машины M. **Сложность по памяти вычисления C** *есть*[3]

$$\mathbf{Space}_M(C) = \max\{|\alpha_i| \mid i = 1, \ldots, k\}.$$

Пусть $C_1, C_2, \ldots, C_l$ – некоторое вычисление машины M над словом x. **Сложность по памяти машины M над x** *есть*

$$\mathbf{Space}_M(x) = \max\{\mathrm{Space}_M(C_i) \mid i = 1, \ldots, l\}.$$

Сложность по памяти машины M *– это функция вида*

$$\mathrm{Space}_M : \mathbb{N}_0 \to \mathbb{N}_0,$$

заданная следующим образом:

$$\mathbf{Space}_M(n) = \max\{\mathrm{Space}_M(x) \mid x \in \Sigma^n\}.$$

Может показаться странным тот факт, что мы определили сложность по памяти некоторой конфигурация как *максимум* длин всех используемых лент, содержание которые не является пустым – вместо *суммы* длин лент. Однако не имеет значения, какая именно из этих двух возможностей выбрана: согласно лемме 4.13, для любого $k \in \mathbb{N}_0$ все k лент могут быть промоделированы с помощью одной – причём её длина не превосходит максимума длин моделируемых k лент. Из этого утверждения выводится следующая лемма.

Лемма 6.3. *Пусть $k \in \mathbb{N}$. Для любой k-ленточной*-МТ *A, которая всегда останавливается, существует эквивалентная* 1*-ленточная*-МТ *B, такая что*

$$\mathrm{Space}_B(n) \le \mathrm{Space}_A(n).$$

[3] Заметим, что сложность по памяти не зависит от количества символов используемого алфавита.

Это свойство сложности по памяти вытекает из того факта, что количество элементов рабочего алфавита машины M (длина машинных слов реального компьютера) не влияет на определение значения $\mathrm{Space}_M(n)$. Поэтому наше определение сложности по памяти не подходит для соответствующих сомножителей-констант.

Лемма 6.4. *Пусть $k \in \mathbb{N}$. Для любой k-ленточной*-МТ A *существует k-ленточная*-МТ B, *такая что $L(A) = L(B)$, и*

$$\mathrm{Space}_B(n) \leq \frac{\mathrm{Space}_A(n)}{2} + 2.$$

Доказательство. Опишем только идею доказательства. Пусть Γ_A – рабочий алфавит машины A. Сконструируем рабочий алфавит Γ_B машины B таким образом, чтобы он содержал все символы Декартова произведения $\Gamma_A \times \Gamma_A$. Если

$$\alpha_1, \alpha_2, \ldots, \alpha_m$$

– содержимое i-й рабочей ленты машины A, $i \in \{1, 2, \ldots, k\}$, и при этом головка установлена на символ α_j для некоторого значения $j \in \{1, 2, \ldots, m\}$, то содержимое i-й ленты машины B является словом

$$\dot{\varsigma}\binom{\alpha_1}{\alpha_2}\binom{\alpha_3}{\alpha_4}\ldots\binom{\alpha_{m-1}}{\alpha_m}, \text{ если } m \text{ чётное, и}$$

$$\dot{\varsigma}\binom{\alpha_1}{\alpha_2}\binom{\alpha_3}{\alpha_4}\ldots\binom{\alpha_{m-1}}{\sqcup}, \text{ если } m \text{ нечётное.}$$

Головка i-й рабочей ленты машины B установлена на пару, которая содержит α_i,[4] а B остаётся в положении, соответствующем тому из двух символов этой пары, на который установлена головка i-й ленты машины A. Итак, B может однозначно представить любую конфигурацию машины A размером $1 + \lceil \mathrm{Space}_A(n)/2 \rceil$.

Достаточно просто описать общий принцип пошагового моделирования работы машины A с помощью машины B. Передвижения головки машины A моделируются либо соответствующим передвижением головки машины B, либо изменением состояния последней. А изменение состояния моделируется путём изменения специального счётчика, который следит за тем, какая именно пара символов читается машиной A. Таким путём мы можем создать ММТ B, такую что

$$L(B) = L \quad \text{и} \quad \mathrm{Space}_B(n) \leq \frac{\mathrm{Space}_A(n)}{2} + 2.$$

□

Применив несколько раз лемму 6.4, мы для любой заданной многоленточной машины и любой константы k можем создать такую эквивалентную ММТ, у которой сложность по памяти ограничена значением

$$\frac{\mathrm{Space}_A(n)}{k} + 2.$$

[4] Т. е. на символ $\binom{\alpha_i}{\alpha_{i+1}}$ для нечётного i, либо на $\binom{\alpha_{i-1}}{\alpha_i}$ для чётного i.

Отметим, что все приведённые нами оценки возможны только потому, что сложность по памяти не зависит от размера используемого алфавита.

Можно уменьшить временну́ю сложность – тем же способом, которым мы уменьшали сложность по памяти в лемме 6.4. Если хранить большее (чем 2) число символов машины ММТ A в одном символе создаваемой нами ММТ – то функция переходов подобной машины, определённая над такими комбинированными символами, будет задавать более сложные операции, чем функция переходов описанной ранее машины B. Это даёт возможность моделировать несколько шагов машины A с помощью меньшего числа шагов машины B – достигая тем самым ускорения вычислений. Сформулируем этот результат в следующем упражнении.

Упражнение 6.5. Докажите такое утверждение. Для любой ММТ M существует эквивалентная ММТ A, такая что

$$\mathrm{Time}_A(n) \leq \frac{\mathrm{Time}_M(n)}{2} + 2n.$$

Утверждения леммы 6.4 и упражнения 6.5 показывают, что введённый вариант измерения сложности является приблизительным. Однако для наших целей это не является недостатком – потому что различия между единицами измерения сложности в различных вычислительных моделях часто значительно больше, чем различия, выраженные константами-сомножителями. Нам нужны надёжные результаты классификации, эффективные для всех корректных вычислительных моделей. Поэтому нас интересуют оценки асимптотического роста функций сложности Time_M и Space_M – значительно больше, чем точные значения этих функций. Для асимптотического анализа функций сложности мы используем стандартные Ω-, O-, Θ- и o-символики, взятые из математического анализа.[5]

Определение 6.6. *Для любой функции* $f : \mathbb{N}_0 \to \mathbb{R}^+$ *определим*

$$\boldsymbol{O(f(n))} = \{r : \mathbb{N}_0 \to \mathbb{R}^+ \mid \exists n_0 \in \mathbb{N}_0, \exists c \in \mathbb{N}_0, \text{ такие что } \forall n \geq n_0 : r(n) \leq c \cdot f(n)\}.$$

Для любой функции $r \in O(f(n))$ *мы будем говорить, что* r **растёт асимптотически не быстрее, чем** $\boldsymbol{f}$.

Для любой функции $g : \mathbb{N}_0 \to \mathbb{R}^+$ *определим*

$$\boldsymbol{\Omega(g(n))} = \{s : \mathbb{N}_0 \to \mathbb{R}^+ \mid \exists n_0 \in \mathbb{N}_0, \exists d \in \mathbb{N}_0, \text{ такие что } \forall n \geq n_0 : s(n) \geq \frac{1}{d} \cdot g(n)\}.$$

Для любой функции $s \in \Omega(g(n))$ *будем говорить, что* s **растёт асимптотически не медленнее, чем** $\boldsymbol{g}$.

Для любой функция $h : \mathbb{N}_0 \to \mathbb{R}^+$ *определим*

$$\begin{aligned}\boldsymbol{\Theta(h(n))} &= \{q : \mathbb{N}_0 \to \mathbb{R}^+ \mid \exists c, d, n_0 \in \mathbb{N}_0, \text{ такие что } \forall n \geq n_0 : \\ &\qquad \frac{1}{d} \cdot h(n) \leq q(n) \leq c \cdot h(n)\} \\ &= O(h(n)) \cap \Omega(h(n)).\end{aligned}$$

[5] Русские названия определяемых понятий согласуются с применяемыми в известных учебниках. См., например, *[Л.Д.Кудрявцев. «Математический анализ», т. 1, М., Высшая школа, 1973]. (Прим. перев.)*

Если $q \in \Theta(h(n))$, то мы будем говорить, что q и h **имеют одинаковый асимптотический рост**.

Пусть f и g – функции из $\mathbb{N}_0$ в $\mathbb{R}^+$. Если

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0,$$

то мы будем говорить, что $\boldsymbol{g}$ **растёт асимптотически быстрее, чем** $\boldsymbol{f}$, *и писать* $\boldsymbol{f(n) = o(g(n))}$.

Упражнение 6.7. Какие из следующих утверждений истинны? (Обоснуйте свой ответ.)

- $2^n \in \Theta(2^{n+a})$ для любой константы $a \in \mathbb{N}$.
- $2^{b\cdot n} \in \Theta(2^n)$ для любой константы $b \in \mathbb{N}$.
- $\log_b n \in \Theta(\log_c n)$ для всех положительных действительных чисел $b, c > 1$.
- $(n+1)! \in O(n!)$.
- $\log(n!) \in \Theta(n \cdot \log n)$.

Упражнение 6.8. Докажите следующие утверждения для всех функций $f, g : \mathbb{N}_0 \to \mathbb{R}^+$:

- $f \in O(g)$ и $g \in O(h) \Rightarrow f \in O(h)$;
- $f \in O(g) \Leftrightarrow g \in \Omega(f)$;
- $f \in \Theta(g) \Leftrightarrow g \in \Theta(f) \Leftrightarrow \Theta(f) = \Theta(g)$.

Рассмотрим машину Тьюринга M, заданную на рис. 4.5 – она принимает язык L_{middle}. Эта машина циклически перемещается между левой и правой границами ленты – и в течение каждого из этих перемещений передвигает левый граничный маркер на одну ячейку направо и правый граничный маркер на одну ячейку налево. Используя эту стратегию, машина M в итоге получает позицию ячейки в середине ленты. Очевидно, что временна́я сложность машины M находится в пределах $O(n^2)$.

Упражнение 6.9. Приведите точный анализ функции $\text{Time}_M(n)$ машины M, заданной на рис. 4.5.

Упражнение 6.10. Опишите подробно 1-ленточную-МТ B, допускающую язык $L(B) = L_{\text{middle}}$ и имеющую временну́ю сложность $\text{Time}_B(n) \in O(n)$.

Упражнение 6.11. Приведите асимптотичесикй анализ временно́й сложности машины A, заданной на рис. 4.6. Эта машина принимает язык $L_{\mathcal{P}}$.

1-ленточная-МТ A, приведённая на рис. 4.10, принимает язык

$$L_{\text{equal}} = \{w\#w \mid w \in (\Sigma_{\text{bool}})^*\}$$

путём копирования на рабочую ленту префикса входного слова – вплоть до символа $\#$, а затем сравнения содержимого входной и рабочей лент. Мы видим, что $\text{Time}_A(n) \leq 3 \cdot n \in O(n)$, а $\text{Space}_M(n) \in O(n)$.

Упражнение 6.12. Опишите подробно 2-ленточную-МТ M с языком $L(M) = L_{\text{equal}}$, сложностью по времени $\text{Time}_M(n) \in O(\frac{n^2}{\log_2 n})$ и сложностью по памяти $\text{Space}_M(n) \in O(\log_2 n)$.

Выше мы описали и рассмотрели временну́ю сложность и сложность по памяти для многоленточных машин Тьюринга. Так как основная цель теории сложности заключается в том, чтобы согласно вычислительной сложности классифицировать алгоритмические проблемы, нам надо ввести новые варианты определений сложности по времени и по памяти – теперь не для машин Тьюринга, а для проблем (задач). Естественная идея – это определение временно́й сложности проблемы U как временно́й сложности «асимптотически оптимальной» ММТ (алгоритма) M для U. При этом оптимальность машины M для заданной проблемы U может быть определена требованием

$$\mathrm{Time}_A(n) \in \Omega(\mathrm{Time}_M(n))$$

для каждой ММТ A, которая также решает проблему U – т. е. не должно существовать никакой ММТ для U, которая бы решала U асимптотически быстрее, чем M.

Итак, основная идея вышесказанного кажется на первый взгляд вполне обоснованной: временна́я сложность проблемы должна являться сложностью лучшего её алгоритмического решения. Но, к сожалению, подобное определение «не работает» – и следующая теорема объясняет, почему это так.

Теорема 6.13. *Существует проблема принадлежности, для которой выполняется следующее утверждение. Для каждой решающей эту проблему* ММТ A *существует также решающая её* ММТ B*, такая что для бесконечно многих натуральных* $n \in \mathbb{N}$ *выполнено неравенство*

$$\mathrm{Time}_B(n) \leq \log_2(\mathrm{Time}_A(n)).$$

Теорема 6.13 утверждает, что есть проблемы, для которых можно существенно улучшить *любой* алгоритм – более того, для них существует *бесконечная* последовательность алгоритмических улучшений. Следовательно, асимптотически лучших (оптимальных) алгоритмов решения таких задач не существует – и поэтому невозможно описать сложность задач предложенным выше способом. Что же нам делать? Мы можем классифицировать проблемы без описания их сложности – просто на основе нижней и верхней границ сложности проблемы, как предложенно в следующем определении.

Определение 6.14. *Пусть* L *– некоторый язык, а* f *и* g *– функции, действующие из* $\mathbb{N}_0$ *в* $\mathbb{R}^+$*. Будем говорить, что* $O(g(n))$ *–* **верхняя граница временно́й сложности языка** $\boldsymbol{L}$*, если существует некоторая* ММТ *(алгоритм)* A*, такая что*

$$L = L(A) \quad и \quad \mathrm{Time}_A(n) \in O(g(n)).$$

Будем говорить, что $\Omega(f(n))$ *–* **нижняя граница временно́й сложности языка** $\boldsymbol{L}$*, если условие*

$$\mathrm{Time}_B(n) \in \Omega(f(n))$$

выполняется для каждой ММТ *(алгоритма)* B*, для которой* $L(B) = L$*.*

Некоторая ММТ *(алгоритм)* C *является* **оптимальной** *для* L *тогда и только тогда, когда:*

- $L(C) = L$*;*
- $\Omega(\mathrm{Time}_C(n))$ *– нижняя граница временно́й сложности языка* L*.*

Чтобы найти какую-нибудь *верхнюю* границу сложности проблемы U, достаточно найти какой-нибудь алгоритм, который решает эту проблему – и проанализировать его сложность. А установление нетривиальной *нижней* границы сложности проблемы U – очень трудная задача, поскольку она требует доказательства того, что каждый из бесконечного числа известных и неизвестных алгоритмов решения проблемы U должен иметь временну́ю сложность в пределах $\Omega(f(n))$ – для некоторой функции f. Это доказательство является ещё одним примером доказательства несуществования – мы должны доказать, что *не* существует алгоритмов, решающих проблему U и имеющих временну́ю сложность, асимптотически меньшую, чем $f(n)$. Лучшей иллюстрацией трудности доказательства нижних пределов сложности проблемы является тот факт, что мы знаем тысячи алгоритмических задач, для которых:

- временна́я сложность лучшего из известных алгоритмов экспоненциальна относительно длины входа;
- ни для какого из этих алгоритмов неизвестно нижней границы, худшей, чем линейная.[6]

Итак, для целого класса подобных задач мы предполагаем, что не существует алгоритмов их решения с полиномиальным относительно размера входа временем работы – но не способны этого доказать. Отсутствие достаточно мощного математического аппарата для определения сложности нижних границ делает классификацию задач относительно их алгоритмической сложности очень трудной. В разделе 6.6 мы покажем, как можно преодолеть эту трудность – выдвинув перед этим одно обоснованное предположение, дающее возможность доказать несуществование полиномиально-временны́х алгоритмов для данных алгоритмических проблем (языков). Подробные рассуждения об обоснованности этого предположения будут приведены в разделах 6.5 и 6.6.

Ранее мы объяснили, как измерять сложность на основе абстрактных машинных моделей. И уже в этом разделе мы обсудим измерение сложности программ – написанных на произвольном языке программирования. Мы выделим два основных измерения сложности – а именно **однородное измерение** (uniform cost) и **логарифмическое измерение**.

Подход, основанный на однородном измерении, проще – но менее точен. Измерение временно́й сложности включает определение суммарного числа элементарных команд (инструкций)[7], выполняемых в рассматриваемом вычислении (т. е. в процессе его выполнения) – а измерение сложности по памяти включает определение числа переменных, используемых в вычислении. Преимущество этого измерения заключается в его простоте – в том смысле, что оно упрощает анализ сложности проблемы (алгоритма). А недостаток в том, что оно не всегда адекватно – потому что предлагает одинаковую «цену» 1 для любых арифметических операций над двумя целыми числами – независимо от их размера. Если операнды являются целыми числами, состоящими из нескольких сотен битов в двоичном представлении, то ни один из них не может поместиться в одно машинное слово (16, 32 или даже 64 бита). Поэтому в подобных случаях операнды должны храниться в нескольких машинных словах – т. е. нам для их хранения необходимо несколько единиц памяти. А выполнение одной арифметической операции

[6] Типа $\Omega(n \log n)$ и т.п.

[7] Элементарные машинные команды – это арифметические команды над целыми числами, сравнение двух целых, чтение, запись, загрузка целых чисел и символов из оперативной памяти, и т.д.

над двумя такими длинными целыми числами (операции длинной арифметики) соответствует работе специальной подпрограммы, выполняющей одну подобную операцию с помощью нескольких операций над целыми числами, записываемыми в машинные слова разумного размера. Итак, однородное измерение можно удачно использовать в тех случаях, когда мы можем предполагать, что значения всех переменных в течение всего вычисления ограничены некоторой фиксированной константой – зависящей от предполагаемой длины машинного слова.

Однако использование однородного измерения может привести к серьёзным проблемам – и для иллюстрации этого рассмотрим следующий пример. Пусть $a \geq 2$ и k – два натуральных числа, длина которых не превышает длину заданного машинного слова. Рассмотрим задачу вычисления числа a^{2^k}. Данное значение может быть вычислено с помощью следующей программы:

$$\texttt{for } i = 1 \texttt{ to } k \texttt{ do } a := a \cdot a$$

Эта программа вычисляет указанное значение путём выполнения следующих k умножений:

$$a^2 := a \cdot a,\ a^4 := a^2 \cdot a^2,\ a^8 := a^4 \cdot a^4,\ \ldots,\ a^{2^k} := a^{2^{k-1}} \cdot a^{2^{k-1}}.$$

Итак, в данном случае однородная временна́я сложность находится в пределах $O(k)$. Однородная сложность по памяти в данном случае равна 3 – поскольку для выполнения этой программы достаточно 3 переменных. Это может быть воспринято как противоречие с тем, что нам необходимо по крайней мере 2^k битов для представления результата a^{2^k}, а также с тем, что для записи 2^k битов любой машине необходимо $\Omega(2^k)$ операций над своими словами. Поскольку всё это выполняется для каждого натурального k, мы в обоих случаях получаем расхождения между однородной сложностью и любым её реальным измерением – причём для временно́й сложности это расхождение является экспоненциальным, а для сложности по памяти – неограниченным.

Решением этой проблемы – в случае неограниченного роста значений переменных – является использование **логарифмического измерения**. С точки зрения этого варианта при измерении сложности по памяти стоимость каждой элементарной операции является суммой размеров бинарного представления её операндов.[8] При логарифмическом измерении временна́я сложность (стоимость) всего вычисления есть сумма временны́х стоимостей всех операций, выполненных в этом вычислении, а сложность по памяти – сумма длин представлений значений всех использованных в нём пременных. Логарифмическое измерение всегда является практичным – но его недостатком является то, что во многих ситуациях его точное вычисление может быть слишком сложным или требовать слишком большого времени.

6.3 Классы сложности. Класс P

Для определения классов сложности мы используем вычислительную модель многоленточной машины Тьюринга. Рассматриваемые здесь классы сложности являются классами языков, т. е. множествами проблем принадлежности.

[8] С теоретической точки зрения для достижения полной точности мы должны к этой величине добавить двоичнную длину адресов памяти всех необходимых для вычисления переменных (операндов).

Определение 6.15. *Для всех функций f, g, действующих из $\mathbb{N}_0$ в $\mathbb{R}^+$, определим:*

$$\begin{aligned}
\mathbf{TIME}(f) &= \{L(B) \mid B - \textit{некоторая ММТ } c\ \mathrm{Time}_B(n) \in O(f(n))\},\\
\mathbf{SPACE}(g) &= \{L(A) \mid A - \textit{некоторая ММТ } c\ \mathrm{Space}_A(n) \in O(g(n))\},\\
\mathbf{DLOG} &= \mathrm{SPACE}(\log_2 n),\\
\mathbf{P} &= \bigcup_{c\in\mathbb{N}_0} \mathrm{TIME}(n^c),\\
\mathbf{PSPACE} &= \bigcup_{c\in\mathbb{N}_0} \mathrm{SPACE}(n^c),\\
\mathbf{EXPTIME} &= \bigcup_{d\in\mathbb{N}_0} \mathrm{TIME}(2^{n^d}).
\end{aligned}$$

В дальнейшем мы изучим основные соотношения между этими классами сложности и некоторые основные свойства временно́й сложности и сложности по памяти.

Лемма 6.16. *Для любой функции $t : \mathbb{N}_0 \to \mathbb{R}^+$ выполнено следующее:*

$$\mathrm{TIME}(t(n)) \subseteq \mathrm{SPACE}(t(n)).$$

Доказательство. Каждая ММТ M, работающая за время $\mathrm{Time}_M(n)$, может посетить не более $\mathrm{Time}_M(n)$ ячеек каждой из рабочих лент. Таким образом, для каждой ММТ M $\mathrm{Space}_M(n) \leq \mathrm{Time}_M(n)$. □

Следствие 6.17.

$$\mathrm{P} \subseteq \mathrm{PSPACE}.$$

Чтобы получить другие отношения между классами сложности, нам необходимо ввести понятие конструктивных функций. Смысл вводимого далее определения заключается в том, что мы хотели бы создать многоленточные машины Тьюринга, способные контролировать самих себя – в том смысле, что они считают количество используемых (ими самими) вычислительных ресурсов, никогда не превышая при этом некоторый заранее установленный верхний предел. Для возможности такого самоконтроля мы должны описать верхние пределы сложности с помощью нескольких подходящих функций.

Определение 6.18. *Функция $s : \mathbb{N}_0 \to \mathbb{N}_0$ называется* **конструктивной по памяти**, *если существует 1-ленточная*-МТ M, *такая что:*

- $\mathrm{Space}_M(n) \leq s(n)$ *для всех* $n \in \mathbb{N}_0$;
- *для каждого входа* 0^n, *где* $n \in \mathbb{N}_0$, *машина* M *на своей рабочей ленте генерирует слово* $0^{s(n)}$ – *после чего останавливается в состоянии* q_{accept}.

Функция $t : \mathbb{N}_0 \to \mathbb{N}_0$ называется **конструктивной по времени**, *если существует некоторая* ММТ A, *такая что*

- $Time_A \in O(t(n))$;
- *для каждого входа* 0^n, *где* $n \in \mathbb{N}_0$, *машина* A *на своей первой рабочей ленте генерирует слово* $0^{t(n)}$ – *после чего останавливается в состоянии* q_{accept}.

Конструктивными по памяти [по времени] являются произвольные монотонные функции, у которых $f(n) \geq \log_2(n+1)$ $[f(n) \geq n]$. Например, на 1-ленточной-МТ функцию $\lceil \log_2(n+1) \rceil$ можно запрограммировать следующим образом. Машина читает слово 0^n слева направо на входной ленте, при этом записывая на свою рабочую ленту *двоичную* кодировку текущей позиции считывающей головки входной ленты. Последнее может быть легко сделано путём добавления на каждом шаге (до правой границы входной ленты) значения 1 к содержимому рабочей ленты. Если головка на входной ленте достигла символа $\sqcup$, то длина двоичного слова на рабочей ленте в точности равна $\lceil \log_2(n+1) \rceil$. После этого для получения слова $0^{\lceil \log_2(n+1) \rceil}$ на рабочей ленте достаточно поменять все её единицы на нули.

Заметим, что не имеет значения, какой именно формализм используется для определения конструктивности по памяти (1-ленточная-МТ или ММТ) – поскольку любая ММТ может быть промоделирована 1-ленточной-МТ, имеющей такую же конструктивность по памяти.

Теперь опишем работу ММТ M, вычисляющую функцию $\lceil \sqrt{n} \rceil$. Идея программирования M заключается в постоянной проверке выполнения условия

$$i^2 \leq n < (i+1)^2.$$

Такая проверка делается для $i = 1, 2, \ldots$ – и для конкретного значения i она может быть выполнена машиной M с помощью следующего вспомогательного алгоритма.

M записывает слово 0^i на первые две ленты. Чтобы проверить неравенство $i \cdot i \geq n$, M пытается продвинуться на $i \cdot i$ шагов направо от символа ¢ входной ленты – а такие действия машина M может выполнить следующим путём. Сначала головка входной ленты и головка первой рабочей ленты одновременно движутся направо, а головка второй рабочей ленты при этом остаётся неподвижной. Когда головка первой рабочей ленты достигает символа $\sqcup$, она возвращается к левому граничному маркеру ¢ – а головка второй рабочей ленты передвигается на одну клетку направо. После этого M продолжает выполнять те же самые действия с головками входной и первой рабочей лент.

Таким образом, для входной ленты количество шагов направо равно произведению длин содержимого двух рабочих лент – и, следовательно, машина M находит наименьшее значение i, такое что $i \cdot i > n$.

Упражнение 6.19. Дайте формальное описание (т. е. диаграммы) многоленточных машин Тьюринга, использовавшихся выше для построения функций $\lceil \log_2(n+1) \rceil$ и $\lceil \sqrt{n} \rceil$.

Заметим, что описанное выше умножение двух длин может использоваться и для доказательства того факта, что конструктивной по времени является функция $f(n) = n^q$ для любого значения $q \in \mathbb{N}_0$.

Упражнение 6.20. Покажите, что следующие функции являются конструктивными по памяти:

- $\lceil \sqrt{n} \rceil^q$ для любого натурального q;
- $\lceil n^{\frac{1}{3}} \rceil$;
- $\lceil n^{\frac{q}{2}} \rceil$ для любого натурального $q \geq 2$;
- 2^n.

Упражнение 6.21. Покажите, что следующие функции являются конструктивными по времени:

- n^j для любого $j \in \mathbb{N}$;
- $c \cdot n$ для любого $c \in \mathbb{N}$;
- 2^n;
- c^n для любого $c \in \mathbb{N} - \{1\}$.

Упражнение 6.22. Пусть $s(n)$ и $t(n)$ – две функции, конструктивные по памяти [по времени]. Докажите, что функция $t(n) \cdot s(n)$ также является конструктивной по памяти [по времени].

Пусть s – функция, конструктивная по памяти. Следующая лемма показывает, что для доказательства существования некоторой ММТ M, принимающей язык $L(M) = L$ и имеющей сложность по памяти в пределах $s(n)$,[9] достаточно описать некоторую ММТ A, которая принимает L и использует не более чем $s(|x|)$ единиц памяти для слов языка L; при этом вычисления над словами из $L^{\complement}$ могут иметь неограниченную сложность по памяти.

Лемма 6.23. *Пусть $s : \mathbb{N}_0 \to \mathbb{N}_0$ – функция, конструктивная по памяти. Пусть M – некоторая* ММТ, *такая что* $\mathrm{Space}_M(x) \leq s(|x|)$ *для всех $x \in L(M)$. Тогда существует некоторая* ММТ A, *допускающая язык $L(A) = L(M)$, для которой*

$$\mathrm{Space}_A(n) \leq s(n)$$

– т. е. $\mathrm{Space}_A(y) \leq s(|y|)$ *для всех y над входным алфавитом машины M.*

Доказательство. Пусть M – некоторая k-ленточная-МТ для $k \in \mathbb{N}$, причём $\mathrm{Space}_M(x) \leq s(|x|)$ для всех $x \in L(M)$. Пусть B – 1-ленточная-МТ, которая вычисляет функцию s. Опишем работу $(k+1)$-ленточной-МТ A с языком $L(A) = L(M)$ и $Space_A(n) \leq s(n)$; пусть x – некоторый её вход.

1. A рассматривает x в качестве $0^{|x|}$ и моделирует на своей $(k+1)$-й рабочей ленте вычисление машины B над словом $0^{|x|}$. Моделирование заканчивается, когда на $(k+1)$-й ленте записано слово $0^{s(|x|)}$.
2. A записывает специальный символ $\# \notin \Gamma_M$ в позицию $s(|x|)$ каждой из рабочих лент.
3. A выполняет пошаговое моделирование работы машины M над входом x, используя для этого первые k рабочих лент. Если моделирование машиной M машины A потребовало перемещения головки направо за символ $\#$ рабочей ленты, то A останавливается в состоянии q_{reject}. Если же машина A смогла полностью смоделировать вычисление машины M над входом x, то A принимает x тогда и только тогда, когда M принимает x.

Очевидно, что $\mathrm{Space}_A(z) \leq s(|z|)$ для всех входов z. Кроме того, $L(A) = L(M)$.

Если $x \in L(M)$, то $\mathrm{Space}_M(x) \leq s(|x|)$. Итак, машина A моделирует полное вычисление машины M над входом x – и заканчивает это моделирование в состоянии q_{accept}. Следовательно, $x \in L(A)$.

[9] Т. е. ММТ M с языком $L = L(M)$ и сложностью по памяти $\mathrm{Space}_M(x) \leq s(|x|)$ для каждого входного слова $x \in \Sigma^*$.

Если $y \notin L(M)$, то мы выделяем следующие два случая.

Если $\mathrm{Space}_M(y) \leq s(|y|)$, то A моделирует полное вычисление машины M над y и отклоняет y.

Если же $\mathrm{Space}_M(y) > s(|y|)$, то A останавливает моделирование как только M пытается использовать более чем $s(|y|)$ ячеек рабочей ленты – при этом A останавливается в состоянии q_{reject}, и, следовательнол, $y \notin L(A)$. □

Следующая лемма 6.24 даёт утверждение, аналогичное лемме 6.23, для сложности по времени.

Лемма 6.24. *Пусть $t : \mathbb{N}_0 \to \mathbb{N}_0$ – некоторая конструктивная по времени функция. Пусть M – некоторая* ММТ, *такая что для всех $x \in L(M)$ выполнено неравенство $\mathrm{Time}_M(x) \leq t(|x|)$. Тогда существует некоторая* ММТ A, *такая что $L(A) = L(M)$ и*

$$\mathrm{Time}_A(n) \in O(t(n)).$$

Упражнение 6.25. Докажите лемму 6.24.

Леммы 6.23 и 6.24 показывают, что для некоторых конструктивной по памяти функции s и конструктивной по времени функции t классы сложности $\mathrm{SPACE}(s)$ и $\mathrm{TIME}(t)$ не меняются, если мы заменяем наши определения классов $Space_M$ и $Time_M$ – то есть

$$\mathrm{Space}_M(n) = \max\{\mathrm{Space}_M(x) \mid x \in \Sigma^n\} \text{ и}$$
$$\mathrm{Time}_M(n) = \max\{\mathrm{Time}_M(x) \mid x \in \Sigma^n\}$$

– на следующие:

$$\mathrm{Space}_M(n) = \max\{\mathrm{Space}_M(x) \mid x \in L(M) \text{ и } |x| = n\} \text{ и}$$
$$\mathrm{Time}_M(n) = \max\{\mathrm{Time}_M(x) \mid x \in L(M) \text{ и } |x| = n\}.$$

Теорема 6.26 формулирует важное соотношение между сложностью по памяти и по времени.

Теорема 6.26. *Для любой конструктивной по памяти функции s, такой что $s(n) \geq \log_2 n$, выполнено следующее:*

$$\mathrm{SPACE}(s(n)) \subseteq \bigcup_{c \in \mathbb{N}_0} \mathrm{TIME}(c^{s(n)}).$$

Доказательство. Пусть $L \in \mathrm{SPACE}(s(n))$. Согласно лемме 6.4, существует некоторая 1-ленточная-МТ $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{accept}}, q_{\mathrm{reject}})$, такая что $L = L(M)$ и $\mathrm{Space}_M(n) \leq s(n)$.

Для каждой конфигурации $C = (q, w, i, x, j)$ машины M определим **внутреннюю конфигурацию для C** как

$$\mathrm{In}(C) = (q, i, x, j).$$

Внутренняя конфигурация $\mathrm{In}(C)$ включает только те части C, которые могут изменяться в процессе вычисления; т. е. она *не* включает содержимое w входной ленты – поскольку это содержимое остаётся неизменным в течении всего вычисления над входом w.

Пусть для некоторого $n \in N$ запись $\mathrm{InConf}(n)$ обозначает множество всех возможных внутренних конфигураций, которые могут возникнуть в процессе вычислений машины M над входными словам длины n. Идея доказательства состоит в следующем: для проверки, принимает ли машина M некоторый вход w, нам достаточно промоделировать не более чем $|\mathrm{InConf}(|w|)|$ шагов вычисления машины M над словом w – поскольку каждое вычисление, длина которого превосходит $|\mathrm{InConf}(|w|)|$, является бесконечным. Используя этот факт, можно описать ММТ A, которая моделирует M и работает за время

$$O(|\mathrm{InConf}(|w|)).$$

Теперь оценим значение $|\mathrm{InConf}(n)|$. Для каждой внутренней конфигурации

$$(q, i, x, j) \in \mathrm{InConf}(n)$$

выполнены такие неравенства: $0 \le i \le n+1$, $|x| \le \mathrm{Space}_M(n) \le s(n)$ и $0 \le j \le \mathrm{Space}_M(n) \le s(n)$. Поэтому

$$\begin{aligned} |\mathrm{InConf}_M(n)| &\le |Q| \cdot (n+2) \cdot |\Gamma|^{\mathrm{Space}_M(n)} \cdot \mathrm{Space}_M(n) \\ &\le (\max\{2, |Q|, |\Gamma|\})^{4 \cdot s(n)} \\ &\le c^{s(n)} \end{aligned}$$

– для константы $c = (\max\{2, |Q|, |\Gamma|\})^4$.

Пусть w – некоторое входное слово длины n. Очевидно, что если $n \ge |\mathrm{InConf}_M(n)|$, то каждое вычисление $D = C_1, C_2, C_3, \ldots$ машины M над w должно содержать две такие конфигурации C_i и C_j, что соответствующие им внутренние конфигурации $\mathrm{In}(C_i)$ и $\mathrm{In}(C_j)$ одинаковы. Применяя определение внутренней конфигурации, мы получаем, что конфигурации C_i и C_j также одинаковы. Поскольку машина M детерминированная, вычисление

$$D = C_1, \ldots, C_{i-1}, C_i, C_{i+1}, \ldots, C_{j-1}, C_i, C_{i+1}, \ldots, C_{j-1}, C_i \ldots$$

является бесконечным – с циклом $C_i, C_{i+1}, \ldots, C_j$. Следовательно, любое конечное вычисление машины M над некоторым входом w имеет длину, не превышающую $|\mathrm{InConf}_M(|w|)|$. Поэтому длина каждого допускающего вычисления со входом длины n не превышает $|\mathrm{InConf}_M(n)|$.

Теперь опишем работу 3-ленточной-МТ A, допускающей язык $L(A) = L(M)$, у которой $\mathrm{Time}_A(n) \in O(k^{s(n)})$ для некоторой константы k. Для любого входа w машина A работает следующим образом.

1. A моделирует вычисления функции s и записывает слово $0^{s(|w|)}$ на первой рабочей ленте.
   {Поскольку 1-ленточная-МТ при вычислении s использует не более чем $s(|w|)$ ячеек рабочей ленты, существует константа d, такая что $\mathrm{Time}_B(n) \le d^{s(n)}$. Следовательно, A генерирует слово $0^{s(|n|)}$ за время $d^{s(n)}$.}
2. A записывает слово $0^{c^{s(|w|)}}$ на вторую рабочую ленту за $c^{s(|w|)}$ шагов.
   {Третью рабочую ленту машина A использует для ускорения работы.}
3. A пошагово моделирует работу машины M над словом w на первой рабочей ленте. После каждого шага моделирования машина A стирает один символ 0 на второй рабочей ленте. Если при этом на второй рабочей ленте больше нет символов 0, но процесс моделирования ещё не завершён, то A останавливается в состоянии q_{reject}.

Если вычисление машины M над словом w содержит не более чем $c^{s(|w|)}$ шагов, то моделирование достигает цели – и A принимает слово w тогда и только тогда, когда его принимает M.

Как уже было отмечено, для выполнения этапов 1 и 2 машине A требуется соответственно $O(d^{s(|w|)})$ и $O(c^{s(|w|)})$ единиц времени. Этап 3 вычисления машины A выполняется за время $O(c^{s(|w|)})$. Следовательно,

$$\mathrm{Time}_A(n) \in O((\max\{c,d\})^{s(|w|)}).$$

Если $x \in L(M)$, то длина вычисления машины M над словом x не превышает $c^{s(|w|)}$ – следовательно, после успешного моделирования A также принимает слово x. Если же $x \notin L(M)$, то A отклоняет x – и не имеет значения, является ли вычисление машины M бесконечным или отклоняющим. □

Следствие 6.27.

$$\mathrm{DLOG} \subseteq \mathrm{P} \quad \textit{и} \quad \mathrm{PSPACE} \subseteq \mathrm{EXPTIME}$$

Применяя следствия 6.17 и 6.27 мы получаем такую фундаментальную иерархию детерминированных классов сложности:

$$\mathrm{DLOG} \subseteq \mathrm{P} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPTIME}.$$

А следующие две теоремы относятся к основным результатам теории сложности. Их доказательства основаны на весьма детально разработанных версиях метода диагонализации – поэтому мы опускаем эти доказательства.

Теорема 6.28.* *Пусть s_1 и s_2 – две функции, действующие из $\mathbb{N}_0$ в $\mathbb{N}_0$, которые удовлетворяют следующим свойствам:*

- $s_2(n) \geq \log_2 n$;
- *s_2 конструктивна по памяти;*
- $s_1(n) = o(s_2(n))$.

Тогда

$$\mathrm{SPACE}(s_1) \subsetneq \mathrm{SPACE}(s_2).$$

Теорема 6.29.* *Пусть t_1 и t_2 – две функции, действующие из $\mathbb{N}_0$ в $\mathbb{N}_0$, которые удовлетворяют следующим свойствам:*

- *t_2 конструктивна по времени;*
- $t_1(n) \cdot \log_2(t_1(n)) = o(t_2(n))$.

Тогда

$$\mathrm{TIME}(t_1) \subsetneq \mathrm{TIME}(t_2).$$

Иерархии классов сложности, получаемые на основе теорем 6.28 и 6.29, показывают существование сколько угодно «трудных» проблем. Например, существуют вычислительные задачи, которые не выполняются за время $\mathrm{TIME}(2^n)$ – т. е. все алгоритмы для решения этих задач имеют временну́ю сложность, превосходящую 2^n. Таблица 6.1 подтверждает тот факт, что выполнение алгоритмов для труднорешаемых задач может превысить границы любых физических возможностей. Для функций с временно́й сложностью $10n$, $2n^2$, n^3, 2^n и $n!$ таблица 6.1 показывает, сколько операций должно

Таблица 6.1.

n / $f(n)$	10	50	100	300
$10n$	100	500	1000	3000
$2n^2$	200	5000	20 000	180 000
n^3	1000	125 000	1 000 000	27 000 000
2^n	1024	16 цифр	31 цифр	91 цифр
$n!$	$\approx 3.6 \cdot 10^6$	65 цифр	158 цифр	615 цифр

быть выполнено над входными словами длины 10, 50, 100 и 300. Некоторые числа в клетках таблицы слишком велики – и нам приходится вместо точных значений записывать их длину.

Пусть у нас есть ПК, выполняющий 10^6 операций в секунду. Тогда некоторый алгоритм A с временно́й сложностью $\mathrm{Time}_A(n) = n^3$ может вычислять результаты для входных данных, длина которых не превосходит $n = 300$, в пределах 3 секунд. Если же $\mathrm{Time}_A(n) = 2^n$, то A требует для входных слов длины 50 более чем 3 лет, а для $n = 100$ – более чем $3 \cdot 10^{15}$ лет вычислений. Сравнивая значения 2^n и $n!$ для реальных размеров входных данных – между 100 и 300 – с предполагаемым числом секунд, прошедших с момента Большого Взрыва (числом, состоящим из 21 цифры), мы получаем, что выполнение алгоритмов экспоненциальной сложности с такими данными превышает любые границы физических возможностей.

Более того, мы обращаем внимание на следующие свойства функций n^3 и 2^n. Если t – время, которое мы можем потратить на ожидание результата, то разработка более мощного компьютера – который выполняет в два раза больше команд за единицу времени – принесёт следующие улучшения.

- Для алгоритма, выполняющегося за время n^3, размер легкорешаемых входных данных может быть увеличен путём умножения на константу $\sqrt[3]{2}$, с $t^{1/3}$ до $\sqrt[3]{2} \cdot t^{1/3}$ – т. е. можно обрабатывать в $\sqrt[3]{2}$ раз более длинные размеры входных данных, чем ранее.
- Однако для алгоритма, выполняющегося за время 2^n, размер легкорешаемых входных данных может быть увеличен только на 1 бит.

Итак, алгоритмы с экспоненциальной временно́й сложностью нельзя считать практически применимыми, а алгоритмы с полиномиальной временно́й сложностью ($O(n^c)$ для малых c) – можно. Поэтому любая проблема, не принадлежащая классу $\mathrm{TIME}(2^n)$, может считаться труднорешаемой (не разрешимой на практике), а любая принадлежащая $\mathrm{TIME}(n^3)$ – легкорешаемой (разрешимой).

Как уже упоминалось, основными целями теории сложности являются

найти формальное описание класса легкорешаемых проблем (разрешимых на практике);

разработать методы классификации алгоритмических проблем – согласно их принадлежности к специальным подклассам этого класса.

Первые попытки поиска рациональной детализации интуитивного понятия легкорешаемых задач привели к следующему определению.[10] В дальнейшем любой алгоритм A с временно́й сложностью $\mathrm{Time}_A(n) \in O(n^c)$ для некоторой константы c будем называть **полиномиально-временны́м алгоритмом**. Итак,

> *проблема является разрешимой на практике (легкорешаемой), если и только если существует решающий её полиномиально-временной алгоритм. Класс сложности* P *– это класс легкорешаемых проблем принадлежности.*

Приведём две причины того, что мы объединяем рассмотрение полиномиально-временны́х вычислений с интуитивным понятием практической разрешимости.

- Первая причина – подход к практической выполнимости, основанный на нашем опыте. Мы уже отмечали, что алгоритмы, имеющие экспоненциальную сложность, не являются практически выполнимыми, а полиномиально-временны́е алгоритмы с малой степенью полиномов – являются. Но что можно сказать о таком времени выполнения, как n^{1000}? Конечно, алгоритм с временно́й сложностью n^{1000} вряд ли может быть полезен – поскольку $n^{1000} > 2^n$ для всех приемлемых длин входных данных n.

  Однако опыт доказал допустимость рассмотрения полиномиально-временны́х вычислений как легкорешаемых. Почти во всех случаях, когда для некоторой алгоритмической проблемы, прежде казавшейся трудной, были найдены полиномиально-временны́е алгоритмы – заодно было получено некоторое ключевое проникновение в суть само́й проблемы, были описаны новые идеи для её решения. Всё это дало положительные результаты и «в обратном направлении»: благодаря таким новым знаниям были разработаны новые полиномиально-временны́е алгоритмы – со степенью полинома ниже, чем раньше.

  Важно отметить, что известно только несколько исключений – примеров нетривиальных задач, для которых лучший полиномиально-временной алгоритм *не* является пригодным на практике.
- А вторая причина имеет теоретическую природу. Любое определение класса важных проблем должно быть столь «жёстким», чтобы описанный класс являлся бы инвариантом относительно всех приемлемых вычислительных моделей. Мы не можем допустить того, чтобы проблема была легкорешаемой для некоторого языка программирования (например, для Явы) – но не для ММТ. А подобная ситуация могла бы возникнуть, если бы мы определили в качестве множества легко решаемых задач $\mathrm{TIME}(n^6)$. Ключевым моментом является то, что относительно всех приемлемых вычислительных моделей класс P и является жёстким – в описанном здесь смысле.

  Доказательство последнего утверждения основано на определении полиномиально-временно́й сводимости для вычислительных моделей. Вычислительная модель $\mathcal{A}$ **полиномиально сводима по времени** к вычислительной модели $\mathcal{B}$, если существует некоторый полином p, такой что для каждого алгоритма $A \in \mathcal{A}$ существует алгоритм $B \in \mathcal{B}$, который решает ту же самую проблему, что и A, причём

$$\mathrm{Time}_B(n) \in O(p(\mathrm{Time}_A(n))).$$

[10] В настоящее время такая детализация алгоритмической легкорешаемости не принята в подобной ограниченной форме – мы приведём более подробное обсуждение этой темы в следующей главе.

Если рассматривать $\mathcal{A}$ как класс машин Тьюринга, а $\mathcal{B}$ – как класс многоленточных машин Тьюринга, то возможность полиномиально-временно́й сводимости $\mathcal{A}$ к $\mathcal{B}$ с полиномом $p(n) = n$ была показана в лемме 4.11.

Фактически, для всех пар приемлемых вычислительных моделей известные полиномиально-временны́е сводимости работают с функцией $p(n) \in O(n^3)$. Итак, если мы разработаем некоторый полиномиально-временной алгоритм для некоторой проблемы U в виде программы на Яве – то существует и полиномиально-временной алгоритм решения U в любом приемлемом формализме. С другой стороны, если доказать, что не существует машины Тьюринга, допускающей заданный язык L и работающей за полиномиальное время, – то можно быть уверенным, что не существует и полиномиально-временно́й компьютерной программы, распознающей L. Итак, описанное здесь «свойство жёсткости» очень важно – оно требуется для любой обоснованной спецификации класса легкорешаемых задач.

Упражнение 6.30. Примените лемму 4.13 для доказательства следующего утверждения. Для каждой ММТ A, имеющей временну́ю сложность $\mathrm{Time}_A(n) \geq n$, существует эквивалентная МТ B, такая что

$$\mathrm{Time}_B(n) \in O((\mathrm{Time}_A(n))^2).$$

6.4 Недетерминированные меры сложности

Недетерминированные машины Тьюринга могут иметь множество различных вычислений над одним входом[11] – и сложность этих вычислений может быть существенно различной. Как же определить сложность недетерминированной машины Тьюринга (недетерминированного алгоритма) M над некоторым входом w? При недетерминированной работе мы придерживаемся «оптимистического» взгляда – считаем, что недетерминированная машина всегда делает «лучший» выбор из всех возможных. *Лучший выбор в данном случае предполагает не только то, что он в дальнейшем обеспечит корректное решение – но также и то, что сложность вычислений в итоге получится минимально возможной.* Интерпретируя недетерминизм этим оптимистическим способом, можно определить сложность работы недетерминированного алгоритма M над входом w как сложность наиболее эффективного вычисления M над w, дающего верный результат. Поэтому далее для проблем принадлежности (проблем распознавания языков) мы будем рассматривать вычисления сложности только над теми входами, которые принадлежат языку.

Определение 6.31. *Пусть M – некоторая недетерминированная машина Тьюринга, либо недетерминированная многоленточная машина Тьюринга. Пусть $x \in L(M) \subseteq \Sigma^*$.*

Сложность по времени M над x, $\mathbf{Time_M(x)}$ *– это длина самого короткого допускающего вычисления машины M над словом x.*

Сложность по времени M *– это функция* $\mathrm{Time}_M : \mathbb{N}_0 \to \mathbb{N}_0$, *определённая как*

$$\mathbf{Time_M(n)} = \max\{\mathrm{Time}_M(x) \mid x \in L(M) \cap \Sigma^n\}.$$

[11] Даже бесконечное (но счётное) множество вычислений.

Пусть $C = C_1, C_2, \ldots, C_m$ *– некоторое допускающее вычисление машины* M *над словом* x, *а* $\mathrm{Space}_M(C_i)$ *– сложность по памяти конфигурации* C_i. *Определим*

$$\mathbf{Space_M(C)} = \max\{\mathrm{Space}_M(C_i) \mid i = 1, 2, \ldots, m\}.$$

Сложность по памяти машины M над x –

$$\mathbf{Space_M(x)} = \min\{\mathrm{Space}_M(C) \mid C \text{ – } \textit{допускающее вычисление } M \textit{ над } x\}.$$

Сложность по памяти M – *функция* $\mathrm{Space}_M : \mathbb{N}_0 \to \mathbb{N}_0$, *определённая как*

$$\mathbf{Space_M(n)} = \max\{\mathrm{Space}_M(x) \mid x \in L(M) \cap \Sigma^n\}.$$

Определение 6.32. *Для всех функций* $f, g : \mathbb{N}_0 \to \mathbb{R}^+$ *определим*

$$\begin{aligned}
\mathbf{NTIME(f)} &= \{L(M) \mid M \text{ – } \textit{недетерминированная } \text{MMT}, \\
&\qquad \textit{причём } Time_M(n) \in O(f(n))\}, \\
\mathbf{NSPACE(g)} &= \{L(M) \mid M \text{ – } \textit{недетерминированная } \text{MMT} \\
&\qquad \textit{причём } \mathrm{Space}_M(n) \in O(g(n))\}, \\
\mathbf{NLOG} &= \mathrm{NSPACE}(\log_2 n), \\
\mathbf{NP} &= \bigcup_{c \in \mathbb{N}_0} \mathrm{NTIME}(n^c), \\
\mathbf{NPSPACE} &= \bigcup_{c \in \mathbb{N}_0} \mathrm{NSPACE}(n^c).
\end{aligned}$$

Рассмотрим отношения между сложностью по времени и по памяти для недетерминированного случая. Отметим, что эти отношения – такие же, как и в детерминированном случае, т. е. аналогичны лемме 6.16 и теореме 6.26.

Лемма 6.33. *Для всех пар конструктивных по памяти функций* t *и* s:

(a) $\mathrm{NTIME}(t) \subseteq \mathrm{NSPACE}(t)$,
(b) $\mathrm{NSPACE}(s) \subseteq \mathrm{NTIME}(c^{s(n)})$ *для некоторой константы* c.

Доказательство. Сначала докажем (a), а потом (b).

(a) Предположим, что $L \in \mathrm{NTIME}(t)$ – т. е. существует недетерминированная ММТ M, такая что для некоторой константы d и всех достаточно больших n выполнены условия

$$L(M) = L \quad \text{и} \quad \mathrm{Time}_M(n) \leq d \cdot t(n).$$

Отсюда для каждого достаточно длинного слова $x \in L(M)$ существует допускающее вычисление C_x машины M над x, длина которого не превосходит $d \cdot t(|x|)$. Поскольку M за $d \cdot t(n)$ шагов может посетить не более, чем $d \cdot t(n)$ ячеек ленты, мы получаем неравенство

$$\mathrm{Space}_M(C_x) \leq d \cdot t(|x|).$$

Итак,

$$\mathrm{Space}_M(n) \leq d \cdot t(n)$$

для всех достаточно больших n – поэтому $L \in \mathrm{NSPACE}(t)$.

(b) Пусть $L \in \mathrm{NSPACE}(s)$, а s – некоторая функция, конструктивная по памяти. Тогда существует недетерминированная ММТ A, такая что для некоторых константы d и всех достаточно длинных слов $x \in L(A) = L$ выполнены условия

$$L = L(M) \quad \text{и} \quad \mathrm{Space}_A(x) \leq d \cdot s(|x|).$$

Это означает, что для всех достаточно длинных слов $x \in L(A)$ существует допускающее вычисление C_x, такое что

$$\mathrm{Space}_A(C_x) \leq d \cdot s(|x|).$$

Пусть C_x – наиболее короткое допускающее вычисление машины A над словом x, обладающее сформулированным свойством. Используя тот же аргумент, что и в доказательстве теоремы 6.26, мы получаем следующий факт: для всех достаточно длинных $x \in L(A)$ существует константа k, такая что длина вычисления C_x не превосходит значения

$$|\mathrm{InConf}_A(|x|)| \leq k^{d \cdot s(|x|)}.$$

Если бы длина вычисления C_x была больше, чем $|\mathrm{InConf}_A(|x|)|$, то существовали бы $i, j \in \mathbb{N}$, $i \neq j$, такие что

$$C_x = C_1, C_2, \ldots, C_{i-1}, C_i, C_{i+1}, \ldots, C_j, C_{j+1}, \ldots, C_m$$

– причём C_i и C_j были бы одинаковы. Тогда последовательность

$$C'_x = C_1, C_2, \ldots, C_{i-1}, C_i, C_{j+1}, \ldots, C_m$$

также являлась бы допускающим вычислением машины A над словом x. При этом, поскольку C'_x короче, чем C_x, мы получили бы противоречие с предположением о том, что C_x – наиболее короткое допускающее вычисление машины A над словом x, такое что $\mathrm{Space}_A(C_x) \leq d \cdot s(|x|)$.
Следовательно, для каждого достаточно длинного слова $x \in L(A)$ существует допускающее вычисление машины A, длина которого не превышает значения

$$k^{d \cdot s(|x|)} = c^{s(|x|)}$$

– где $c = k^d$ для выбранных ранее констант k и d. Итак,

$$\mathrm{Time}_A(n) \in O(c^{s(n)}),$$

и $L \in \mathrm{NTIME}(c^{s(n)})$. □

Упражнение 6.34. Пусть M – некоторая недетерминированная ММТ, такая что $\mathrm{Time}_M(n) \leq t(n)$ для некоторой конструктивной по времени функции t. Докажите, что существует недетерминированная ММТ A, для которой:

- $L(A) = L(M)$;
- существует константа d, такая что для любого $w \in \Sigma^*$ длина каждого вычисления машины A над w не превосходит $d \cdot t(|w|)$.

Следующая теорема показывает фундаментальные соотношения между различными вариантами измерения сложности в детерминированном и недетерминированном случаях.

Теорема 6.35. *Для каждых функции $t : \mathbb{N}_0 \to \mathbb{R}^+$ и конструктивной по памяти и по времени функции $s : \mathbb{N}_0 \to \mathbb{N}_0$, такой что $s(n) \geq \log_2 n$:*

(a) $\mathrm{TIME}(t) \subseteq \mathrm{NTIME}(t)$;
(b) $\mathrm{SPACE}(t) \subseteq \mathrm{NSPACE}(t)$;
(c) $\mathrm{NTIME}(s(n)) \subseteq \mathrm{SPACE}(s(n)) \subseteq \bigcup\limits_{c \in \mathbb{N}_0} \mathrm{TIME}(c^{s(n)})$.

Доказательство. Утверждения (a) и (b) очевидны – поскольку каждая ММТ является одновременно недетерминированной ММТ. Включение

$$\mathrm{SPACE}(s(n)) \subseteq \bigcup_{c \in \mathbb{N}_0} \mathrm{TIME}(c^{s(n)})$$

было доказано в теореме 6.26. Поэтому для доказательства утверждения (c) достаточно показать, что

$$\mathrm{NTIME}(s(n)) \subseteq \mathrm{SPACE}(s(n)).$$

Пусть $L \in \mathrm{NTIME}(s(n))$, т. е. существует недетерминированная k-ленточная-МТ $M = (Q, \Sigma, \Gamma, \delta_M, q_0, q_{\mathrm{accept}}, q_{\mathrm{reject}})$, такая что

$$L = L(M) \text{ и } \mathrm{Time}_M(n) \in O(s(n)).$$

Очевидно, что значение

$$r = r_M = \max\{|\delta_M(U)| \mid U = (q, a, b_1, \ldots, b_k) \in Q \times (\Sigma \cup \{¢, \$\}) \times \Gamma^k\}$$

является верхней границей числа различных возможных действий машины M в любой конфигурации. Пусть $T_{M,x}$ – дерево вычислений машины M над входом $x \in \Sigma^*$. Если упорядочить недетерминированные альтернативы δ_M для каждого набора аргументов U, то можно пометить их числами $1, 2, \ldots, r$, соответствующими дугам дерева $T_{M,x}$ (рис. 6.1). Таким путём мы для любого вычисления машины M над x с l недетерминированными вариантами однозначно определяем слово

$$z = z_1 z_2 \ldots z_l \in \{1, 2, \ldots, r\}^*.$$

Итак, на основе заданных машины M и входного слова x последовательность z:

- либо однозначно определяет префикс некоторых вычислений M над словом x (например, слово $z = r322$ для дерева $T_{M,x}$, приведённого на рис. 6.1, определяет такой префикс вычислений, в котором мы выбираем r-ю возможность для первого выбора, 3-ю возможность для второго выбора, 2-ю возможность для третьего выбора и 2-ю возможность для четвёртого выбора);
- либо является бессмыслицей (например, никакое вычисление из приведённых на рис. 6.1 не соответствует никакому слову вида $z = 24\ldots$ – поскольку в данном случае не существует 4-й возможности для второго выбора).

Согласно упражнению 6.34, существует такая константы d, что все вычисления машины M над входом w имеют длину, не превышающую $d \cdot s(|w|)$. Следовательно, никакое вычисление машины M над w не использует более чем $d \cdot s(|w|)$ единиц памяти.

Теперь опишем $(k+2)$-ленточную машину Тьюринга A, моделирующую все вычисления машины M, длина которых не превосходит $|\mathrm{InConf}_M(n)|$. Для каждого входа $w \in \Sigma^*$ машина A работает следующим образом.

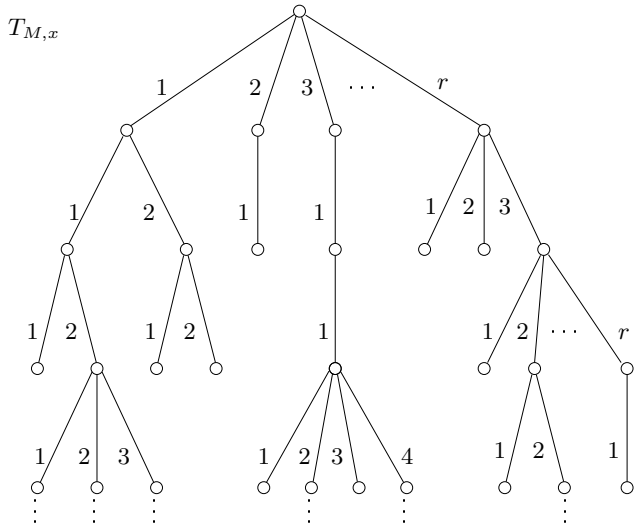


Рис. 6.1.

1. A записывает слово $0^{s(|w|)}$ на $(k+2)$-ю ленту.
2. A записывает слово $0^{d\cdot s(|w|)}$ на $(k+1)$-ю ленту и стирает содержимое $(k+2)$-й ленты.
3. A последовательно генерирует все слова $z \in \{1, 2, \ldots, r_M\}^*$, длина которых не превосходит $d \cdot s(|w|)$. Эти слова генерируются в каноническом порядке на $(k+2)$-й ленте. Для каждого сгенерированного слова z машина A с помощью своих k первых рабочих лент моделирует соответствующее вычисление машины M над словом w – если такое вычисление существует. Если M достигает своего допускающего состояния в каком-либо из смоделированных вычислений, то A допускает w. Если же все смоделированные вычисления машины M не являются допускающими (т. е. ни в одном из этих вычислений не достигается состояние q_{accept}), то M отклоняет w.

Длина любого вычисления машины M на входах длины n ограничена значением $d \cdot s(n)$, поэтому для каждого такого входа w машина A проверяет все вычисления машины M над w – следовательно, $L(A) = L(M)$. Поскольку $\text{Space}_M(n) \leq \text{Time}_M(n)$, мы получаем, что $\text{Space}_M(n) \leq d \cdot s(n)$ – и поэтому машина A никогда не использует более чем $\text{Space}_M(n) \leq d \cdot s(n)$ ячеек на первых k рабочих лентах. А на $(k+1)$-й ленте для слова $O^{d\cdot s(|w|)}$ используется в точности $d \cdot s(n)$ ячеек. То же самое выполняется для $(k+2)$-й ленты, которая во время моделирования всегда содержит одно слово языка $\{1, 2, \ldots, r\}^*$, имеющее длину, не превосходящую $d \cdot s(n)$. Итак,

$$\text{Space}_A(n) \leq d \cdot s(n).$$

Последний факт заканчивает доказательство теоремы 6.35. □

Следствие 6.36.

$$\text{NP} \subseteq \text{PSPACE}$$

Упражнение 6.37. Проанализируйте временну́ю сложность $(k+2)$-ленточной машины Тьюринга A, описанной в доказательстве теоремы 6.35.

К сожалению, неизвестно более эффективного детерминированного моделирования недетерминированных алгоритмов кроме *последовательного рассмотрения всех вычислений* недетерминированного алгоритма с заданным входом. Мы это уже рассматривали выше: в одном из случаев доказательства теоремы 6.35 – где мы выполняли поиск в ширину в дереве вычислений, а также в теореме 4.27 – при поиске в глубину. Оба эти варианта моделирования требуют, вообще говоря, экспоненциального времени работы – поскольку количество шагов вычисления может экспоненциально зависеть от глубины дерева вычислений. В настоящее время большинство исследователей *предполагают*, что не существует возможности моделирования недетерминизма с ростом временно́й сложности, полиномиально зависящим от длины входа, – но никто не может *доказать*, что такого моделирования действительно не существует.[12]

Упражнение 6.38. Оцените временну́ю сложность детерминированной ММТ, рассмотренной в доказательстве теоремы 4.27, – т. е. оцените временну́ю сложность поиска в ширину детерминированного моделирования недетерминизма.

Следующая теорема знакомит с наиболее эффективным из известных видом моделирования недетерминированного пространства событий за детерминированное время. Отметим, что это моделирование является столь же эффективным, как и моделирование детерминированного пространства состояний за детерминированное время в теореме 6.26.

Теорема 6.39.* *Для каждой конструктивной по памяти функции s, такой что $s(n) \geq \log_2 n$, выполнено следующее:*

$$\mathrm{NSPACE}(s(n)) \subseteq \bigcup_{c \in \mathbb{N}_0} \mathrm{TIME}(c^{s(n)}).$$

Доказательство. Пусть M – недетерминированная ММТ, у которой

$$L(M) = L \quad \text{и} \quad \mathrm{Space}_M(n) \in O(s(n)).$$

Без ограничения общности мы можем предполагать следующее.

(a) Существует константа d, такая что для каждого входа w сложность по памяти всех вычислений машины M над w не превосходит

$$d \cdot s(n).$$

(b) Для каждого входа $w \in L(M)$ у машины M имеется единственная допускающая конфигурация

$$(q_{\mathrm{accept}}, w, 0, \lambda, \ldots, \lambda, 0)$$

– т. е. перед достижением состояния q_{accept} машина M стирает содержание всех своих рабочих лент и устанавливает все свои головки на левый граничный маркер ¢.

[12] Эта проблема подробно обсуждается в следующих двух главах.

Согласно (a) и лемме 6.33, существует некоторая константа c, такая что для каждого входа w число всех различных конфигураций с w на входной ленте не превышает значения

$$|\mathrm{InConf}(|w|)| \leq c^{s(|w|)}.$$

Расположим все эти конфигурации в каноническом порядке – в последовательности

$$C_0, C_1, \ldots, C_{|\mathrm{InConf}(|w|)|}.$$

Опишем (детерминированную) ММТ A, допускающую язык $L(A) = L$. Для каждого входа w машина A работает следующим образом.

1. A создаёт матрицу смежности $M(w)$ ориентированного графа $G(w)$, чьи вершины являются $|\mathrm{InConf}(|w|)|$ конфигурациями с w на входной ленте, а сложность по памяти ограничена значением $d \cdot s(n)$. При этом в орграфе $G(w)$ дуга из C_i в C_j присутствует в том и только том случае, когда $C_i \vdash_M C_j$ – т. е. C_j может быть достигнута из C_i за один шаг вычисления машины M.
2. Пусть C_k – единственная допускающая конфигурация машины M со словом w на входной ленте. Пусть C_0 – инициальная конфигурация машины M над w. Очевидно, что M допускает w если и только если существует путь из C_0 в C_k в графе $G(w)$. Используя какой-нибудь стандартный подход (например, алгоритм Флойда), A проверяет, действительно ли вершина (конфигурация) C_k достижима из вершины C_0. Если достижима, то A допускает w, иначе отклоняет.

Равенство $L(A) = L(M)$ очевидно.

Теперь проанализируем временну́ю сложность алгоритма A. Чтобы создать $M(w)$, алгоритм A должен вычислить

$$|\mathrm{InConf}(|w|)| \cdot |\mathrm{InConf}(|w|)| \leq c^{d \cdot s(|w|)} \cdot c^{d \cdot s(|w|)} \leq c^{2d \cdot s(|w|)}$$

элементов m_{ij} матрицы $M(w)$. Чтобы определить каждый элемент m_{ij}, мы должны сгенерировать вершины (конфигурации) C_i и C_j. Любая конфигурация может быть сгенерирована за время

$$d \cdot s(|w|) \cdot |\mathrm{InConf}(|w|)| \leq c^{2d \cdot s(|w|)}.$$

За время $2d \cdot s(|w|)$ можно проверить, можно ли получить C_j из C_i за один шаг вычислений M. Итак, временна́я сложность части 1 вычислений машины A не превосходит значения

$$c^{2d \cdot s(|w|)} \cdot (2c^{2d \cdot s(|w|)} + 2d \cdot s(|w|)) \leq c^{12d \cdot s(|w|)}.$$

Проверка существования пути из C_0 в C_k in $G(w)$ может быть произведена в пределах полиномиального времени (относительно значения $|\mathrm{InConf}(|w|)|$): многоленточная машина Тьюринга может выполнить эту задачу за $O(|\mathrm{InConf}(|w|)|^4)$ шагов. И, поскольку

$$(c^{d \cdot s(|w|)})^4 = c^{4d \cdot s(|w|)},$$

мы получаем, что

$$\mathrm{Time}_A(n) \in O(c^{12 \cdot d \cdot s(n)}).$$

□

Следствие 6.40.

$$\mathrm{NLOG} \subseteq \mathrm{P} \quad и \quad \mathrm{NPSPACE} \subseteq \mathrm{EXPTIME}.$$

Немного более сложный вариант поиска в орграфе $G(w)$ всех возможных конфигураций над словом w даёт следующий результат.[13]

Теорема 6.41 (Савича).* *Пусть s – некоторая функция, конструктивная по памяти, такая что $s(n) \geq \log_2 n$. Тогда*

$$\mathrm{NSPACE}(s(n)) \subseteq \mathrm{SPACE}(s(n)^2).$$

Следствие 6.42.

$$\mathrm{PSPACE} = \mathrm{NPSPACE}.$$

Нам до сих пор неизвестно, существует ли улучшение какого-либо из описанных выше вариантов детерминированного моделирования недетерминированных вычислений – т. е. варианты, которые давали бы более эффективное моделирование. Следующая фундаментальная **иерархия классов сложности** последовательных вычислений

$$\mathrm{DLOG} \subseteq \mathrm{NLOG} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPTIME}$$

фактически является последовательностью результатов описанного выше моделирования.

При этом для каждого из этих включений остаётся открытым вопрос, является ли оно собственным. Однако в приведённой последовательности (иерархии классов сложности) некоторые включения обязательно должны быть собственными – поскольку собственными являются включения

$$\mathrm{DLOG} \subsetneq \mathrm{PSPACE} \quad и \quad \mathrm{P} \subsetneq \mathrm{EXPTIME}.$$

Два последних условия являются прямыми следствиями теорем этого раздела – и могут быть названы теоремами об иерархии. В течение последних 30 лет проверка истинности включений фундаментальной иерархии классов сложности была и остаётся центральной нерешённой проблемой теоретической информатики.

6.5 Класс NP и проверка доказательств

Центральная открытая проблема фундаментальной иерархии сложности – это отношение между классами P и NP. Вопрос

$$\text{верно ли, что } \mathrm{P} = \mathrm{NP} \ ? \qquad \text{или же } \mathrm{P} \subsetneq \mathrm{NP} \ ?$$

является самой известной открытой проблемой теоретической информатики – и даже всей математики вообще!

Существует много причин для столь большого интереса. Одна из них заключается в том, что полиномиальное время работы алгоритма – а, следовательно, и класс P,

[13] Этот вариант поиска производится без создания графа $G(w)$ – потому что для этого потребовалось бы слишком много памяти.

– связаны с практической разрешимостью. С другой стороны, мы знаем более 4000 алгоритмических проблем в классе NP, таких что ни для одной из них нет (детерминированных) алгоритмов, работающих за полиномиальное время. И про каждую из эти проблем нам хотелось бы знать, где именно она находится: в P или NP – P (т. е. за пределами P). Другая причина нашего интереса к изучению классов P и NP связана с формальным доказательством – как фундаментальным понятием математики. В некоторых приложениях временна́я сложность детерминированных вычислений соответствует сложности алгоритмического *создания* (алгоритмического поиска) математического доказательства данной теоремы – в то время как временна́я сложность недетерминированных вычислений соответствует сложности алгоритмической *проверки* правильности заранее заданного доказательства.

Поэтому сравнение классов P и NP эквивалентно вопросу

«легче» ли проверка данного формального доказательства, чем его создание?

А главная цель этого раздела – показать взаимосвязь между классом NP и алгоритмической проверкой доказательств за полиномиальное время.

Связь между вычислениями и доказательствами опишем сначала интуитивно. Пусть C – некоторое допускающее вычисление машины Тьюринга M над входом x; при этом мы можем рассматривать C в качестве доказательства утверждения $x \in L(M)$. Аналогично, некоторое отклоняющее вычисление (детерминированной) машины Тьюринга M над входом x можно рассматривать как доказательство утверждения $x \notin L(M)$.

Приведённая интерпретация близка к классическому взгляду на математические доказательства. А именно, если рассматривать L как язык, который включает все верные теоремы некоторой математической теории,[14] – то доказательство того, что $x \in L(M)$, является доказательством истинности (справедливости, достоверности) утверждения x,[15] а доказательство того, что $x \notin L(M)$, – доказательством ложности x.

В качестве примера рассмотрим язык $L = \mathrm{SAT}$, где

$$\mathrm{SAT} = \{x \in (\Sigma_{\mathrm{logic}})^* \mid x \text{ кодирует выполнимую формулу в КНФ}\}.$$

Тогда утверждения «$\Phi \in \mathrm{SAT}$» и «формула Φ выполнима» эквивалентны.

Получим соотношение между недетерминированными вычислениями и проверкой корректности доказательств. Как правило, недетерминированное вычисление начинается с предположения (угадывания варианта) и продолжается проверкой правильности сделанного предположения. При этом угадывание варианта обычно производится с помощью некоторого недетерминированного алгоритма, а проверка варианта – с помощью детерминированного; некоторые примеры уже были приведены ранее. В рассматриваемом нами случае предположением является условие $x \in L(M)$.

Проиллюстрируем последнее рассуждение с помощью проблемы выполнимости. Рассмотрим недетерминированную машину Тьюринга M, языком которой является $L(M) = \mathrm{SAT}$. Для некоторой формулы Φ над n Булевыми переменными $x_1, \ldots, x_n$ машина M угадывает какой-либо из возможных наборов $\alpha_1, \ldots, \alpha_n$ значений переменных $x_1, \ldots, x_n$ (иными словами – какую-либо из возможных подстановок); это происходит

[14] Точнее – представление всех таких теорем.

[15] Некоторое допускающее вычисление машины M над входом x и является доказательством данного факта.

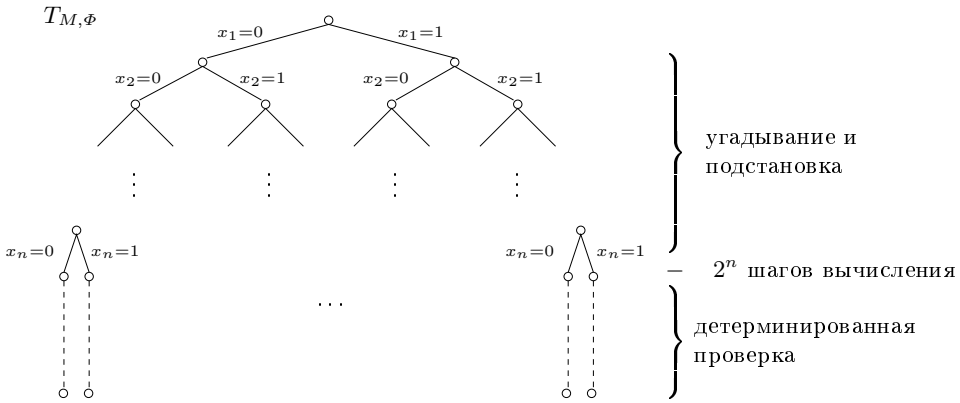


Рис. 6.2.

на первых n шагах вычисления. После этого M вычисляет значение $\Phi(\alpha_1, \ldots, \alpha_n)$ – для проверки того, соответствует ли предположенная (угаданная) подстановка $\alpha_1, \ldots, \alpha_n$ формуле Φ (рис. 6.2). Если последовательность значений $\alpha_1, \ldots, \alpha_n$ удовлетворяет формуле Φ, и при этом мы уже знаем доказательство для последовательности $\alpha_1, \ldots, \alpha_n$ – то мы можем эффективно создать доказательство утверждения «формула Φ выполнима»; это доказательство является вычислением значения формулы Φ для значений аргументов $\alpha_1, \ldots, \alpha_n$. Вследствие этого[16] последовательность $\alpha_1, \ldots, \alpha_n$ называется свидетельством – или доказательством утверждения «формула Φ выполнима». Временна́я сложность машины M является суммой временно́й сложностью угадывания доказательства и временно́й сложностью его проверки. Поскольку сложность угадывания доказательства не превышает длину входа, сложность машины M асимптотически эквивалентна сложности проверки доказательства.

Наша цель – показать, что на основе любой недетерминированной машины Тьюринга, работающей за полиномиальное время и принимающей язык L, можно построить эквивалентную НМТ, которая сначала недетерминированно выбирает (угадывает) некоторого кандидата w для доказательства утверждения $x \in L$, а затем детерминированно проверяет, на самом ли деле w является таким доказательством. Таким образом, временну́ю сложность недетерминированных алгоритмов можно свести к временно́й сложности проверки детерминированных доказательств. Для этого нам необходимо следующее формальное понятие.

Определение 6.43. *Пусть $L \subseteq \Sigma^*$ – некоторый язык, а $p : \mathbb{N}_0 \to \mathbb{N}_0$ – некоторое отображение. Будем говорить, что* ММТ *(алгоритм) A, работающая над входами из множества $\Sigma^* \times (\Sigma_{\mathrm{bool}})^*$, является* **$p$-верификатором для L**, *и записывать этот факт в виде* $\mathbf{V(A) = L}$, *если выполнены следующие 3 условия.*

- $\mathrm{Time}_A(w, x) \leq p(|w|)$ *для каждого входа* $(w, x) \in \Sigma^* \times (\Sigma_{\mathrm{bool}})^*$.
- *Для каждого* $w \in L$ *существует* $x \in (\Sigma_{\mathrm{bool}})^*$, *такое что*

$$|x| \leq p(|w|) \text{ и } (w, x) \in L(A) \text{ (т. е. } A \text{ принимает } (w, x)).$$

[16] Т. е. вследствие того, что мы *уже* знаем конкретную последовательность значений $\alpha_1, \ldots, \alpha_n$. *(Прим. перев.)*

Слово x при этом называется **доказательством** *утверждения $w \in L$.*

- *Для всех $y \notin L$ и $z \in (\Sigma_{\text{bool}})^*$ выполнено условие $(y, z) \notin L(A)$.*

Если $p(n) \in O(n^k)$ для некоторого натурального k, то мы будем говорить, что A – **полиномиально-временной верификатор**. *Определим* **класс языков, проверяемых за полиномиальное время**, *как*

$$\mathbf{VP} = \{V(A) \mid A - \textit{полиномиально-временной верификатор}\}.$$

Заметим, что для некоторого p-верификатора A языки $L(A)$ и $V(A)$ различны. Согласно определению 6.43,

$$V(A) = \{w \in \Sigma^* \mid \exists x \in (\Sigma_{\text{bool}})^* : \ |x| \leq p(|w|), \ (w, x) \in L(A)\}.$$

Итак, верификатор A для некоторого языка L представляет собой детерминированный алгоритм, который для каждого входа (w, x) проверяет, является ли x доказательством утверждения $w \in L$. Слово w принадлежит $V(A)$ если существует доказательство x утверждения $w \in L$, имеющую длину $|x| \leq p(|w|)$.

Полиномиально-временной верификатор A для SAT может быть описан следующим образом. Для любого входа (w, x) алгоритм A сначала проверяет, является ли w кодом некоторой формулы Φ_w в КНФ. Если нет, то A отклоняет вход. В противном случае A подсчитывает число Булевых переменных в Φ_w (пусть n), и проверяет, не превосходит ли длина слова $x \in \{0,1\}^*$ этого значения. Если $|x| < n$, то A отклоняет вход. Иначе $|x| \geq n$ – и A рассматривает первые n бит слова x в качестве подстановки Булевых переменных, входящих в формулу Φ_w.

Очевидно, что A принимает пару (w, x) тогда и только тогда, когда описанная подстановка удовлетворяет формуле Φ_w. При этом A действительно является полиномиально-временны́м верификатором для SAT – поскольку для любого w существует доказательство x, такое что

$$|x| \leq |w| \quad \text{и} \quad (w, x) \in L(A),$$

и можно эффективно вычислить значение Булевой формулы в КНФ для заданных значений всех её переменных (заданной подстановки).[17]

k-клика графа G с n вершинами, где $k \leq n$, есть полный подграф графа G, имеющий k вершин. Пусть

$$\mathbf{CLIQUE} = \{x\#y \mid x, y \in \{0,1\}^*, \ x \text{ кодирует граф } G_x, \\ \text{содержащий } \mathit{Number}(y)\text{-клику}\}.$$

Полиномиально-временной верификатор B для CLIQUE может быть описан следующим образом. Для любого входа (w, z), алгоритм B проверяет, представимо ли слово w в виде $w = x\#y$, где x – представление графа G_x, а $y \in (\Sigma_{\text{bool}})^*$. Если не представимо, то B отклоняет (w, z). Иначе B сначала подсчитывает число n вершин графа G_x.

Далее, пусть $v_1, \ldots, v_n$ – все вершины графа G_x. Теперь B проверяет, выполнены ли одновременно условия

[17] Многоленточная машина Тьюринга может выполнить это за квадратичное время. Более того, при использовании удобных структур данных мы можем получить даже линейное время работы.

$$Number(y) \leq n \quad \text{и} \quad |z| \geq \lceil \log_2 n \rceil \cdot Number(y).$$

Если нет, то B отклоняет свой вход (w, z). Иначе B интерпретирует префикс слова z длины $\lceil \log_2 n \rceil \cdot Number(y)$ как код $Number(y)$ чисел множества $\{1, 2, \ldots, n\}$. B проверяет, действительно ли все эти $Number(y)$ чисел попарно различны. Если нет, то B также отклоняет свой вход.

Пусть $i_1, i_2, \ldots, i_{Number(y)}$ – рассматриваемые $Number(y)$ различных натуральных чисел. Тогда B проверяет, образуют ли вершины $v_{i_1}, v_{i_2}, \ldots, v_{i_{Number(y)}}$ полный подграф графа G_x. При этом B принимает вход (w, z) тогда и только тогда, когда последняя проверка завершилась успехом.

Упражнение 6.44. Разработайте полиномиально-временной верификатор для языка

$$\text{COMPOSITE} = \{x \in (\Sigma_{\text{bool}})^* \mid Number(x) - \text{составное число}\}.$$

Упражнение 6.45. Разработайте полиномиально-временной верификатор и полиномиально-временну́ю НМТ для языка HC – т. е. для проблемы Гамильтонова цикла, рассматривавшейся в разделе 2.3.

Следующая теорема показывает, что каждая полиномиально-временна́я НМТ может быть преобразована в эквивалентную[18] полиномиально-временну́ю НМТ – такую, которая принимает все свои недетерминированные решения на самой ранней стадии своих вычислений, а затем проверяет правильность своих решений.

Теорема 6.46.

$$\text{VP} = \text{NP}.$$

Доказательство. Установим это равенство, доказав включения $\text{NP} \subseteq \text{VP}$ и $\text{VP} \subseteq \text{NP}$.

$\supseteq$ Во-первых, докажем включение $\text{NP} \subseteq \text{VP}$. Пусть $L \in \text{NP}$, $L \subseteq \Sigma^*$ для некоторого алфавита Σ. Тогда для некоторого $k \in \mathbb{N}$ существует полиномиально-временна́я НМТ M с языком $L = L(M)$ и временно́й сложностью $\text{Time}_M(n) \in O(n^k)$. Без ограничения общности мы можем предполагать, что при любых значениях аргументов своей функции переходов машина M имеет выбор не более чем из двух возможностей. Опишем верификатор A, работающий для каждого входа $(x, c) \in \Sigma^* \times (\Sigma_{\text{bool}})^*$ следующим образом.

1. A интерпретирует слово c в качестве «навигатора» для проводимого ею моделирования недетерминированного выбора машины M – т. е. A пошагово моделирует её работу над входом w. Если M имеет выбор из двух возможностей, то A выбирает первую из них в том случае, когда очередной бит слова c равен 0, и вторую – когда он равен 1. Таким путём A моделирует ровно одно из вычислений машины M над x.
   {Эта стратегия моделирования совпадает с использовавшейся в доказательстве теоремы 6.35 – где недетерминизм моделировался путём поиска в глубину в дереве вычислений.}
2. Если M всё ещё нуждается в выборе, а A уже использовала все биты слова c, то A останавливается, отклоняя вход (x, c).

[18] Аналогично другим формализмам, недетерминированные машины Тьюринга называются эквивалентными, если они принимают один и тот же язык.

3. Если машина A достигает успеха в моделировании всего вычисления машины M над словом x, то A останавливается и принимает пару (x, c) тогда и только тогда, когда M также принимает слово x в своём вычислении.
   {Отметим ещё раз, что это вычисление машины M моделируется машиной A с помощью слова c.}

Теперь мы должны показать, что A – полиномиально-временной верификатор, причём

$$L(A) = L(M).$$

Если $x \in L(M)$, то самое короткое допускающее вычисления $C_{M,x}$ машины M над x выполняется за время $O(|x|^k)$. Следовательно, существует доказательство (навигатор) c, которое определяет $C_{M,x}$, причём $|c| \leq |C_{M,x}| \in O(|x|^k)$. Поскольку A моделирует вычисления $C_{M,x}$ пошагово, вычисления машины A над парой (x, c) также выполняются за время $O(|x|^k)$.

Если $x \notin L(M)$, то не существует ни одного допускающего вычисления машины M над x, поэтому A отклоняет вход (x, d) для всех $d \in (\Sigma_{\text{bool}})^*$.

Отсюда мы заключаем, что для языка $L(M)$ алгоритм A является $O(n^k)$-верификатором.[19]

$\subseteq$ Во-вторых, докажем включение VP $\subseteq$ NP.

Пусть $L \subseteq \Sigma^*$ (для некоторого алфавита Σ) – язык класса VP. Тогда существует полиномиально-временной верификатор A, такой что $V(A) = L$. Полиномиально-временна́я НМТ M, моделирующая A, может быть задана следующим образом.

Вход: Некоторое слово $x \in \Sigma^*$.

Этап 1. M недетерминированно генерирует слово $c \in (\Sigma_{\text{bool}})^*$.

Этап 2. M пошагово моделирует работу машины A над входом (x, c).

Этап 3. M принимает слово x, если A принимает пару (x, c) – и отклоняет x, если A отклоняет (x, c).

Очевидно, что $L(M) = V(A)$ и

$$\text{Time}_M(x) \leq 2 \cdot \text{Time}_A(x, c)$$

для каждого $x \in L(M)$ и самого короткого доказательства c для этого слова x. Поэтому машина M работает за полиномиальное время – следовательно, $L \in$ NP. □

Согласно теореме 6.46, класс NP является множеством всех таких языков L, что для каждого слова $x \in L$ существует доказательство c_x утверждения $x \in L$, обладающее следующими свойствами.

- Длина c_x полиномиально зависит относительно длины x.
- За полиномиальное время можно проверить, является ли c_x доказательством утверждения $x \in L$.

6.6 NP-полнота

В отличие от теории вычислимости – которая нам даёт хорошо разработанную методологию для классификации проблем, деления их на алгоритмически разрешимые

[19] Т. е. $V(A) = L(M)$.

и неразрешимые – в теории сложности не удалось создать удачного математического аппарата для классификации алгоритмических проблем, принадлежащих классу P, относительно их разрешимости на практике. Достаточно мощные методы доказательства нижних границ сложности конкретных проблем не достигают желаемого результата. Следующий факт показывает, как далеко мы находимся от доказательства неразрешимости задач класса NP за полиномиальное время: наибольшей известной нижней границей временно́й сложности многоленточных машин Тьюринга, предназначенных для решения конкретных проблем класса NP, является тривиальная нижняя граница $\Omega(n)$ – совпадающая со временем, необходимым просто для чтения всего входа. То есть ни для какой проблемы класса NP мы пока не способны даже доказать нижнюю границу $\Omega(n \cdot \log n)$ – несмотря на то, что самые лучшие известные алгоритмы для тысяч задач класса NP выполняются только за экспоненциальное время. Следовательно, только наш опыт позволяет нам *верить* в то, что $\Omega(2^n)$ и является нижней границей временно́й сложности многих задач – но нами пока не удалось *доказать*, что для них существует нижняя граница, превышающая $\Omega(n)$.

Для того, чтобы хотя бы немного восполнить пробел между математической реальностью (неудовлетворённостью разработкой методов доказательств) – и верой в существование экспоненциальных нижних границ для упомянутых выше задач, некоторые учёные предложили создать метод классификации задач (определяющий их практическую разрешимость) – в котором можно допускать достоверность недоказанного, но правдоподобного предположения. Введение такого допущения привело к созданию концепции NP-сложности, которая позволила доказать, что некоторые проблемы являются неразрешимыми за полиномиальное время – с *дополнительным предположением*, что класс P является *собственным* подмножеством класса NP. Цель настоящего раздела – познакомить читателя с такой концепцией.

Обоснованность предположения $\mathrm{P} \subsetneq \mathrm{NP}$ начнём с обсуждения *теоретических* доводов; некоторые из них были приведены в разделе 6.5. Трудно поверить в то, что проверка доказательства является столь же трудной, как и его создание. К тому же, несмотря на значительные достижения в рассматриваемой нами области теоретической информатики, всё ещё не разработаны методы моделирования недетерминизма с помощью детерминизма, которые бы существенно отличались от последовательного перебора всех недетерминированных вычислений. Так как деревья вычислений недетерминированных алгоритмов могут содержать экспоненциально много различных вычислений (относительно своей глубины), а сама глубина фактически представляет собой временну́ю сложность недетерминированного алгоритма – то при применении недетерминированного моделирования кажется неизбежным экспоненциальное увеличение временно́й сложности.

Укажем также *практическую* причину обоснованности утверждения $\mathrm{P} \subsetneq \mathrm{NP}$ – она основана на нашем опыте разработки алгоритмов решения трудных задач. Мы знаем более 3000 задач класса NP, для которых неизвестно детерминированных полиномиально-временны́х алгоритмов; при этом многие из таких задач детально изучались в течение последних 40 лет. Вряд ли этот факт является следствием того, что мы неспособны найти для них эффективные алгоритмы. Даже если бы всё обстояло именно таким образом, классы P и NP *с точки зрения современной практики* были бы

различны[20] – по той же самой причине, т. е. поскольку мы не знаем полиномиально-временны́х алгоритмов решения многих задач класса NP.

Отметим, что предположение $\mathrm{P} \subsetneq \mathrm{NP}$ *не* аналогично предположению об обоснованности тезиса Чёрча–Тьюринга.[21] В то время как тезис Чёрча–Тьюринга – это аксиома, которая не может быть доказана, верность или неверность утверждения $\mathrm{P} \subsetneq \mathrm{NP}$ в будущем может быть доказана – и поэтому это предположение нельзя рассматривать в качестве аксиомы.

До конца этого раздела мы будем предполагать, что $\mathrm{P} \subsetneq \mathrm{NP}$. Как это предположение может помочь доказательству результатов типа $L \notin \mathrm{P}$ – для конкретных языков L? Идея заключается в том, чтобы точно определить подмножество *самых* трудных проблем – среди принадлежащих классу NP. Это подмножество долно быть определено таким образом, чтобы из принадлежности классу P любой такой проблемы прямо выводилось бы равенство $\mathrm{P} = \mathrm{NP}$. А поскольку мы уже предположили, что $\mathrm{P} \subsetneq \mathrm{NP}$, никакая из таких трудных проблем класса NP не может принадлежать классу P.

Так же, как и в теории сложности, мы будем использовать математическую концепцию сводимости – для точного определения подкласса задач, трудных относительно полиномиально-временно́й разрешимости. Некоторый язык (задача принадлежности) L является трудным, если распознавание любого языка класса NP (т. е. решение любой проблемы принадлежности из NP) может быть эффективно сведено к распознаванию языка L.

Определение 6.47. *Пусть $L_1 \subseteq \Sigma_1^*$ и $L_2 \subseteq \Sigma_2^*$ – некоторые языки. Будем говорить, что язык L_1* **сводим полиномиально по времени к языку L_2**, *и писать при этом $L_1 \leq_p L_2$, если существует полиномиально-временна́я* МТ *(алгоритм) A (рис. 6.3), которая вычисляет отображение, действующее из Σ_1^* в Σ_2^*, такое что для каждого $x \in \Sigma_1^*$ выполнено следующее:*

$$x \in L_1 \Leftrightarrow A(x) \in L_2.$$

A называется **полиномиально-временно́й сводимостью** *языка L_1 к L_2.*

Отметим, что сводимость $\leq_p$ может быть получена из сводимости $\leq_{\mathrm{m}}$ с помощью дополнительного требования эффективности (рис. 6.3). С другой стороны, условие

$$L_1 \leq_p L_2$$

означает, что язык L_2 является по крайней мере столь таким же трудным, как и L_1, – но трудность при этом рассматривается относительно полиномиально-временно́й разрешимости, вместо (просто) разрешимости.

Определение 6.48. *Язык L называется* **NP-трудным**, *если для каждого $L' \in \mathrm{NP}$ выполнено условие*

$$L' \leq_p L.$$

Язык L называется **NP-полным**, *если:*

[20] Если мы доказываем несуществование полиномиально-временно́го алгоритма решения некоторой конкретной проблемы, делая при этом предположение $\mathrm{P} \subsetneq \mathrm{NP}$, – то это действительно означает, что данная задача является трудной для современной практики.

[21] Несмотря на употребление синонимов – «тезис» и «предположение». *(Прим. перев.)*

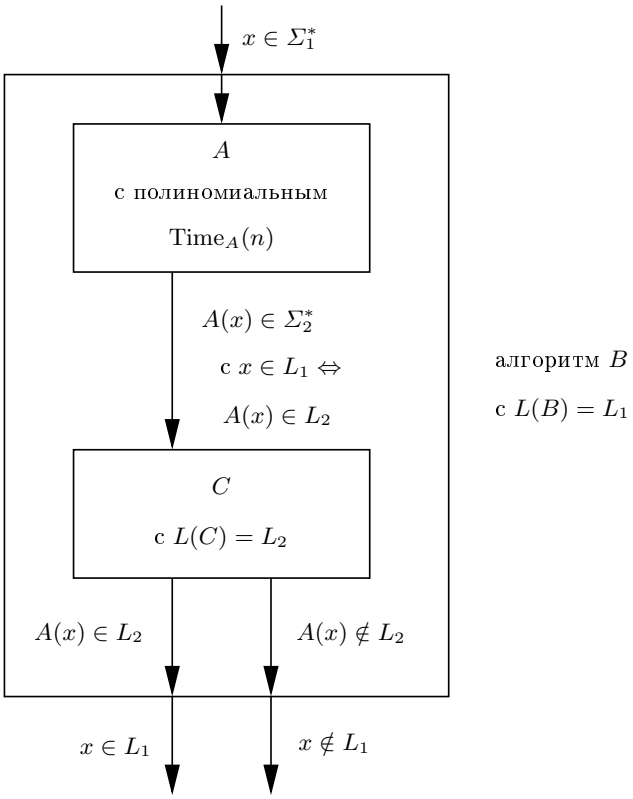


Рис. 6.3.

- $L \in \mathrm{NP}$;
- L – NP-*трудный.*

Далее множество NP-полных языков рассматривается в качестве именно того подкласса наиболее трудных проблемы класса NP, который мы и пытались неформально описать ранее. Следующая лемма показывает, что наша спецификация понятия трудности действительно соответствует такому свойству трудных проблем – а именно, что при сделанном предположении $\mathrm{P} \neq \mathrm{NP}$ пересечение класса P и множества трудных проблем является пустым (рис. 6.4).

Лемма 6.49. *Если* $L \in \mathrm{P}$ *и* L – NP-*трудный, то* $\mathrm{P} = \mathrm{NP}$.

Доказательство. Факт $L \in \mathrm{P}$ влечёт существование полиномиально-временно́й машины Тьюринга M с языком $L = L(M)$. Докажем, что для каждого языка $U \in \mathrm{NP}$ над некоторым алфавитом Σ существует полиномиально-временна́я ММТ (алгоритм) A_U, допускающая язык $L(A_U) = U$, т. е. что $U \in \mathrm{P}$; очевидно, что из последнего следует равенство $\mathrm{P} = \mathrm{NP}$.

Поскольку для каждого языка $U \in \mathrm{NP}$ выполнено условие $U \leq_p L$, существует полиномиально-временна́я МТ B_U, такая что

$$x \in U \Leftrightarrow B_U(x) \in L.$$

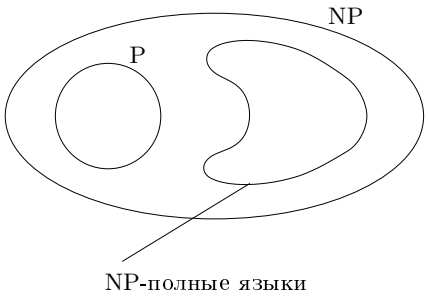


Рис. 6.4.

Опишем работу полиномиально-временно́й ММТ A_U, допускающей язык $L(A_U) = U$. Для любого входа $x \in \Sigma^*$ машина A_U работает следующим образом.

1. A_U моделирует вычисление машины B_U над словом x и вычисляет $B_U(x)$.
2. A_U моделирует работу машины M над выходом $B_U(x)$ – и допускает x тогда и только тогда, когда M допускает $B_U(x)$.

Так как условия $x \in U$ и $B_U(x) \in L$ равносильны, выполнено равенство $L(A_U) = U$. И, поскольку выполнены следующие три условия:

- $\mathrm{Time}_{A_U}(x) = \mathrm{Time}_{B_U}(x) + \mathrm{Time}_M(B_U(x))$;
- функция $|B_U(x)|$ является полиномиальной относительно длины $|x|$;
- обе машины – B_U и M – работают за полиномиальное время

– мы получаем, что A_U также работает за полиномиальное время. □

Теперь мы получим искомую спецификацию наиболее трудных проблемы класса NP – доказав, что множество NP-полных языков не пусто. Теорема 6.52 покажет, что соответствующие «опасения» напрасны: проблема SAT является NP-полной[22], и поэтому концепция NP-полноты действительно обоснована.

NP-трудность проблемы SAT говорит о том, что язык Булевых формул является весьма мощным средством – достаточном для представления любой проблемы принадлежности класса NP. Более точно это означает, что для любого языка $L \in \mathrm{NP}$ вопрос

«принадлежит ли слово x языку L?»

можно переформулировать как

«выполнима ли заданная Булева формула $\Phi_{x,L}$?»

Описательные возможности языка Булевых формул не должны вызывать удивления: формулы могут быть использованы для описания любого текста, а тексты, в свою очередь, могут применяться для формализации произвольных объектов – таких, как теоремы, доказательства, вычисления, графы и т.д.

Приведём простой пример, подтверждающий это интуитивное соображение. Пусть нам нужно описать метод создания таких Булевых формул в КНФ, которые могли бы

[22] Напомним, что мы уже доказали, что $\mathrm{SAT} \in \mathrm{NP}$, поэтому остаётся доказать, что проблема SAT – NP-трудная.

быть использованы для описания матриц размера 3×3, содержащих значения -1, 0 и 1. При этом формула должна быть выполнимой тогда и только тогда, когда в каждой строке и в каждом столбце соответствующей матрицы имеется ровно один элемент 1.

Для этого рассмотрим 27 переменных $x_{i,j,k}$ – для каждого варианта значений индексов $i, j \in \{1, 2, 3\}$ и каждого $k \in \{-1, 0, 1\}$. Следующий смысл присваивания значений этим переменным[23] позволяет нам описать любую матрицу $A = (a_{ij})_{i,j=1,2,3}$ размера 3×3:

$$x_{i,j,1} = 1 \Leftrightarrow a_{ij} = 1,$$
$$x_{i,j,0} = 1 \Leftrightarrow a_{ij} = 0,$$
$$x_{i,j,-1} = 1 \Leftrightarrow a_{ij} = -1.$$

Используя такую интерпретацию, можно однозначно определить матрицу

$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

с помощью следующей подстановки значений рассматриваемым переменным:

$$x_{1,1,1} = 1,\ x_{1,1,0} = 0,\ x_{1,1,-1} = 0,$$
$$x_{1,2,1} = 0,\ x_{1,2,0} = 1,\ x_{1,2,-1} = 0,$$
$$x_{1,3,1} = 0,\ x_{1,3,0} = 1,\ x_{1,3,-1} = 0,$$
$$x_{2,1,1} = 0,\ x_{2,1,0} = 0,\ x_{2,1,-1} = 1,$$
$$x_{2,2,1} = 0,\ x_{2,2,0} = 1,\ x_{2,2,-1} = 0,$$
$$x_{2,3,1} = 1,\ x_{2,3,0} = 0,\ x_{2,3,-1} = 0,$$
$$x_{3,1,1} = 0,\ x_{3,1,0} = 1,\ x_{3,1,-1} = 0,$$
$$x_{3,2,1} = 1,\ x_{3,2,0} = 0,\ x_{3,2,-1} = 0,$$
$$x_{3,3,1} = 0,\ x_{3,3,0} = 1,\ x_{3,3,-1} = 0.$$

Однако отметим, что существует такие варианты подстановок, которые *не* определяют матрицы – т. е. являются бессмыслицей[24]. Например, две подстановки

$$x_{1,1,0} = x_{1,1,1} = 1$$

означали бы, что элемент a_{11} должен принимать одновременно два значения, 0 и 1, – что невозможно. Для исключения подобных возможностей опишем построение Булевой формулы, принимающей значение 1 только в результате таких подстановок значений её переменным, которые действительно соответствуют требуемому представлению матрицы 3×3 над множеством $\{-1, 0, 1\}$.

[23] При использовании алфавита Σ_{bool} и переменных над этим алфавитом мы называли подобные действия подстановкой. Ниже будем иногда употреблять это название и для других алфавитов, в том числе – для рассматриваемого нами $\{-1, 0, 1\}$. *(Прим. перев.)*

[24] Это слово употреблялось и ранее – в разделе 6.4. В этом разделе (здесь и ниже) смысл слова «бессмыслица», казалось бы, иной – однако с точки зрения практического программирования (в том числе – для машин Тьюринга) он практически одинаков: генерируется какой-либо объект (навигатор, матрица и др.) – котоый впоследствии *не* обрабатывается. *(Прим. перев.)*

Начнём с построения вспомогательных формул. Для каждой пары $i, j \in \{1, 2, 3\}$ формула

$$F_{i,j} = (x_{i,j,1} \vee x_{i,j,0} \vee x_{i,j,-1}) \wedge \\ (\bar{x}_{i,j,1} \vee \bar{x}_{i,j,0}) \wedge (\bar{x}_{i,j,1} \vee \bar{x}_{i,j,-1}) \wedge (\bar{x}_{i,j,0} \vee \bar{x}_{i,j,-1})$$

гарантирует, что в точности одна из переменных

$$x_{i,j,1},\ x_{i,j,0},\ x_{i,j,-1}$$

принимает значение 1 – поэтому для рассматриваемой пары i, j значение позиции (i, j) матрицы определено однозначно.[25]

При применении этого подхода каждая подстановка, удовлетворяющая формуле

$$\Phi = \bigwedge_{1 \le i,j \le 3} F_{i,j},$$

однозначно определяет матрицу 3×3, все элементы которой принадлежат множеству $\{-1, 0, 1\}$. Далее, для $i = 1, 2, 3$ формула

$$Z_i = (x_{i,1,1} \vee x_{i,2,1} \vee x_{i,3,1}) \wedge \\ (\bar{x}_{i,1,1} \vee \bar{x}_{i,2,1}) \wedge (\bar{x}_{i,1,1} \vee \bar{x}_{i,3,1}) \wedge (\bar{x}_{i,2,1} \vee \bar{x}_{i,3,1})$$

гарантирует, что i-я строка содержит в точности одно значение 1. Аналогично, формула

$$S_j = (x_{1,j,1} \vee x_{2,j,1} \vee x_{3,j,1}) \wedge \\ (\bar{x}_{1,j,1} \vee \bar{x}_{2,j,1}) \wedge (\bar{x}_{1,j,1} \vee \bar{x}_{3,j,1}) \wedge (\bar{x}_{2,j,1} \vee \bar{x}_{3,j,1})$$

гарантирует, что в точности одно значение 1 содержит j-й столбец – также для $j = 1, 2, 3$.

Следовательно,

$$\Phi \wedge \bigwedge_{i=1,2,3} Z_i \wedge \bigwedge_{j=1,2,3} S_j$$

– искомая Булева формула в КНФ.

Упражнение 6.50. Задайте множество Булевых переменных, достаточное для описания любой позиции (ситуации) в шахматах. Опишите процесс создания такой формулы над этими переменными, которая принимает значение 1 тогда и только тогда, когда на доске находятся ровно 8 ферзей, кроме них других фигур нет – и при этом ни один из ферзей не атакует другого.

Используя подобную стратегию, мы можем описать представление текста, приведённого на заданной странице. Страницу можно разделить на $n \times m$ элементов, содержащих символы алфавита Σ_{keyboard}; эти элементы формируют матрицу (двумерный массив). При этом для представления любого текста, приведённого на заданной странице, достаточно $n \cdot m \cdot |\Sigma_{\text{keyboard}}|$ Булевых переменных.

[25] Немного подробнее. Элементарная дизъюнкция $x_{i,j,1} \vee x_{i,j,0} \vee x_{i,j,-1}$ гарантирует, что по крайней мере одна из будущих переменных – $x_{i,j,1}$, $x_{i,j,0}$ и $x_{i,j,0}$ – примет значение 1. А элементарная дизъюнкция $\bar{x}_{i,j,1} \vee \bar{x}_{i,j,0}$ гарантирует, что по крайней мере одна из входящих в неё переменных примет значение 0. И так далее.

Упражнение 6.51. Задайте множество Булевых переменных, достаточное для описания любого текста на странице размера 33×77. Задайте формулу, принимающую значение 1 в случае только одного варианта подстановки значений этим переменным – такого варианта, при котором эта подстановка соответствует тексту второй страницы учебника «Разработка и анализ компьютерных алгоритмов» Ахо, Хопкрофта и Ульмана.

Итак, мы можем описать (представить) произвольный текст с помощью формулы. Поскольку любая конфигурация машины Тьюринга тоже является текстом, т. е. словом над некоторым алфавитом, мы можем использовать подобную формулу для описания конфигураций. Суть доказательства следующей теоремы заключается в том, что можно использовать выполнимость Булевой формулы даже для описания семантических отношений различных частей текста – например, такого: «первая конфигурация достижима из второй за один шаг вычислений». Способность выразить один шаг вычислений в виде отношения позволяет полностью описать любое последовательное вычисление. Другой важный момент доказательства теоремы заключается в том, что подобные формулы могут быть созданы с помощью некоторого алгоритма – и при этом длина каждой из них относительно длины рассматриваемого вычисления является полиномом.[26]

Теорема 6.52 (Кука).* *Проблема* SAT – NP-*полная.*

Доказательство. Мы уже использовали пример 6.1 (см. также рис. 6.2) для доказательства того, что проблема SAT входит в класс VP – который совпадает с классом NP. Поэтому для завершения доказательства остаётся показать, что все языки класса NP полиномиально сводимы по времени к языку SAT.

Согласно определению класса NP, для каждого языка $L \in \mathrm{NP}$ существует некоторая НМТ M, такая что для некоторого $c \in N$ выполнены условия

$$L(M) = L \quad \text{и} \quad \mathrm{Time}_M(n) \in O(n^c).$$

Будем рассматривать M в качестве конечного представления языка L – что может быть названо «входной частью» разрабатываемой нами полиномиально-временно́й сводимости. Таким образом, мы должны показать, что для любой полиномиально-временно́й НМТ M выполнено условие $L(M) \leq_p \mathrm{SAT}$.

Пусть $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{accept}}, q_{\mathrm{reject}})$ – произвольная НМТ, такая что для некоторого полинома p выполнено неравенство

$$\mathrm{Time}_M(n) \leq p(n).$$

Пусть $Q = \{q_0, q_1, \ldots, q_{s-1}, q_s\}$ и $\Gamma = \{X_1, \ldots, X_m\}$, причём $q_{s-1} = q_{\mathrm{reject}}$, $q_s = q_{\mathrm{accept}}$, $X_m = \sqcup$. Разработаем полиномиально-временно́е сведе́ние $B_M : \Sigma^* \to (\Sigma_{\mathrm{logic}})^*$, такое что для всех $x \in \Sigma^*$ выполнено условие

$$x \in L(M) \iff B_M(x) \in \mathrm{SAT}.$$

Пусть w – произвольное слово языка Σ^*. Машина B_M должна создавать формулу $B_M(w)$, такую что

[26] Отметим, что если вычисление является полиномом относительно длины входных данных, то длина подобной формулы также полиномиально зависит от этой длины.

$$w \in L(M) \iff \text{формула } B_M(w) \text{ является выполнимой.}$$

Разработаем формулу $B_M(w)$ таким образом, чтобы различные варианты подстановки для её переменных можно было использовать для описания всех возможностей работы машины M над словом w – т. е. всех возможных вычислений. Основная идея этого построения заключается в выборе таких значений входящих в эту формулу Булевых переменных, чтобы мы могли описать любую конфигурацию машины M, в любой момент времени её работы – т. е. после любого числа шагов вычисления, начинающегося с инициальной конфигурации для слова w.

Напомним, что

$$\mathrm{Space}_M(|w|) \leq \mathrm{Time}_M(|w|) \leq p(|w|)$$

– поэтому любая конфигурация[27] может быть описана некоторым словом, длина которого не превышает $p(|w|)+1$; заметим, что для зафиксированного входа w значение $p(|w|)$ тоже является фиксированным натуральным числом.

Для упрощения описания конфигураций мы представим любую из них, т. е.

$$(\text¢ Y_1 Y_2 \ldots Y_{i-1} q Y_i \ldots Y_d)$$

для некоторого $d \leq p(|w|)$, как

$$(\text¢ Y_1 Y_2 \ldots Y_{i-1} q Y_i \ldots Y_d Y_{d+1} \ldots Y_{p(|w|)})$$

– где $Y_{d+1} = Y_{d+2} = \ldots = Y_{p(|w|)} = \sqcup$. Таким образом, представление каждой конфигурации имеет одну и ту же длину, $p(|w|)+1$. А для упрощения поиска некоторой допускающей конфигурации расширим определение функции δ путём добавления следующего перехода для любого $X \in \Gamma$:

$$\delta(q_{\text{accept}}, X) = (q_{\text{accept}}, X, N);$$

отметим, что это расширение не изменяет язык $L(M)$. Итак, когда машина M достигает состояния q_{accept}, она остаётся в нём всё время – не изменяя какую-либо часть своей текущей конфигурации. Поэтому для проверки принадлежности слова w языку $L(M)$ достаточно искать состояние q_{accept} только среди конфигураций, достижимых из инициальной ровно за $p(|w|)$ шагов машины M.

Создадим формулу $B_M(w)$ из переменных следующих типов.

- $\boldsymbol{C\langle i, j, t\rangle}$ для $0 \leq i \leq p(|w|)$, $1 \leq j \leq m$, $0 \leq t \leq p(|w|)$.
  Смысл переменных $C\langle i, j, t\rangle$ следующий:

  $C\langle i, j, t\rangle = 1 \Leftrightarrow$ i-я ячейка ленты машины M
  содержит символ $X_j \in \Gamma$ в момент времени t
  (после t шагов вычисления из стартовой
  конфигурации машины M над словом w).

  Количество таких переменных –

$$m \cdot ((p(|w|)+1)^2 \in O((p(|w|))^2).$$

[27] Представление конфигурации включает содержимое ячеек ленты с номерами $0, 1, \ldots, p(|w|)$, а также текущее состояние.

- $\boldsymbol{S\langle k,t\rangle}$ для $0 \le k \le s$, $0 \le t \le p(|w|)$.
  Смысл переменных $S\langle k,t\rangle$ следующий:
  $S\langle k,t\rangle = 1 \Leftrightarrow$ НМТ M в момент времени t находится в состоянии q_k.
  Количество таких переменных –

$$(s+1)\cdot(p(|w|)+1) \in O(p(|w|)).$$

- $\boldsymbol{H\langle i,t\rangle}$ для $0 \le i \le p(|w|)$, $0 \le t \le p(|w|)$.
  Смысл переменных $H\langle i,t\rangle$ следующий:

  $H\langle i,t\rangle = 1 \Leftrightarrow$ головка машины M в момент времени t
  находится i-й позиции ленты.

  Количество таких переменных –

$$(p(|w|)+1)^2 \in O((p(|w|))^2).$$

Отметим, что мы можем описать произвольную конфигурацию путём выбора значений всех переменных, соответствующих ранее зафиксированному параметру t. Например, конфигурация

$$(X_{j_0}X_{j_1}\ldots X_{j_{i-1}}q_rX_{j_i}\ldots X_{j_{p(|w|)}})$$

может быть описана следующим образом:

- $C\langle 0,j_0,t\rangle = C\langle 1,j_1,t\rangle = \ldots = C\langle p(|w|),j_{p(|w|)},t\rangle = 1$, и $C\langle k,l,t\rangle = 0$ для всех оставшихся переменных этого типа,
- $H\langle i,t\rangle = 1$ и $H\langle j,t\rangle = 0$ для всех $j \in \{0,1,\ldots,p(|w|)\}, j \ne i$,
- $S\langle r,t\rangle = 1$ и $S\langle l,t\rangle = 0$ для всех $l \in \{0,1,\ldots,s\}, l \ne r$.

Итак, зафиксировав слово w, а также соответствующую конфигурацию для каждого возможного значения времени t, мы можем описать все возможные вычисления машины M длины $p(|w|)$ над этим словом путём присваивания подходящих значений рассматриваемым Булевым переменным. Важно отметить, что среди них существуют такие присваивания, у которых нет никакой интерпретации для вычислений машины над словом w, – т. е. они являются бессмыслицей. Например, присваивание

$$S\langle 1,3\rangle = S\langle 3,3\rangle = S\langle 7,3\rangle = 1 \quad \text{и} \quad C\langle 2,1,3\rangle = C\langle 2,2,3\rangle = 1$$

означало бы, что после 3 шагов вычисления машина может находиться в каждом из состояний q_1, q_3 и q_7 – и, кроме того, вторая позиция ленты содержит как символ X_1, так и символ X_2. Поэтому очевидно, что такое присваивание значений переменным, соответствующим $t = 3$, не может определять никакую конфигурацию.

Чтобы прояснить эту ситуацию, рассмотрим лист размера

$$(p(|w|)+2) \times (p(|w|)+1).$$

Присваивание значений переменным может использоваться для того, чтобы в каждой позиции (i,j), где $0 \le i,j \le p(|w|)$, зафиксировать некоторой символ. Результат такого присваивания может быть бессмыслицей – однако существуют и такие присваивания, в которых каждая строка листа соответствует некоторой конфигурации, и, более того, все строки в рассматриваемом порядке соответствуют некоторому вычислению. Такое

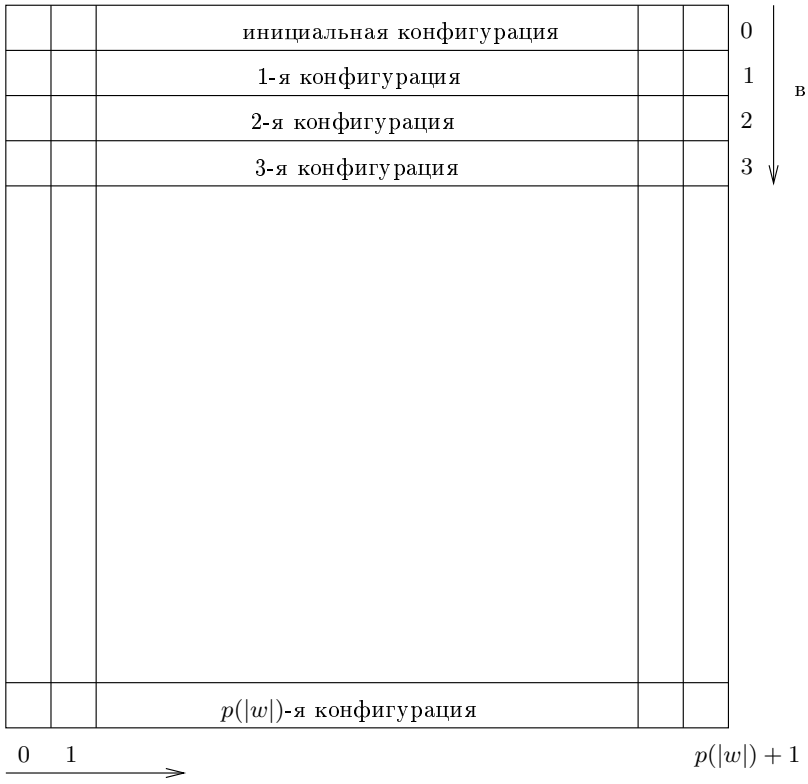


Рис. 6.5.

присваивание и определяет необходимый нам подход к представлению вычислений – и в общих чертах этот подход приведён на рис. 6.5.

Теперь наша задача состоит в создании формулы $B_M(w)$ над переменными $C\langle i,j,t\rangle$, $S\langle k,t\rangle$ и $H\langle i,t\rangle$ в КНФ – являющейся выполнимой тогда и только тогда, когда существует некоторое допускающее вычисление машины M над словом w. Алгоритм B_M создаёт формулу

$$B_M(x) = A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G$$

за следующие семь этапов – ниже мы по-отдельности опишем формулы A, B, C, D, E, F и G.

- A гарантирует, что в любой момент времени t головка установлена в точности на одну ячейку (позицию) ленты. Иными словами, формула A будет принимать значение 1, если значение 1 присвоено ровно одной из переменных множества $H\langle i,t\rangle$ – для любого зафиксированного значения времени t.
- B имеет значение 1, если в любой момент времени t любая позиция ленты содержит в точности один символ алфавита Γ.
- C имеет значение 1, если в любой момент времени t машина M находится в точности в одном состоянии.

D обеспечивает, что на любом шаге вычисления может быть изменён только тот символ, на который установлена головка.

E обеспечивает, что на любом шаге вычисления изменение состояния, движение головки и замена символа на ленте согласуются с возможной работой машины M в данной конфигурации – т. е. с функцией переходов δ.

F обеспечивает, что переменные, соответствующие значению $t = 0$, определяют начальную конфигурацию машины M над словом w.

G обеспечивает, что последняя (т. е. $((p(|w|) + 1)$-я) конфигурация является допускающей.

Таким образом, выполнимость части $A \wedge B \wedge C$ формулы $B_M(x)$ обеспечивает тот факт, что каждая строка листа, приведённого на рис. 6.5, содержит некоторую конфигурацию. Часть $D \wedge E$ дополнительно обеспечивает, что текст на листе соответствует некоторому вычислению машины M. F гарантирует, что это вычисление является вычислением над словом w, а G – что это вычисление достигает состояния q_{accept}.

В дальнейших построениях нам часто будет нужна вспомогательная формула – которая принимает значение 1 если и только если в точности одна из её переменных равна 1. Пусть $x_1, x_2, \ldots, x_n$ – Булевы переменные; тогда искомой является формула

$$U(x_1, x_2, \ldots, x_n) = (x_1 \vee x_2 \vee \ldots \vee x_n) \wedge (\bigwedge_{\substack{1 \le i,j \le n \\ i \ne j}} (\overline{x_i} \vee \overline{x_j})).$$

Первая часть формулы $U(x_1, x_2, \ldots, x_n)$ гарантирует, что значение 1 должно быть присвоено по крайней мере одной из переменных $x_1, x_2, \ldots x_n$. А поскольку для всех пар значений $i, j \in \{1, \ldots, n\}$, таких что $i \ne j$, вторая часть этой формулы содержит элементарные клаузы $\overline{x_i} \vee \overline{x_j}$, мы не можем присвоить значение 1 обеим этим переменным. Следовательно, любая подстановка, удовлетворяющая второй части формулы, содержит не более одного значения 1. Заметим, что длина формулы $U(x_1, x_2, \ldots, x_n)$ относительно числа переменных n квадратична.

Теперь опишем само создание формул A, B, C, D, E, F и G. Всюду при их описании считаем, что $i, t \in \{0, 1, 2, \ldots, p(|w|)\}$ – если не сказано иного.

A: Для каждого t положим

$$A_t = U(H\langle 0, t\rangle, H\langle 1, t\rangle, \ldots, H\langle p(|w|), t\rangle).$$

Формула A_t выполняется в том и только том случае, если в *конкретный* момент времени t (т. е. после t шагов вычисления) головка установлена в точности на одну позицию ленты.

Положим

$$A = A_0 \wedge A_1 \wedge \ldots \wedge A_{p(|w|)} = \bigwedge_{0 \le i \le p(|w|)} A_i$$

Из предыдущего следует, что формула A выполняется в том и только том случае, если в *каждый* момент времени головка установлена в точности на одну позицию ленты. Число символов в формуле A есть $O((p(|w|))^3)$ – поскольку число символов в каждой формуле A_t квадратично зависит от $p(|w|) + 1$.

B: Для каждой пары i, t положим

$$B_{i,t} = U(C\langle i, 1, t\rangle, C\langle i, 2, t\rangle, \ldots, C\langle i, m, t\rangle).$$

Формула $B_{i,t}$ принимает значение 1 тогда и только тогда, когда в *конкретный* момент времени t *конкретная*, i-я ячейка ленты содержит в точности один символ. Поскольку $|\Gamma| = m$ – некоторая константа, число символов формулы $B_{i,t}$ есть $O(1)$.
Выполнение формулы

$$B = \bigwedge_{0 \le i,t \le p(|w|)} B_{i,t}$$

гарантирует, что в *каждый* момент времени *каждая* ячейка ленты содержит в точности один символ. Очевидно, что число символов формулы B есть $O((p(|w|))^2)$.

C: Для всех t определим

$$C_t = U(S\langle 0,t\rangle, S\langle 1,t\rangle, \ldots, S\langle s,t\rangle).$$

Если некоторая подстановка переменных $S\langle 0,t\rangle, \ldots, S\langle s,t\rangle$ удовлетворяет формуле C_t, то машина M в *конкретный* момент времени t находится в точности в каком-то одном состоянии. Поскольку $|Q| = s+1$ является константой, число символов формулы C_t есть $O(1)$.
Формула C такова:

$$C = \bigwedge_{0 \le t \le p(|w|)} C_t\,.$$

Выполнение условия C гарантирует то, что M в *каждый* момент времени находится ровно в одном состоянии. Число символов формулы C есть $O(p(|w|))$.

D: Формула

$$D_{i,j,t} = (C\langle i,j,t\rangle \leftrightarrow C\langle i,j,t+1\rangle) \vee H\langle i,t\rangle$$

для $1 \le j \le m$ и $0 \le t \le p(|w|)-1$ утверждает, что на $(t+1)$-м шаге вычисления может быть заменён только один символ ленты – тот символ, на который установлена головка; точнее, если $H\langle i,t\rangle = 0$, то символ на i-й ячейке ленты *не* может быть заменён на данном шаге вычисления. Очевидно, что $D_{i,j,t}$ может быть преобразована в КНФ[28] – причём таким путём, что её длина остаётся в пределах $O(1)$.
Искомая формула D такова:

$$D = \bigwedge_{\substack{0 \le i \le p(|w|) \\ 1 \le j \le m \\ 0 \le t \le p(|w|)-1}} D_{i,j,t}\,.$$

При этом D содержит $O((p(|w|))^2)$ символов.[29]

E: Для всех четвёрок, состоящих из i, t, а также $j \in \{1,\ldots,m\}$ и $k \in \{0,1,\ldots,s\}$, рассмотрим формулу

$$E_{i,j,k,t} = \overline{C\langle i,j,t\rangle} \vee \overline{H\langle i,t\rangle} \vee \overline{S\langle k,t\rangle} \vee \bigvee_l (C\langle i,j_l,t+1\rangle \wedge S\langle k_l,t+1\rangle \wedge H\langle i_l,t+1\rangle),$$

[28] Формула $x \leftrightarrow y$ эквивалентна формуле $(\overline{x} \vee y) \wedge (x \vee \overline{y})$. Следовательно,

$$D_{i,j,t} \leftrightarrow (\overline{C\langle i,j,t\rangle} \vee C\langle i,j,t+1\rangle \vee H\langle i,t\rangle) \wedge (C\langle i,j,t\rangle \vee \overline{C\langle i,j,t+1\rangle} \vee H\langle i,t\rangle).$$

[29] Поскольку m является константой. Она была введена выше, в пункте (b). *(Прим. перев.)*

где l пробегает все возможные переходы машины M для набора аргументов (q_k, X_j), причём

$$(q_{k_l}, X_{j_l}, z_l) \in \delta(q_k, X_j),\ z_l \in \{L, R, N\},$$
$$i_l = i + \varphi(z_l),\ \varphi(L) = -1,\ \varphi(R) = 1,\ \varphi(N) = 0.$$

Формула $E_{i,j,k,t}$ представляется в виде дизъюнкции четырёх условий; поясним их.

- $\overline{C\langle i,j,t\rangle}$ означает, что i-я позиция ленты в момент времени t *не* содержит X_j.
- $\overline{H\langle i,t\rangle}$ означает, что головка в момент времени t находится *не* в i-й позиции ленты.
- $\overline{S\langle k,t\rangle}$ означает, что M в момент времени t находится *не* в состоянии q_k.
- А последнее условие соответствует изменению t-й конфигурации – т. е. возможному переходу для набора аргументов (q_k, X_j) и головки, установленной в i-й позиции.

Теперь идея создания формулы $E_{i,j,k,t}$ становится очевидной. Если не выполнено ни одно из первых трёх ограничений, то (q_k, X_j) является парой текущих аргументов для $(t{+}1)$-го шага, а головка установлена на i-ю позицию ленты. В этом случае изменения производятся согласно функции переходов δ для пары аргументов (q_k, X_j). Если мы выбираем l-й возможный переход (q_{k_l}, X_{j_l}, z_l) для пары аргументов (q_k, X_j), то в i-й ячейке ленты символ X_j заменяется на X_{j_l}. При этом машина M переходит в состояние q_{k_l}, а её головка передвигается в соответствии со значением z_l.

Поскольку l является константой, формула $E_{i,j,k,t}$ содержит $O(1)$ символов – даже будучи преобразованной в КНФ. Следовательно, искомой является формула

$$E = \bigwedge_{\substack{0\le i,t\le p(|w|)\\ 1\le j\le m\\ 0\le k\le s}} E_{i,j,k,t}\,,$$

причём она состоит из $O((p(|w|))^2)$ символов.

F: Напомним, что инициальная конфигурация машины M над словом w содержит на ленте слово ¢w, головка установлена на позицию 0, при этом машина находится в состояние q_0. Пусть $w = X_{j_1}X_{j_2}\ldots X_{j_n}$ для некоторых $j_r \in \{1,2,\ldots,m\}$, $n \in \mathbb{N}_0$, и пусть $X_1 =$ ¢. Тогда тот факт, что машина M начинает вычисления в момент времени $t = 0$ в инициальной конфигурации, может быть выражен с помощью следующей формулы:

$$F = S\langle 0,0\rangle \wedge H\langle 0,0\rangle \wedge C\langle 0,1,0\rangle \wedge \bigwedge_{1\le r\le n} C\langle r, j_r, 0\rangle \wedge \bigwedge_{n+1\le d\le p(|w|)} C\langle d, m, 0\rangle\,.$$

Число символов формулы F находится в пределах $O(p(|w|))$, и при этом F записана в КНФ.

G: Формула G проста:

$$G = S\langle s, p(|w|)\rangle\,;$$

она обеспечивает то, что последняя (в нашем случае – $p(|w|)$-я) конфигурация содержит состояние q_{accept}.

Согласно описанному процессу построения формулы $B_M(w)$, мы получаем, что:

- каждая подстановка, удовлетворяющая $B_M(w)$, соответствует некоторому допускающему вычислению машины M над словом w – поэтому формула $B_M(w)$ выполнима тогда и только тогда, когда существует допускающее вычисление машины M над w;
- формула $B_M(w)$ может быть алгоритмически получена на основе заданных M, w и $p(|w|)$ за время, линейное относительно длины этой формулы.

Число символов формулы $B_M(w)$ есть $O((p(|w|))^3)$. Для записи этой формулы над алфавитом Σ_{logic} каждая переменная должна быть представлена в двоичном виде. Поскольку число переменных есть $O((p(|w|))^2)$, каждая переменная может быть закодирована с помощью $O(\log_2(|w|))$ битов. Поэтому длина формулы $B_M(w)$ – и, следовательно, временна́я сложность сводимости B_M – находятся в пределах

$$O((p(|w|))^3 \cdot \log_2(|w|)).$$

Итак B_M является полиномиально-временно́й сводимостью языка $L(M)$ к SAT. □

Доказательство теоремы 6.52 показывает, что все языки (проблемы принадлежности) класса NP полиномиально сводимы по времени к SAT. Главное следствие этого заключается в том, что каждый частный случай любой проблемы принадлежности класса NP может быть представлен как проблема выполнимости Булевой формулы. Одна из возможных интерпретаций этого факта заключается в том, что язык Булевых формул является достаточным для описания любой проблемы класса NP.

NP-полнота проблемы SAT является отправной точкой для классификации проблем принадлежности – с точки зрения класса P.[30] А для доказательства NP-полноты других языков мы используем метод сводимости, который основан на следующем утверждении.

Лемма 6.53. *Пусть L_1 и L_2 – некоторые языки. Если $L_1 \leq_p L_2$ и L_1 – NP-трудный, то L_2 – также NP-трудный.*

Упражнение 6.54. Докажите лемму 6.53.

Будем использовать лемму 6.53 для доказательства NP-полноты некоторых других языков класса NP. При этом главной целью будет доказательство того, что язык графов (язык отношений) достаточен для описания некоторой проблемы класса NP. Для более понятного описания вариантов полиномиально-временно́й сводимости будем пользоваться терминами «граф» и «формула» – вместо «представление графа» и «представление формулы». Ниже будем рассматривать такие проблемы:

$$\begin{aligned}
\text{SAT} &= \{\Phi \mid \Phi - \text{выполнимая формула в КНФ}\};\\
3\,\text{SAT} &= \{\Phi \mid \Phi - \text{выполнимая формула в } 3\,\text{КНФ}\};\\
\text{CLIQUE} &= \{(G,k) \mid G - \text{граф, который содержит } k\text{-клику}\};\\
\text{VC} &= \{(G,k) \mid G - \text{граф, у которого имеется}\\
&\qquad\qquad \text{вершинное покрытие размера } k\},
\end{aligned}$$

где:

[30] Приведём такую аналогию. SAT играет в теории сложности ту же самую роль, что и язык L_{diag} в теории вычислимости.

- формула представлена в 3КНФ, если она записана в КНФ, и при этом каждая клауза состоит не более чем из трёх символов;
- вершинное покрытие графа $G = (V, E)$ – любое подмножество $U \subseteq V$, такое что у каждой дуги[31] множества E имеется по крайней мере один элемент, принадлежащий подмножеству U.

Пусть Φ и φ – некоторые формула и подстановка переменных для этой формулы. Ниже мы будем обозначать истинность значения формулы Φ для сделанной подстановки φ записью $\varphi(\Phi)$.

Лемма 6.55.

$$\text{SAT} \leq_p \text{CLIQUE}.$$

Доказательство. Пусть

$$\Phi = F_1 \wedge F_2 \wedge \ \ldots \ \wedge F_m$$

– некоторая формула в КНФ, где

$$F_i = (l_{i1} \vee l_{i2} \vee \cdots \vee l_{ik_i}), \quad k_i \in \mathbb{N} \ \text{ для } i = 1, 2, \ldots, m\,.$$

Создадим частный случай проблемы клики (G, k), такой что

$$\Phi \in \text{SAT} \ \Leftrightarrow \ (G, k) \in \text{CLIQUE}.$$

Положим

$k = m$;
$G = (V, E)$, где:
$V = \{[i, j] \mid 1 \leq i \leq m, 1 \leq j \leq k_i\}$ – т. е. мы выбираем (новую) вершину для каждого вхождения литерала в формулу Φ,
$E = \{\{[i, j], [r, s]\} \mid$ для всех $[i, j], [r, s] \in V$, где $i \neq r$ и $l_{ij} \neq \overline{l}_{rs}\}$ – т. е. дуга $\{u, v\}$ содержит вершины, соответствующие символам различных клауз, и при этом u не является отрицанием v.

В качестве примера рассмотрим формулу

$$\Phi = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_3) \wedge \overline{x}_2.$$

Здесь $k = 4$, и соответствующий граф G приведён на рис. 6.6.

Очевидно, что проблема (G, k) может быть создана из Φ за полиномиальное время. Теперь покажем, что

$$\Phi \textit{ выполнима} \iff G \textit{ содержит клику размера } k = m. \tag{6.1}$$

Идея доказательства этого факта состоит в следующем. Вершины, соответствующие в графе G символам l_{ij} и l_{rs}, являются смежными тогда и только тогда, когда они относятся к различным клаузам (т. е. $i \neq r$), и при этом оба могут независимо принимать значение 1.[32] Следовательно, некоторая клика графа G соответствует таким подстановкам переменных формулы Φ, при которых все символы вершин клики получают значение 1. Например, клика

[31] Напомним, что дуга рассматриваемая как неупорядоченная пара вершин.
[32] Т. е. существует некоторая подстановка φ, такая что $\varphi(l_{ij}) = \varphi(l_{rs}) = 1$.

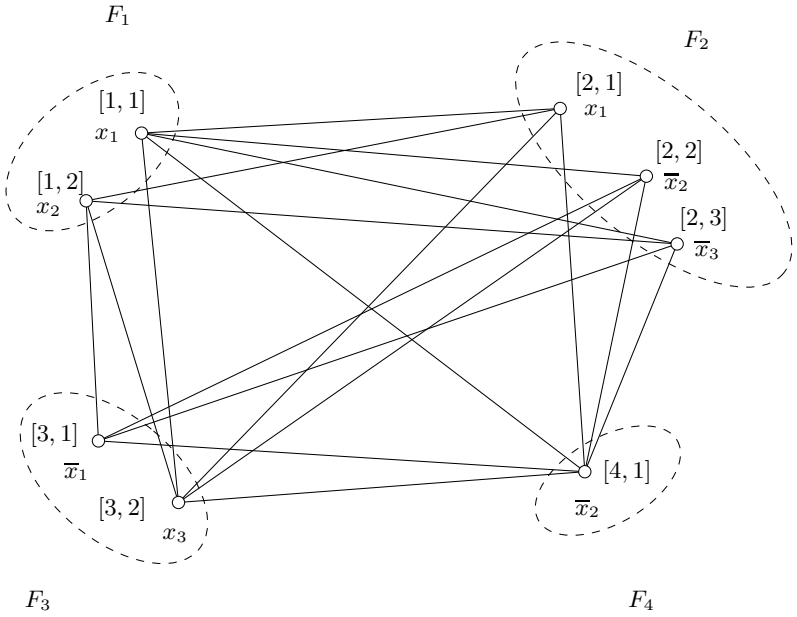


Рис. 6.6.

$$\{[1,1],[2,1],[3,2],[4,1]\},$$

приведённая на рис. 6.6, определяет подстановку $x_1 = 1$, $x_2 = 0$ ($\overline{x_2} = 1$) и $x_3 = 1$.

Покажем необходимую нам эквивалентность (6.1) путём доказательства соответствующих импликаций.

$\Rightarrow$ Пусть Φ – некоторая выполнимая формула. Тогда существует подстановка φ переменных формулы Φ, такая что $\varphi(\Phi) = 1$. Следовательно, $\varphi(F_i) = 1$ для всех $i \in \{1,\ldots,m\}$. Итак, для любого $i \in \{1,\ldots,m\}$ существует индекс $\alpha_i \in \{1,\ldots,k_i\}$, такой что $\varphi(l_{i\alpha_i}) = 1$. Докажем, что множество

$$\{[i,\alpha_i] \mid 1 \le i \le m\}$$

является кликой графа G. Отметим, что вершины $[1,\alpha_1]$, $[2,\alpha_2]$, $\ldots$, $[m,\alpha_m]$ соответствуют символам различных клауз.

Для любых i, j, таких что $i \ne j$, равенство $l_{i\alpha_i} = \overline{l}_{j\alpha_j}$ для каждой подстановки ω переменных формулы Φ влечёт неравенство $\omega(l_{i\alpha_i}) \ne \omega(l_{j\alpha_j})$. Поскольку $\varphi(l_{i\alpha_i}) = \varphi(l_{j\alpha_j}) = 1$ для всех пар $i, j \in \{1, \ldots m\}$, мы получаем, что неравенство $l_{i\alpha_i} \ne \overline{l}_{j\alpha_j}$ должно выполняться для всех пар (i,j), таких что $i \ne j$. Итак,

$$\{[i,\alpha_i],[j,\alpha_j]\} \in E$$

для всех $i, j \in \{1, \ldots, m\}$. Следовательно, $\{[i,\alpha_i] \mid 1 \le i \le m\}$ действительно является кликой размера m.

$\Leftarrow$ Пусть Q – некоторая клика графа G, имеющая $k = m$ вершин. Поскольку в G две вершины связаны только в том случае, когда они соответствуют элементам из

различных клауз, существуют $\alpha_1, \alpha_2, \ldots, \alpha_m$, $\alpha_p \in \{1, 2, \ldots, k_p\}$ для $p = 1, \ldots, m$, такие что

$$Q = \{[1, \alpha_1], [2, \alpha_2], \ldots, [m, \alpha_m]\}.$$

Согласно способу построения графа G, мы получаем существование некоторой подстановки φ переменных формулы Φ, такой что

$$\varphi(l_{1\alpha_1}) = \varphi(l_{2\alpha_2}) = \cdots = \varphi(l_{m\alpha_m}) = 1.$$

Отсюда непосредственно следует, что

$$\varphi(F_1) = \varphi(F_2) = \cdots = \varphi(F_m) = 1,$$

поэтому подстановка φ удовлетворяет формуле Φ, и эта формула является выполнимой. □

Лемма 6.56.

$$\text{CLIQUE} \leq_p \text{VC}.$$

Доказательство. Пусть граф $G = (V, E)$ и число k являются входом, представляющим собой частный случай проблемы клики. Создадим входные данные для частного случая проблемы вершинного покрытия следующим образом:

$m := |V| - k$;
$\overline{G} = (V, \overline{E})$, где $\overline{E} = \{\{v, u\} \mid u, v \in V, v \neq u, \{u, v\} \notin E\}$.

Рис. 6.7 иллюстрирует построение графа $\overline{G}$ по заданному графу G. Поскольку $\overline{G}$ может быть получен на основе G путём замены в матрице смежности графа G всех симолов 1 на 0 и всех 0 на 1, очевидно, что это построение может быть осуществлено за линейное время.

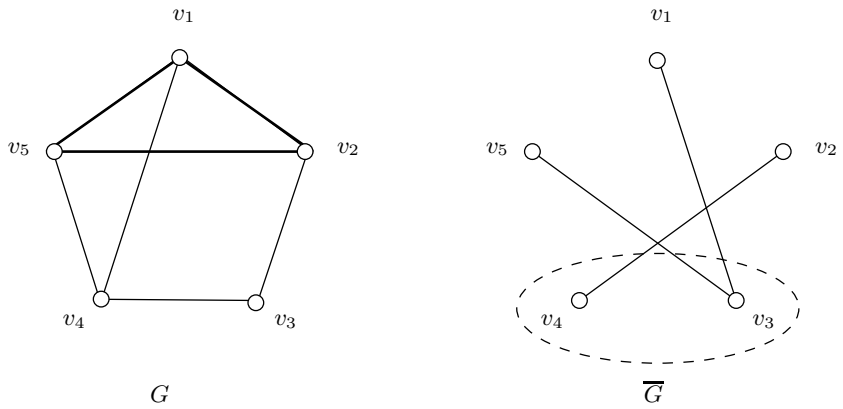


Рис. 6.7.

Для доказательства того, что

$$(G,k) \in \text{CLIQUE} \iff (\overline{G}, |V| - k) \in \text{VC}$$

нам достаточно показать следующее:

$$S \subseteq V \text{ – } \textit{клика графа } G \quad \Leftrightarrow \quad V - S \text{ – } \textit{вершинное покрытие графа } \overline{G}.$$

Продолжим рассмотрение примера, приведённого на рис. 6.7. Мы видим, что клика $\{v_1, v_2, v_5\}$ графа G определяет вершинное покрытие $\{v_3, v_4\}$ графа $\overline{G}$. Аналогично клика $\{v_1, v_4, v_6\}$ определяет вершинное покрытие $\{v_2, v_3\}$, а клика $\{v_2, v_3\}$ – вершинное покрытие $\{v_1, v_4, v_5\}$.

Покажем необходимую нам эквивалентность путём доказательства соответствующих импликаций.

$\Rightarrow$ Пусть S – некоторая клика графа G. Тогда в графе $\overline{G}$ ни одна пара вершин, принадлежащих S, не имеет ребра между ними. Последнее равносильно тому, что каждое ребро графа $\overline{G}$ содержит по крайней мере одну вершину множества $V - S$; поэтому $V - S$ – вершинное покрытие графа $\overline{G}$.

$\Leftarrow$ Пусть $C \subseteq V$ – некоторое вершинное покрытие графа $\overline{G}$. Согласно определению вершинного покрытия, каждое ребро графа $\overline{G}$ инцидентно по крайней мере одной вершине множества C. Следовательно, у графа $\overline{E}$ нет ни одного ребра $\{u, v\}$, такого что $u, v \in V - C$. Поэтому $\{u, v\} \in E$ для всех $u, v \in V - C$, $u \neq v$, а отсюда получаем, что $V - C$ – клика графа G. □

Следующий результат показывает, что проблема выполнимости остаётся трудной, даже если сократить множество её частных случаев до специального подкласса формул. Выше мы уже рассматривали определение 3 КНФ; Проблема принадлежности для 3 КНФ (проблема 3 SAT) заключается в определении того, является ли выполнимой некоторая формула, заданная в 3 КНФ.

Будем рассматривать подстановку φ Булевых переменных множества $X = \{x_1, \ldots, x_n\}$ как отображение $\varphi : X \to \{0, 1\}$. Пусть $Y = \{y_1, \ldots, y_r\}$ – некоторое множество Булевых переменных, причём $X \cap Y = \emptyset$. Будем говорить, что некоторое отображение $\omega : X \cup Y \to \{0, 1\}$ есть **расширение** отображения $\varphi : X \to \{0, 1\}$ до $X \cup Y$, если

$$\omega(z) = \varphi(z) \text{ для всех } z \in X.$$

Лемма 6.57.

$$\text{SAT} \leq_p 3\,\text{SAT}.$$

Доказательство. Пусть $F = F_1 \wedge F_2 \wedge \ \ldots \ \wedge F_m$ – некоторая формула в КНФ над множеством Булевых переменных $\{x_1, \ldots, x_n\}$. Создадим формулу C в 3КНФ, такую что формула F выполнима ($F \in \text{SAT}$) тогда и только тогда, когда формула C также выполнима ($C \in 3\,\text{SAT}$). Полиномиально-временна́я сводимость формулы F к C осуществляется для каждой клаузы $F_1, \ldots, F_m$ одним и тем же способом.

Если F_i содержит не более чем три символа, то мы полагаем $C_i = F_i$.

Иначе, пусть

$$F_i = z_1 \vee z_2 \vee \cdots \vee z_k, \quad \text{где } k \geq 4, \quad z_i \in \{x_1, \overline{x}_1, \cdots, x_n, \overline{x}_n\}.$$

Построим формулу C_i над множеством переменных

$$\{x_1, \ldots, x_n,\ y_{i,1}, y_{i,2}, \ldots, y_{i,k-3}\},$$

где $y_{i,1}, y_{i,2}, \ldots, y_{i,k-3}$ – некоторые новые переменные, не использовавшиеся в построении формул C_j при $j \neq i$. Положим

$$C_i = (z_1 \vee z_2 \vee y_{i,1}) \wedge (\overline{y}_{i,1} \vee z_3 \vee y_{i,2}) \wedge (\overline{y}_{i,2} \vee z_4 \vee y_{i,3}) \\ \wedge \cdots \wedge (\overline{y}_{i,k-4} \vee z_{k-2} \vee y_{i,k-3}) \wedge (\overline{y}_{i,k-3} \vee z_{k-1} \vee z_k).$$

Например, для $F_i = \overline{x}_1 \vee x_3 \vee \overline{x}_2 \vee x_7 \vee \overline{x}_9$ мы строим формулу следующим образом:

$$C_i = (\overline{x}_1 \vee x_3 \vee y_{i,1}) \wedge (\overline{y}_{i,1} \vee \overline{x}_2 \vee y_{i,2}) \wedge (\overline{y}_{i,2} \vee x_7 \vee \overline{x}_9).$$

Чтобы показать, что формула $F = F_1 \wedge \cdots \wedge F_m$ выполнима тогда и только тогда, когда формула $C = C_1 \wedge \cdots \wedge C_m$ также выполнима, нам достаточно доказать такое утверждение:

Подстановка φ переменных $\{x_1, \ldots, x_n\}$ удовлетворяет формуле F_i тогда и только тогда, когда существует некоторое расширение φ' отображения φ до $\{x_1, \ldots, x_n, y_{i,1}, \ldots y_{i,k-3}\}$, которое удовлетворяет формуле C_i.

Покажем эту эквивалентность путём доказательства соответствующих импликаций.

$\Rightarrow$ Пусть φ – некоторая подстановка переменных $\{x_1, x_2, \ldots, x_n\}$, такая что $\varphi(F_i) = 1$. Поэтому существует $j \in \{1, \ldots, k\}$, для которого $\varphi(z_j) = 1$. Выберем отображение

$$\varphi' : \{x_1, \ldots, x_n, y_{i,1}, \ldots, y_{i,k-3}\} \to \{0, 1\},$$

такое что:
- $\varphi(x_l) = \varphi'(x_l)$ для $l = 1, \ldots, n$;
- $\varphi'(y_{i,1}) = \cdots = \varphi'(y_{i,j-2}) = 1$;
- $\varphi'(y_{i,j-1}) = \cdots = \varphi'(y_{i,k-3}) = 0$.

Равенство $\varphi'(z_j) = 1$ влечёт выполнение $(j-1)$-й клаузы формулы C_i. Из $\varphi'(y_{i,r}) = 1$ для $r = 1, \ldots, j-2$ следует выполнение всех r-х клауз формулы C_i. Равенство $\varphi'(y_{i,s}) = 0$ (т. е. $\overline{y}_{i,s} = 1$) гарантирует выполнение $(s+1)$-х клауз формулы C_i для всех $s = j-1, j, \ldots, k-3$. Следовательно, отображение φ' удовлетворяет всем $k-2$ клаузам формулы C_i, т. е. $\varphi'(C_i) = 1$.

$\Leftarrow$ Докажем это утверждение от противного. Пусть φ – некоторая подстановка, такая что $\varphi(F_i) = 0$. Необходимо доказать, что не существует расширения φ' отображения φ, такого что $\varphi'(C_i) = 1$.
Равенство $\varphi(F_i) = 0$ влечёт

$$\varphi(z_1) = \varphi(z_2) = \cdots \varphi(z_k) = 0.$$

Для выполнения первой клаузы формулы C_i мы должны положить $y_{i,1} = 1$. Тогда $\varphi'(\overline{y}_{i,1}) = 0$, и для выполнения второй клаузы формулы C_i мы должны положить $\varphi'(y_{i,2}) = 1$. Используя те же самые доводы, мы получаем, что для выполнения первых $k-3$ клауз формулы C_i необходимо выполнение условий

$$\varphi'(y_{i,1}) = \varphi'(y_{i,2}) = \cdots = \varphi'(y_{i,k-3}) = 1.$$

Отсюда $\varphi'(\overline{y}_{i,k-3}) = 0$, и, поскольку $\varphi(z_{k-1}) = \varphi(z_k) = 0$, последняя клауза $\overline{y}_{i,k-3} \vee z_{k-1} \vee z_k$ формулы C_i остаётся невыполненной.

□

Упражнение 6.58. Докажите следующие полиномиально-временны́е сводимости:

- VC $\leq_p$ CLIQUE;
- 3 SAT $\leq_p$ VC.

Концепция NP-полноты стала основным инструментом классификации алгоритмических задач относительно их трудности. В настоящее время нам известно более чем 3000 NP-полных проблем. Ниже мы покажем, что мы можем развить этот подход – чтобы применить его для классификации оптимизационных проблем. Для этого нам прежде всего необходимо определить некоторые аналоги классов P и NP в случае оптимизационных проблем.

Определение 6.59. *Определим* **NPO** *как подкласс класса оптимизационных проблем, причём будем считать, что*

$$U = (\Sigma_I, \Sigma_O, L, \mathcal{M}, cost, goal) \in \text{NPO}$$

если выполняются следующие условия.

(1) $L \in P$.
{*Можно эффективно проверить, действительно ли слово* $x \in \Sigma_I^*$ *является допустимым входом – т. е. представлением некоторого частного случая проблемы* U.}
(2) Существует полиномиальная функция p_U, *такая что:*
(a) для всех $x \in L$ *и* $y \in \mathcal{M}(x)$ *выполнено неравенство* $|y| \leq p_U(|x|)$;
{*Размер*[33] *любого допустимого решения является полиномиальным относительно длины входа.*}
(b) существует полиномиально-временной алгоритм A, *который для любых* $y \in \Sigma_O^*$ *и* $x \in L$, *таких что* $|y| \leq p_U(|x|)$, *определяет, действительно ли* $y \in \mathcal{M}(x)$.
(3) Функция cost может быть вычислена за полиномиальное время.

Иными словами – некоторая оптимизационная проблема U находится в классе NPO при выполнении следующих условий.

- Можно эффективно проверить, является ли данное слово частным случаем проблемы U – т. е. трудность проблемы U не зависит от решения проблемы принадлежности (Σ_I, L).
- Размер каждого из допустимых решений для некоторого входа полиномиально ограничен размером этого входа, и при этом можно эффективно проверить, действительно ли некоторое решение-кандидат является допустимым – т. е. трудность проблемы U не зависит от решения проблемы принадлежности $(\Sigma_O, \mathcal{M}(x))$ для любого входа x.
- Можно эффективно вычислить стоимость любого допустимого решения – т. е. вычисление «качества» допустимого решения также не влияет на трудность проблемы U.

Заметим, что условие (2.b) – главная аналогия между классами NPO и VP. Но самый важный момент здесь заключается в том, что все три условия, (1), (2) и (3), являются «естественными» – поскольку они:

[33] Т. е. длина представления.

- сводят трудность решения некоторой оптимизационной проблемы U к простому процессу оптимизации – т. е. к поиску лучших решений среди всех допустимых;
- делают трудность проблемы U независимой от проблем принадлежности – т. е. к проверкам, действительно ли некоторый вход представляет собой частный случай проблемы проблемы U, а некоторое решение является решением-кандидатом для заданного входа.

Следующие рассуждения показывают, что в класс NPO входит проблема MAX-SAT.

- Можно эффективно определить, действительно ли некоторое слово $x \in \Sigma^*_{\text{logic}}$ кодирует Булеву формулу, заданную в КНФ.
- Для любого входа x каждая подстановка значений переменных формулы Φ_x (пусть это – некоторая подстановка $\alpha \in \{0,1\}^*$) обладает свойством $|\alpha| < |x|$. При этом мы можем за линейное время проверить, действительно ли $|\alpha|$ является числом переменных формулы Φ_x.
- Для любой такой подстановки мы можем вычислить число выполняющихся клауз формулы Φ_x – за время, линейно зависящее от $|x|$. Иными словами, мы можем эффективно вычислить стоимость подстановки.

Теперь рассмотрим следующие оптимизационные проблемы. **Проблема максимального разреза (MAX-CUT)** заключается в том, чтобы для заданного графа $G = (V, E)$ найти разрез на 2 части, имеющий максимальную стоимость. При этом **разрезом** графа $G = (V, E)$ является любая пара (V_1, V_2), такая что

$$V_1 \cup V_2 = V, \quad \text{а} \quad V_1 \cap V_2 = \emptyset,$$

а стоимость этого разреза равна числу рёбер между V_1 и V_2, т. е.

$$cost(V_1, V_2) = |E \cap \{\{v, u\} \mid v \in V_1, u \in V_2\}|.$$

Проблема минимального вершинного покрытия (MIN-VC) заключается в том, чтобы для заданного графа $G = (V, E)$ найти вершинное покрытие минимальной мощности.

Упражнение 6.60. Приведите формальные определения оптимизационных проблем MAX-CUT и MIN-VC. Покажите, что обе эти проблемы принадлежат классу NPO.

Следующее определение вводит класс PO таких оптимизационных проблем, которые является естественной аналогией класса P.

Определение 6.61. *Определим* **PO** *как подкласс класса оптимизационных проблем, причём*

$$U = (\Sigma_I, \Sigma_O, L, \mathcal{M}, cost, goal) \in \text{PO}$$

если выполняются следующие условия:

- $U \in NPO$;
- *существует полиномиально-временной алгоритм A, такой что для каждого входа $x \in L$ слово $A(x)$ является оптимальным решением для этого входа.*

Теперь получим понятие NP-трудности оптимизационных проблем – просто рассмотрев сводимость конкретных проблем принадлежности к оптимизационным проблемам.

Определение 6.62. *Пусть* $U = (\Sigma_I, \Sigma_0, L, \mathcal{M}, cost, goal)$ *– некоторая оптимизационная проблема класса* NPO. *Определим* **пороговый язык проблемы** $\boldsymbol{U}$ *как*

$$\boldsymbol{Lang_U} = \{(x, a) \in L \times \Sigma^*_{\text{bool}} \mid Opt_U(x) \leq Number(a)\},$$

если $goal = minimum$, *и*

$$\boldsymbol{Lang_U} = \{(x, a) \in L \times \Sigma^*_{\text{bool}} \mid Opt_U(x) \geq Number(a)\},$$

если $goal = maximum$.

Будем говорить, что проблема U *является* **NP-трудной**, *если* NP-*трудным является язык* $Lang_U$.

Теперь покажем, что на основе определения 6.62 мы получаем специальный подход к проверке трудности оптимизационных проблем. Точнее, доказательство NP-трудности языка $Lang_U$ является возможным способом доказательства факта $U \notin$ PO – при дополнительно сделанном предположении P $\neq$ NP. Важный момент этого подхода заключается в том, что в данном случае не нужно делать других дополнительных предположений – вроде PO $\neq$ NPO и т.п.

Лемма 6.63. *Если для некоторой оптимизационной проблемы выполнено включение* $U \in$ PO, *то* $Lang_U \in$ P.

Доказательство. Если $U \in$ PO, то существует полиномиально-временной алгоритм A, который для каждого входа – т. е. для каждого частного случая x проблемы U – вычисляет оптимальное решение для x, и, следовательно, значение $Opt_U(x)$. Поэтому алгоритм A может быть использован и для решения проблемы принадлежности для языка $Lang_U$. □

Теорема 6.64. *Пусть* $U \in$ NPO. *Если* $Lang_U$ *является* NP-*трудным и* P $\neq$ NP, *то* $U \notin$ PO.

Доказательство. Приведём доказательство от противного – т. е. предположим, что $U \in$ PO. Применяя лемму 6.63, мы получаем, что $Lang_U \in$ P. Поскольку язык $Lang_U$ является NP-трудным, условие $Lang_U \in$ P влечёт равенство P = NP. □

Описанный нами метод доказательства трудности оптимизационных проблем весьма прост. Для иллюстрации этого факта приведём следующие примеры.

Лемма 6.65. *Проблема* MAX-SAT *является* NP-*трудной.*

Доказательство. Согласно определению 6.62, мы должны доказать NP-трудность языка $Lang_{\text{MAX-SAT}}$. Поскольку мы знаем, что SAT является NP-трудной, достаточно доказать следующее:

$$\text{SAT} \leq_p Lang_{\text{MAX-SAT}}.$$

Пусть x кодирует формулу Φ_x, состоящую из m клауз. Возьмём в качестве входа (т. е. частного случая проблемы принадлежности $(Lang_{\text{MAX-SAT}}, \Sigma^*)$) пару (x, m). При этом очевидно, что

$$(x, m) \in Lang_{\text{MAX-SAT}} \iff \Phi_x \text{ формула является выполнимой.}$$

□

Проблема максимальной клики (MAX-CL) заключается в том, чтобы в заданном графе G найти клику максимального размера.

Лемма 6.66. MAX-CL *является* NP-*трудной.*

Доказательство очевидно: поскольку CLIQUE $= Lang_{\text{MAX-CL}}$, и при этом проблема CLIQUE – NP-трудная, MAX-CL также является NP-трудной. □

Упражнение 6.67. Докажите, что проблемы MAX-CUT и MIN-VC являются NP-трудными.

6.7 Заключение

Основной целью теории сложности является классификация алгоритмических проблем относительно необходимых для их решения ресурсов компьютера. Результаты исследований этой теории – получение количественных законов теории обработки информации, а также описание границ практической алгоритмической разрешимости («лёгкости» решения).

Основными единицами измерения сложности являются сложность по времени (временна́я сложность) и сложность по памяти. А основной вычислительной моделью абстрактной теории сложности является многоленточная машина Тьюринга. Сложность некоторой ММТ (некоторого алгоритма) рассматривается как функция f, зависящая от длины входа – причём $f(n)$ определяется как максимум сложностей всех вычислений с длиной входной информации n. Этот вариант измерения сложности называется сложностью в худшем.

Существуют алгоритмически разрешимые задачи с произвольно большой сложностью. Алгоритмы экспоненциальной сложности обычно рассматриваются как невыполнимые на практике.

Лёгкость решения проблемы связана с полиномиально-временно́й сложностью. Класс P – это множество всех проблем принадлежности, которые могут быть решены с помощью полиномиально-временны́х алгоритмов. Определение класса P является «жёстким» – в том смысле, что этот класс не зависит от выбора конкретной вычислительной модели.

Временна́я сложность недетерминированной ММТ M над входным словом w – это длина самого короткого корректного вычисления машины M над w. Типичная работа недетерминированного алгоритма начинается с недетерминированных предположений и продолжается детерминированной проверкой каждого из этих предположений. Класс NP – это класс всех языков, которые могут быть приняты полиномиально-временны́ми недетерминированными алгоритмами. Вопрос, является ли класс P *собственным* подмножеством класса NP – это самая известная нерешённая проблема теоретической информатики. Класс NP включает множество интересных для практики задач, про которые неизвестно, находятся ли они в классе P. Можно показать, что вопрос о том, является ли класс P собственным подмножеством класса NP, эквивалентен вопросу, легче ли *проверка* корректности данного математического доказательства чем *создание* доказательства для данной теоремы.

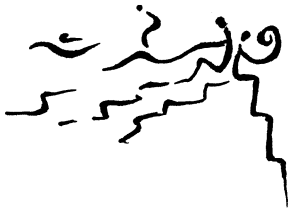
До сих пор нет существенных достижений в получении нижних пределов сложности конкретных проблем – и поэтому нам необходима методология для классификации проблем, разделения их на легкорешаемые и труднорешаемые. Концепция NP-полноты позволяет нам доказывать результаты типа $L \notin \mathrm{P}$ (язык L не является легкорешаемым) – при дополнительном предположении $\mathrm{P} \neq \mathrm{NP}$. Основная идея этого подхода заключается в том, что NP-полные проблемы являются наиболее трудными проблемами класса NP – в том смысле, что если бы некоторая NP-полная проблема принадлежала бы классу P, то последний класс был бы эквивалентен классу NP.

Иерархии сложности были введены Хартманисом, Стернсом и Льюисом [24, 25]. Концепции полиномиально-временно́й сводимости и NP-полноты были введены в конструктивных работах Кука [13] и Карпа [35]. Превосходная классическая книга Гэри и Джонсона [19] даёт подробное представление теории NP-полноты. Интересные рассуждения о «практической разрешимости» можно найти у Льюиса и Пападимитриу [42], а также у Стокмэйера и Чандры [66].

Существует несколько хороших учебников, посвящённых теории сложности. Превосходным введением в эту теорию являются учебники Хопкрофта и Ульмана [28], а также Сипсера [65]. Всесторонее её изложение дано, например, Боветом и Крешензи [7], Пападимитриу [49], а также Балказаром, Диазом и Габорро [2, 3]. Отметим также учебник Рейшука [56].

Когда учёный говорит:
«Это – самый конец,
никто здесь больше ничего не сделает. . . » –
это не учёный.

Л. Гоулд



7

Алгоритмизация труднорешаемых задач

7.1 Цели и задачи главы

Теория сложности обеспечивает методы классификации алгоритмических проблем относительно их трудности – которая измеряется количеством вычислительных ресурсов, необходимых для их решения. А *теория алгоритмов* посвящена разработке эффективных алгоритмов для решения конкретных проблем.

В этой главе мы ограничимся разработкой алгоритмов для сложных (например, NP-трудных) проблем. Это может показаться несколько неожиданным – поскольку в главе 6 было установлено, что решение NP-трудных проблем находится за границами физических возможностей компьютера. Например, выполнение некоторого алгоритма с временно́й сложностью 2^n для входов длиной $n = 100$ требует больше времени, чем возраст известной части Вселенной.

Однако существуют многие трудные проблемы, имеющие огромную важность для ежедневной практики. Существование таких проблем объясняет, почему специалисты в области теоретической информатики в течение уже по крайней мере 40 лет пытаются хоть немного продвинуться в их решении. Основная идея здесь заключается в том, чтобы сделать трудные проблемы проще (разрешимыми за полиномиальное время) – путём небольшого изменения спецификаций проблемы или ослабления некоторых ограничений на её входные данные.

Настоящее искусство создания алгоритмов состоит в том, чтобы находить различные возможности для получения максимальной пользы от осуществимых на практике вычислений, сокращая время их выполнения до нескольких минут на обычном компьютере, – но при этом как можно меньше измененяя накладываемые на задачу ограничения. Для достижения подобных эффектов, позволяющих добиваться практической разрешимости трудных проблем, можно использовать один из следующих подходов – или некоторую их комбинацию.

1. *Рассмотрение алгоритмов, предназначенных для обычных частных случаев проблемы – вместо наиболее сложных.*
   Мы измеряем временну́ю сложность как сложность в худшем случае – это означает, что временна́я сложность $\mathrm{Time}(n)$ есть сложность наиболее трудных частных случаев задачи, имеющих размерность n. Часто частные случаи наиболее трудных проблем далеки от того, чтобы их можно было назвать «естественными», репрезентативными – и, следовательно, они вряд ли могут встретиться на практике.

Поэтому может быть полезной классификация частных случаев проблемы – также относительно их трудности. Удачная классификация может привести к выделению большого подкласса эффективно разрешимых частных случаев трудной задачи. И если в некотором приложении *типичные* частные случаи проблемы относятся к такому лёгкому подклассу – то для этого приложения мы можем эффективно её решать.

2. *Применение экспоненциальных алгоритмов.*
Наша цель – не разработка полиномиальных алгоритмов решения трудных задач, а поиск тех экспоненциальных алгоритмов, которые всё же возможно применить на практике. Основная идея при этом основывается на том, что некоторые показательные функции не принимают слишком больших значений для реальных размеров входа. Таблица 7.1 ясно показывает, что выполнение $(1.2)^n$ машинных операций для $n = 100$, или $10 \cdot 2^{\sqrt{n}}$ операций для $n = 300$, заканчивается в пределах нескольких секунд.

Таблица 7.1.

Сложность	$n = 10$	$n = 50$	$n = 100$	$n = 300$
2^n	1024	16 цифр	31 цифр	91 цифр
$2^{\frac{n}{2}}$	32	$\approx 33 \cdot 10^6$	16 цифр	46 цифр
$(1.2)^n$	≈ 6	≈ 9100	$\approx 83 \cdot 10^6$	24 цифр
$10 \cdot 2^{\sqrt{n}}$	≈ 89	≈ 1345	10240	$\approx 1.64 \cdot 10^6$

3. *Ослабление ограничений, накладываемых на частные случаи проблемы.*
Мы можем несколько отклониться от имеющихся ограничений, причём даже различными способами – чтобы гарантировать корректный результат вычислений. Типичные представители такого подхода – рандомизированные и аппроксимационные алгоритмы.
В случае рандомизированных алгоритмов мы заменяем детерминированное выполнение программы на рандомизированное (недетерминированное, вероятностное). «Минусом» такой замены является отсутствие уверенности в том, что результат работы алгоритма корректен – т. е. ненулевая вероятность получения неверного результата. А её «плюс» заключается в существенном уменьшении сложности вычислений. Если вероятность ошибки для каждого входа меньше, чем 10^{-9}, то не существует практической разницы между надёжностью детерминированного и рандомизированного алгоритмов. Описанным здесь способом можно эффективно решать многие проблемы.
Аппроксимационные алгоритмы используются для решения оптимизационных проблем. Вместо требования *оптимальности* решения мы допускаем *выполнимые* решения – чья стоимость (качество), однако, не слишком отличается от стоимости оптимального – т. е. не намного хуже него. «Плюсом» такого подхода может быть экспоненциальное уменьшение вычислительной сложности.

Целью данной главы является описание некоторых конкретных приложений этих подходов. В разделе 7.2 мы вводим псевдополиномиальные алгоритмы, которые являются характерным примером поиска важных для практики классов лёгких частных случаев трудных проблем. Концепция аппроксимационных алгоритмов объясняется и

иллюстрируется в разделе 7.3. Раздел 7.4 посвящён локальным алгоритмам – которые дают возможность применения всех трёх вышеупомянутых подходов.[1] Локальный поиск является основой для нескольких эвристик. В разделе 7.5 мы представим эвристический алгоритм имитационной нормализации, который основан на аналогии с оптимизацией некоторых физических систем. А концепция рандомизации, по-видимому, наиболее важна – и поэтому мы посвятим ей всю следующую главу.

7.2 Псевдополиномиальные алгоритмы

В этом разделе мы имеем дело со специальным классом задач, входы которых могут рассматриваться как последовательности целых чисел. Такие задачи называются **целочисленными проблемами**.

Фактически мы можем интерпретировать как вход целочисленной проблемы любое слово языка $\{0,1,\#\}^*$, имеющее произвольное число символов $\#$. Пусть

$$x = x_1\#x_2\#\ldots\#x_n,\ x_i \in \{0,1\}^* \ \text{ для } i = 1,2,\ldots,n$$

– некоторое слово над алфавитом $\{0,1,\#\}^*$. Мы можем интерпретировать x как вектор

$$\mathbf{Int}(\boldsymbol{x}) = (\mathit{Number}(x_1), \mathit{Number}(x_2), \ldots, \mathit{Number}(x_n)),$$

состоящий из n натуральных чисел. Каждая проблема теории графов, частные случаи которой могут быть заданы с помощью матриц смежности,[2] может рассматриваться как целочисленная проблема – поскольку каждая матрица смежности может быть представлена как некоторая последовательность, состоящая из символов 0 и 1. Например, целочисленной проблемой является TSP – в ней целые числа входной последовательности представляют собой стоимости конкретных рёбер.

Для любого $x = x_1\#x_2\#\ldots\#x_n$, где $x_i \in \{0,1\}^*$ для $i = 1,2,\ldots,n$, определим

$$\mathbf{MaxInt}(\boldsymbol{x}) = \max\{\mathit{Number}(x_i) \mid i = 1,2,\ldots,n\}.$$

Идея концепции псевдополиномиальных алгоритмов заключается в следующем. Мы разрабатываем алгоритмы, являющиеся эффективными для таких частных случаев x рассматриваемых проблем, у которых значение $\mathrm{MaxInt}(x)$ может только незначительно превышать $|x|$. Поскольку для слова y число $\mathit{Number}(y)$ экспоненциально растёт в зависимости от длины двоичного представления y, это требование формулирует важное ограничение.[3]

---

[1] То есть – рассмотрение обычных частных случаев проблемы (а не наиболее сложных), применение экспоненциальных алгоритмов для реальных размеров входа и ослабление ограничений, накладываемых на частные случаи проблемы.

[2] Проблема вершинного покрытия, проблема максимальной клики, и мн. др.

[3] Сам термин «псевдополиномиальные» («псевдо-полиномиально-временны́е») алгоритмы можно объяснить и немного по-другому. Мы разделяем множества всех частных случаев трудной проблемы на подмножества «лёгких» и «трудных» – с идеей разработки эффективных алгоритмов для лёгких частных случаев. Именно для лёгких входов мы и разрабатываем полиномиально-временны́е алгоритмы – и поэтому для всей проблемы применяется префикс «псевдо». *(Прим. перев.)*

Определение 7.1. *Пусть $\mathcal{U}$ – некоторая целочисленная проблема, а A – некоторый алгоритм для её решения. Будем говорить, что A –* **псевдополиномиальный алгоритм для решения $\mathcal{U}$**, *если существует полиномиальная функция p, зависящая от двух переменных, такая что для всех частных случаев x проблемы $\mathcal{U}$*

$$\mathrm{Time}_A(x) \in O(p(|x|, \mathrm{MaxInt}(x))).$$

Заметим, что если задана некоторая полиномиальная функция h, то для частных случаев проблемы, таких что $\mathrm{MaxInt}(x) \leq h(|x|)$, временна́я сложность $\mathrm{Time}_A(x)$ является полиномиальной относительно $|x|$.

Определение 7.2. *Пусть $\mathcal{U}$ – некоторая целочисленная проблема, а h – некоторое отображение, действующее из $\mathbb{N}_0$ в $\mathbb{N}_0$. Назовём* **h-ограниченным вариантом проблемы $\mathcal{U}$ (Value(h)-$\mathcal{U}$)** *такой, у которого все частные случаи удовлетворяют условию*

$$\mathrm{MaxInt}(x) \leq h(|x|).$$

Следующая теорема показывает, как мы можем строго определять большие классы простых частных случаев некоторой трудной проблемы.

Теорема 7.3. *Пусть $\mathcal{U}$ – некоторая целочисленная проблема, а A – некоторый псевдополиномиальный алгоритм для её решения. Тогда для любой полиномиальной функции h существует некоторый полиномиально-временно́й алгоритм, решающий проблему* Value(h)-$\mathcal{U}$.[4]

Доказательство. Если A – псевдополиномиальный алгоритм для проблемы $\mathcal{U}$, то существует некоторая полиномиальная функция p, зависящая от двух переменных, такая что для любого частного случая x проблемы $\mathcal{U}$

$$\mathrm{Time}_A(x) \in O(p(|x|, \mathrm{MaxInt}(x)).$$

Поскольку h – полиномиальная функция, существует некоторая константа c, такая что для всех частных случаев x проблемы Value(h)-$\mathcal{U}$

$$\mathrm{MaxInt}(x) \in O(|x|^c).$$

Следовательно,

$$\mathrm{Time}_A(x) \in O(p(|x|, |x|^c)).$$

Поэтому A является полиномиально-временны́м алгоритмом для проблемы Value(h)-$\mathcal{U}$. □

Покажем применение концепции псевдополиномиальных алгоритмов для решения проблемы рюкзака – одной из NP-трудных оптимизационных проблем.

Проблема рюкзака

[4] При этом если $\mathcal{U}$ – проблема принадлежности, то Value(h)-$\mathcal{U} \in$ P. А если $\mathcal{U}$ – оптимизационная проблема, то Value(h)-$\mathcal{U} \in$ PO.

Вход: $2n+1$ натуральных чисел $w_1, w_2, \ldots, w_n, c_1, c_2, \ldots, c_n, b$ для некоторого $n \in \mathbb{N}$.
{Эти натуральные числа дают информацию об n объектах, при этом w_i – вес i-го объекта, а c_i – его стоимость; $i = 1, 2, \ldots, n$. Число b является пределом (верхней границей) веса всего содержимого рюкзака. А само содержимое является подмножеством множества заданных n объектов.}

Допустимые решения: каждая последовательность $I = (w_1, w_2, \ldots, w_n,\ c_1, \ldots, c_n, b)$ определяет множество допустимых решений

$$\mathcal{M}(I) = \left\{ T \subseteq \{1, \ldots, n\} \;\middle|\; \sum_{i \in T} w_i \leq b \right\}.$$

{Допустимым решением для некоторой последовательности I является любое подмножество её объектов, общий вес которых не превышает b.}

Стоимость: для всех входов I и всех $T \in \mathcal{M}(I)$

$$cost(T, I) = \sum_{i \in T} c_i.$$

{Стоимость допустимого решения T является общей стоимостью всех объектов, содержащихся в рюкзаке.}

Цель: maximum.

Теперь для проблемы рюкзака разработаем псевдополиномиальный алгоритм, используя для этого метод динамического программирования. Для вычисления оптимального решения частного случая проблемы, описываемого последовательностью $I = (w_1, w_2, \ldots, w_n, c_1, \ldots, c_n, b)$, мы начинаем с рассмотрения упрощённого варианта этого частного случая: $I_1 = (w_1, c_1, b)$. Далее мы продолжаем рассматривать последовательность подобных упрощённых вариантов

$$I_i = (w_1, w_2, \ldots, w_i, c_1, \ldots, c_i, b)$$

для $i = 2, 3, \ldots, n$ – пока не достигнем последовательности $I = I_n$.

Немного подробнее. Для каждой последовательности I_i и для каждого $k \in \{0, 1, 2, \ldots \sum_{j=1}^{i} c_j\}$ мы вычисляем тройку

$$(k, W_{i,k}, T_{i,k}) \in \left\{ 0, 1, 2, \ldots, \sum_{j=1}^{i} c_j \right\} \times \{0, 1, 2, \ldots, b\} \times \mathcal{P}(\{1, \ldots, i\});$$

здесь $W_{i,k}$ – минимальный вес, с которым мы можем достичь стоимости k для частного случая проблемы, описываемого последовательностью I_i. Множество $T_{i,k} \subseteq \{1, \ldots, i\}$ является множеством индексов, которое определяет стоимость k с весом $W_{i,k}$ – т. е.

$$\sum_{j \in T_{i,k}} c_j = k \quad \text{и} \quad \sum_{j \in T_{i,k}} w_j = W_{i,k}.$$

Заметим, что может существовать несколько различных множеств индексов, которые определяют одну и тот же стоимость k и один и тот же вес $W_{i,k}$. В подобных случаях мы просто выбираем произвольное из этих множеств – для подтверждения достижимости

пары $(k, W_{i,k})$.[5] С другой стороны, может случиться, что стоимость k для частного случая I_i недостижима – в таком случае троек с первым элементом k не существует.

Обозначим ножество всех троек, определённых таким образом для последовательности I_i, записью $\mathbf{TRIPLE_i}$. Заметим, что

$$|\mathrm{TRIPLE}_i| \leq \sum_{j=1}^{i} c_j + 1.$$

При этом важно отметить, что мы можем вычислить значение TRIPLE_{i+1} на основе TRIPLE_i за время $O(\mathrm{TRIPLE}_i)$.

Для формирования множества TRIPLE_{i+1} мы в первую очередь вычисляем множество

$$\mathrm{SET}_{i+1} = \mathrm{TRIPLE}_i \cup \{(k + c_{i+1}, W_{i,k} + w_{i+1}, T_{i,k} \cup \{i+1\}) \mid (k, W_{i,k}, T_{i,k}) \in \mathrm{TRIPLE}_i \text{ и } W_{i,k} + w_{i+1} \leq b\}$$

– путём добавления $(i+1)$-го объекта в любую тройку множества TRIPLE_i, если, однако, это добавление не приведёт к тому, что общая стоимость превысит значение b.

Теперь TRIPLE_{i+1} является подмножеством SET_{i+1} – таким, что оно получено путём выбора некоторой тройки минимального веса для каждой достижимой стоимости k. Очевидно, что тройка множества TRIPLE_n с максимальной стоимостью даёт оптимальное решение для частного случая проблемы $I = I_n$. Итак, мы может формально описать разработанный алгоритм следующим образом.

Алгоритм DPR

Вход: $I = (w_1, w_2, \ldots, w_n, c_1, \ldots, c_n, b) \in \mathbb{N}^{2n+1}$ для некоторого $n \in \mathbb{N}$.
Этап 1. $\mathrm{TRIPLE}(1) = \{(0, 0, \emptyset)\} \cup \{(c_1, w_1, \{1\})\} \mid$ если $w_1 \leq b\}$.
Этап 2.

```
for i = 1 to n − 1 do
  begin SET(i + 1) := TRIPLE(i);
    for (для) каждой пары (k, w, T) ∈ TRIPLE(i) do
      begin if w + w_{i+1} ≤ b then
        SET(i + 1) := SET(i + 1)
                      ∪ {(k + c_{i+1}, w + w_{i+1}, T ∪ {i + 1})};
      end
    определить TRIPLE(i + 1) −
    как подмножество множества SET(i + 1),
    где для каждой достижимой стоимости k
    множество SET(i + 1) содержит в точности в точности
    одну тройку стоимости k – путём выбора
    для рассматриваемого k тройки минимального веса.
  end
```

Этап 3. Вычислить

$$c := \max\{k \in \{1, 2, \ldots, \sum_{i=1}^{n} c_i\} \mid (k, w, T) \in \mathrm{TRIPLE}(n)\}.$$

[5] Т. е. достижимости стоимости k с весом $W_{i,k}$.

Выход: Множество индексов T, такое что $(c, w, T) \in \text{TRIPLE}(n)$.

Проиллюстрируем работу алгоритма DPR для частного случая проблемы $I = (w_1, w_2, \ldots, w_5, c_1, \ldots, c_5, b)$, где:

$$w_1 = 23,\ w_2 = 15,\ w_3 = 15,\ w_4 = 33,\ w_5 = 32,$$
$$c_1 = 33,\ \ c_2 = 23,\ \ c_3 = 11,\ \ c_4 = 35,\ \ c_5 = 11,\ \text{и}$$
$$b = 65.$$

Очевидно, что $I_1 = (23, 33, 65)$, и достижимыми стоимостями являются только 0 и 33. Отсюда

$$\text{TRIPLE}_1 = \{(0, 0, \emptyset),\ (33, 23, \{1\})\}.$$

$I_2 = (23, 15, 33, 23, 65)$, и достижимыми стоимостями для I_2 являются значения 0, 23, 33, и 56. Следовательно, мы получаем

$$\text{TRIPLE}_2 = \{(0, 0, \emptyset),\ (23, 15, \{2\}),\ (33, 23, \{1\}),\ (56, 38, \{1, 2\})\}.$$

Далее, $I_3 = (23, 15, 15, 33, 23, 11, 65)$. Добавление третьего объекта возможно для любой из троек множества TRIPLE_2 – сделав это, мы всегда получаем новую стоимость. Итак, число элементов множества TRIPLE_3 в 2 раза больше, чем число элементов множества TRIPLE_2 (т. е. $\text{SET}_3 = \text{TRIPLE}_3$), и

$$\begin{aligned}\text{TRIPLE}_3 = \{&(0, 0, \emptyset),\ (11, 15, \{3\}),\ (23, 15, \{2\}),\\ &(33, 23, \{1\}),\ (34, 30, \{2, 3\}),\ (44, 38, \{1, 3\}),\\ &(56, 38, \{1, 2\}),\ (67, 53, \{1, 2, 3\})\}.\end{aligned}$$

Для тройки $(44, 38, \{1, 3\})$, $(56, 38, \{1, 2\})$ и $(67, 53, \{1, 2, 3\})$ из множества TRIPLE_3 мы не можем упаковать в рюкзак четвёртый объект. Следовательно,

$$\begin{aligned}\text{TRIPLE}_4 = \text{TRIPLE}_3 \cup \{&(35, 33, \{4\}),\ (46, 48, \{3, 4\}),\\ &(58, 48, \{2, 4\}),\ (68, 56, \{1, 4\}),\ (69, 63, \{2, 3, 4\})\}.\end{aligned}$$

Наконец, для $I = I_5$ мы получаем

$$\begin{aligned}\text{TRIPLE}_5 = \{&(0, 0, \emptyset),\ (11, 15, \{3\}),\ (22, 47, \{3, 5\}),\\ &(23, 15, \{2\}),\ (33, 23, \{1\}),\ (34, 30, \{2, 3\}),\\ &(35, 33, \{4\}),\ (44, 38, \{1, 3\}),\ (45, 62, \{2, 3, 5\}),\\ &(46, 48, \{3, 4\}),\ (56, 38, \{1, 2\}),\ (58, 48, \{2, 4\}),\\ &(67, 53, \{1, 2, 3\}),\ (68, 56, \{1, 4\}),\ (69, 63, \{2, 3, 4\})\}.\end{aligned}$$

Следовательно $\{2, 3, 4\}$ – оптимальное решение для I, поскольку $(69, 63, \{2, 3, 4\})$ – тройка множества TRIPLE_5 с максимально возможной стоимостью 69.

Упражнение 7.4. Продемонстрируйте работу алгоритма DPR для частного случая проблемы рюкзака $(1, 3, 5, 6, 7, 4, 8, 5, 9)$.

Теперь проанализируем временну́ю сложность алгоритма DPR.

Теорема 7.5. *Для каждого частного случая I проблемы рюкзака выполнено следующее:*

$$\mathrm{Time}_{\mathrm{DPR}}(I) \in O(|I|^2 \cdot \mathrm{MaxInt}(I)),$$

и поэтому DPR *– псевдополиномиальный алгоритм для проблемы рюкзака.*

Доказательство. Проанализируем отдельно временну́ю сложность каждого из трёх этапов алгоритма.

1. Для первого этапа сложность равна $O(1)$.
2. Для частного случая проблемы $I = (w_1, w_2, \ldots, w_n, c_1, \ldots, c_n, b)$ алгоритм DPR строит $n-1$ множеств TRIPLE$(i+1)$. Построение множества TRIPLE$(i+1)$ по заданному TRIPLE(i) может быть осуществлено за время $O(|\mathrm{TRIPLE}(i+1)|)$. Поскольку для каждого $i \in \{0, 1, \ldots, n\}$

$$|\mathrm{TRIPLE}(i+1)| \leq \sum_{j=1}^{n} c_j \leq n \cdot \mathrm{MaxInt}(I),$$

   т. е. временна́я сложность второго этапа – $O(n^2 \cdot \mathrm{MaxInt}(I))$.
3. Временна́я сложность третьего этапа находится в пределах $O(n \cdot \mathrm{MaxInt}(I))$ – поскольку для вычисления максимума мы должны рассмотреть не более чем $n \cdot \mathrm{MaxInt}(I)$ значений.

Поскольку $n \leq |I|$, мы получаем, что

$$\mathrm{Time}_{\mathrm{DPR}}(I) \in O(|I|^2 \cdot \mathrm{MaxInt}(I)).$$

□

Заметим, что псевдополиномиальные алгоритмы могут быть весьма успешными во многих приложениях. Веса и стоимости обычно принадлежат некоторым заранее фиксированным интервалам – и поэтому являются независимыми от числа аргументов частных случаев проблемы. Следовательно, мы можем эффективно вычислять решения для типичных[6] частных случаев проблемы рюкзака.

При разработке псевдополиномиальных алгоритмов также представляет интерес вопрос их классификации:

- для каких NP-трудных целочисленных проблем существуют псевдополиномиальные алгоритмы?
- какие NP-трудные целочисленные проблемы трудны настолько, что не допускают псевдополиномиальных алгоритмов для своего решения?

Рассмотрим метод, позволяющий нам доказать, что некоторые конкретные проблемы действительно не допускают псевдополиномиальных алгоритмов – при сделанном предположении $\mathrm{P} \neq \mathrm{NP}$. Мы увидим, что концепция NP-полноты работает и для этой цели.

Определение 7.6. *Целочисленная проблема $\mathcal{U}$ называется* **сильно NP-трудной** *(strongly* NP*-hard), если существует полиномиальная функция p, такая что проблема* Value(p)*-$\mathcal{U}$ является* NP*-трудной.*

[6] В вышеуказанном смысле.

Следующая теорема показывает, что концепция сильной NP-трудности даёт инструмент для доказательства несуществования псевдополиномиальных алгоритмов для конкретных проблем.

Теорема 7.7. *Пусть $\mathcal{U}$ – сильно* NP*-трудная целочисленная проблема. Если* $\mathrm{P} \neq \mathrm{NP}$*, то не существует псевдополиномиальных алгоритмов для решения проблемы $\mathcal{U}$.*

Доказательство. Поскольку проблема $\mathcal{U}$ является сильно NP-трудной, существует некоторая полиномиальная функция p, такая что проблема Value(p)-$\mathcal{U}$ является NP-трудной. Предположим, что существует псевдополиномиальный алгоритм решения проблемы $\mathcal{U}$. Тогда, согласно теореме 7.3, для каждой полиномиальной функции h мы получаем существование полиномиально-временно́го алгоритма решения проблемы Value(h)-$\mathcal{U}$. А последний факт влечёт существование полиномиально-временно́го алгоритма для NP-трудной проблемы Value(p)-$\mathcal{U}$ – и, следовательно, влечёт равенство $\mathrm{P} = \mathrm{NP}$, которое противоречит сделанному нами предположению $\mathrm{P} \neq \mathrm{NP}$. □

Мы можем снова применить метод сводимости – теперь для доказательства несуществования псевдополиномиального алгоритма, решающего некоторую заданную проблему. Для этого мы представим одно из приложений концепции сильной NP-трудности – покажем, что сильно NP-трудной является TSP. А чтобы доказать последнее, мы используем следующий факт: ответ на вопрос, содержит ли некоторый граф Гамильтонов цикл[7], является NP-полной проблемой.

Лемма 7.8. *Проблема* TSP *является сильно* NP*-трудной.*

Доказательство. Поскольку HC является NP-трудной проблемой, достаточно показать, что

$$\mathrm{HC} \leq_p Lang_{\mathrm{Value}(p)\text{-TSP}}$$

для полиномиальной функции $p(n) = n$.

Пусть $G = (V, E)$ – частный случай HC, причём $|V| = n$ для некоторого натурального n. Построим полный взвешенный граф (K_n, c), такой что $K_n = (V, E_{\mathrm{complete}})$, следующим образом. Положим

$$E_{\mathrm{complete}} = \{\{u, v\} \mid u, v \in V, u \neq v\},$$

а весовую функцию $c : E_{\mathrm{complete}} \to \{1, 2\}$ определим так:

$$c(e) = 1 \quad \text{при} \quad c \in E;$$
$$c(e) = 2 \quad \text{при} \quad c \notin E.$$

Граф G содержит Гамильтонов цикл тогда и только тогда, когда стоимость оптимального решения частного случая TSP (K_n, c) в точности равна n – т. е. когда

$$((K_n, c), n) \in Lang_{\mathrm{Value}(p)\text{-TSP}}.$$

Следовательно, каждый алгоритм, допускающий язык $Lang_{\mathrm{Value}(p)\text{-TSP}}$, может быть использован для решения HC. □

[7] Т. е. рассматривается проблема Гамильтонова цикла, HC. Напомним, что Гамильтонов цикл (тур) графа G – это цикл, содержащий каждую вершина графа G в точности один раз.

Упражнение 7.9. Рассмотрим следующее обобщение проблемы вершинного покрытия. Пусть $G = (V, E)$ – некоторый граф, а $w : V \to \mathbb{N}_0$ – отображение, ставящее в соответствие каждой вершине графа G некоторый вес. Для любого вершинного покрытия S определим его стоимость как сумму весов вершин множества S. Проблема минимального взвешенного вершинного покрытия заключается в построении самого дешёвого вершинного покрытия заданного графа. Докажите, что эта проблема – сильно NP-трудная.

7.3 Аппроксимационные алгоритмы

Введём концепцию аппроксимационных алгоритмов[8], которые также предназначены для решения трудных оптимизационных проблем. Основная идея этих алгоритмов заключается в переходе от экспоненциально-временно́й сложности к полиномиально-временно́й – путём некоторого ослабления требований, накладываемых на решение. А именно, вместо требования получить *оптимальное* решение нам будет достаточно получения *«почти оптимального»*, или «близкого к оптимальному». Что означает термин «почти оптимальное» – объясняется ниже.

Определение 7.10. *Пусть* $\mathcal{U} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, cost, goal)$ *– оптимизационная проблема.*
Будем говорить, что A – **непротиворечивый алгоритм для решения** $\mathcal{U}$, *если для каждого входа* $x \in L$ *выход* $A(x)$ *есть одно из возможных решений для* x *– т. е.* $A(x) \in \mathcal{M}(x)$.
Пусть A *– непротиворечивый алгоритм для решения проблемы* $\mathcal{U}$. *Для каждого* $x \in L$ *определим* **аппроксимационное отношение** $\mathbf{R_A(x)}$ **алгоритма** $\boldsymbol{A}$ **над входом** $\boldsymbol{x}$ *как*

$$\mathbf{R_A(x)} = \max\left\{\frac{cost(A(x))}{\mathrm{Opt}_{\mathcal{U}}(x)}, \frac{\mathrm{Opt}_{\mathcal{U}}(x)}{cost(A(x))}\right\},$$

где $\mathrm{Opt}_{\mathcal{U}}(x)$ *– стоимость оптимального решения частного случая* x *проблемы* $\mathcal{U}$.
Для любого положительного действительного числа $\delta > 1$ *будем говорить, что* A *–* **δ-аппроксимацонный алгоритм для $\mathcal{U}$**, *если для каждого* $x \in L$

$$\mathrm{R}_A(x) \leq \delta.$$

Проиллюстрируем концепцию аппроксимационных алгоритмов применительно к проблеме минимального вершинного покрытия. Идея заключается в том, чтобы эффективно найти паросочетание[9,10] данного графа G, а затем в качестве вершинного покрытия выбрать все вершины, инцидентные рёбрам этого паросочетания.

[8] Как уже было отмечено в предисловии, в русской литературе применяется также термин «приближённые алгоритмы». *(Прим. перев.)*

[9] Паросочетание графа $G = (V, E)$ – это подмножество множества рёбер $M \subseteq E$, такое что не существует вершины из множества V, инцидентной более чем одному ребру множества M. Паросочетание называется максимальным, если для каждого ребра $e \in E - M$ множество $M \cup \{e\}$ не является паросочетанием графа G.

[10] Эквивалентные определения: паросочетание – произвольное подмножество попарно несмежных ребер графа; оно называется максимальным, если не содержится в каком-либо паросочетании с бóльшим числом рёбер.

Алгоритм VCA

Вход: Граф $G = (V, E)$.
Этап 1. $C := \emptyset$;
{В процессе работы алгоритма множество C всегда является некоторым подмножеством V, а в конце вычисления – некоторым вершинным покрытием графа G.}
$A := \emptyset$;
{В процессе работы алгоритма множество A всегда является некоторым подмножеством E (точнее – паросочетанием графа G), а когда работа алгоритма заканчивается, A является одним из максимальных паросочетаний.}
$E' := E$;
{В процессе работы алгоритма множество $E' \subseteq E$ содержит те и только те рёбра, которые не являются инцидентными вершинам, входящим в текущее значением множества C. В конце вычисления $E' = \emptyset$.}
Этап 2.

```
while E' ≠ ∅ do
  begin выбрать произвольное ребро {u, v} из E';
    C := C ∪ {u, v};
    A := A ∪ {{u, v}};
    E' := E' − {все рёбра, инцидентные u либо v};
  end
```

Выход: C

Рассмотрим возможный вариант выполнения алгоритма VCA – для графа, приведённого на рис. 7.1 (a). Пусть $\{b, c\}$ – первое ребро, выбранное в процессе работы алгоритма. Тогда

$$C = \{b, c\},\ A = \{\{b, c\}\},\ E' = E - \{\{b, a\}, \{b, c\}, \{c, e\}, \{c, d\}\}$$

– как показано на рис. 7.1 (b). Если второе ребро, выбранное алгоритом VCA из множества E', есть $\{e, f\}$ (рис. 7.1 (c)), то

$$C = \{b, c, e, f\}, A = \{\{b, c\}, \{e, f\}\} \text{ и } E' = \{\{d, h\}, \{d, g\}, \{h, g\}\}.$$

А если последний выбор алгоритма VCA – ребро $\{d, g\}$, то

$$C = \{b, c, e, f, d, g\}, A = \{\{b, c\}, \{e, f\}, \{d, g\}\} \text{ и } E' = \emptyset.$$

Следовательно, C – вершинное покрытие стоимости 6. Заметим, что множество $\{b, e, d, g\}$ является оптимальным вершинным покрытием – т. е. оно не может быть улучшено в результате какого-либо иного варианта выбора рёбер алгоритмом VCA.

Упражнение 7.11. Для графа G, приведённого на рис. 7.1(a), найдите такой порядок выбора рёбер на втором этапе работы алгоритма VCA, что итоговое вершинное покрытие C содержит все его вершины.

В литературе на английском языке – «matching». На русском языке употребляются и другие названия, в том числе довольно часто – «независимое множество ребер».

Отметим, что принятая терминология является очень неудачной: так, среди максимальных паросочетаний определяются наибольшие – число ребер в каждом из них является наибольшим для всех паросочетаний графа. *(Прим. перев.)*

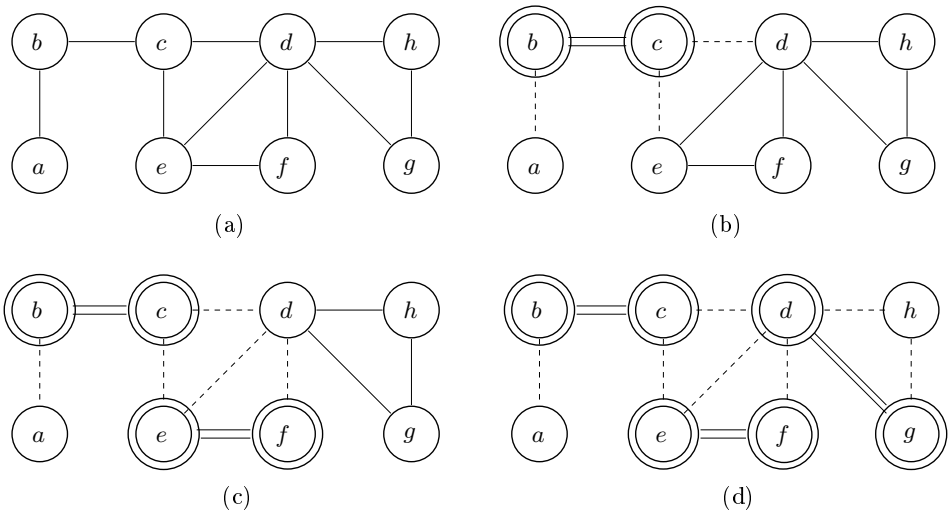


Рис. 7.1.

Теорема 7.12. *Алгоритм* VCA *является 2-аппроксимационным алгоритмом для* MIN-VCP*, при этом для любого частного случая* $G = (V, E)$

$$\mathrm{Time}_{\mathrm{VCA}}(G) \in O(|E|).$$

Доказательство. Выполнение условия

$$\mathrm{Time}_{\mathrm{VCA}}(G) \in O(|E|)$$

очевидно – поскольку каждое ребро множества E в процессе работы алгоритма VCA обрабатывается в точности один раз.

Поскольку в конце любого вычисления $E' = \emptyset$, алгоритм VCA действительно вычисляет некоторое вершинное покрытие графа G – т. е. является для проблемы MIN-VCP непротиворечивым алгоритмом.

Чтобы доказать, что для каждого графа G

$$\mathrm{R}_{\mathrm{VCA}}(G) \leq 2,$$

в первую очередь заметим, что:

- $|C| = 2 \cdot |A|$, и
- A – паросочетание графа G.

Для покрытия $|A|$ рёбер паросочетания A нам нужно выбрать по крайней мере $|A|$ вершин графа. Поскольку $A \subseteq E$, мощность любого вершинного покрытия графа G не меньше $|A|$, т. е.

$$\mathrm{Opt}_{\text{MIN-VCP}}(G) \geq |A|.$$

Следовательно,

$$\frac{|C|}{\mathrm{Opt}_{\text{MIN-VCP}}(G)} = \frac{2 \cdot |A|}{\mathrm{Opt}_{\text{MIN-VCP}}(G)} \leq 2.$$

□

Упражнение 7.13. Для любого $n \in \mathbb{N}$ опишите такой граф G_n, что его оптимальное вершинное покрытие имеет мощность n, – но при этом при работе алгоритма VCA может получиться вершинное покрытие мощностью $2n$.

Даёт ли гарантию (получения удовлетворительных результатов в результате работы соответствующих алгоритмов) аппроксимационное отношение, равное 2? Ответ на этот вопрос зависит от конкретного приложения, но обычно мы стараемся достигнуть ещё меньших аппроксимационных отношений – что, однако, требует намного более серьёзных алгоритмических идей. С другой стороны, мы оцениваем аппроксимационное отношение в худшем случае – поэтому применение 2-аппроксимацонного алгоритма к типичным частным случаям проблемы может дать решения с существенно лучшими, чем 2, аппроксимационными отношениями.

Однако существуют такие оптимизационные проблемы, которые являются трудными и для концепции аппроксимации. (Здесь слово «трудный» означает, что при сделанном предположении $\mathrm{P} \neq \mathrm{NP}$ ни для какого натурального d не существует полиномиально-временно́го d-аппроксимационного алгоритма, решающего данную проблему.) В разделе 7.2 мы показали, что TSP является слишком трудной для концепции псевдополиномиальных алгоритмов; теперь покажем, что TSP не может быть решена и с применением концепции аппроксимационных алгоритмов.

Лемма 7.14. *Если* $\mathrm{P} \neq \mathrm{NP}$*, то ни для какого натурального* d *не существует полиномиально-временно́го* d*-аппроксимацонного алгоритма для решения* TSP.

Доказательство. Докажем это методом от противного: предположим, что можно выбрать некоторое натуральное d, для которого существует полиномиально-временной d-аппроксимацонный алгоритм A, решающий TSP. Покажем при этом предположении, что существует полиномиально-временной алгоритм B для NP-полной проблемы HC – а это будет противоречить сделанному нами предположению $\mathrm{P} \neq \mathrm{NP}$.

Подобный алгоритм B, решающий HC, может быть описан для входа $G = (V, E)$ следующим образом.

1. B создаёт частный случай $(K_{|V|}, c)$ проблемы TSP, где:
$$K_{|V|} = (V, E'), \qquad E' = \{\{u, v\} \mid u, v \in V, u \neq v\},$$
$$c(e) = 1 \quad \text{при } e \in E,$$
$$c(e) = (d-1) \cdot |V| + 2 \quad \text{при } e \notin E.$$
2. B моделирует работу алгоритма A над входом $(K_{|V|}, c)$. Если решение, получаемое в результате работы алгоритма A, является Гамильтоновым циклом со стоимостью, в точности равной $|V|$, то B принимает свой вход G; иначе B отклоняет G.

Построение частного случая TSP $(K_{|V|}, c)$ может быть осуществлено за время $O(|V|^2)$. Второй этап алгоритма B выполняется за полиномиальное время – поскольку за полиномиальное время работает алгоритм A, а графы G и $K_{|V|}$ имеют один и тот же размер.

Остаётся показать, что B действительно решает проблему HC. Для этого в первую очередь заметим следующие факты.

(a) Если граф G содержит Гамильтонов цикл, то в графе $K_{|V|}$ имеется Гамильтонов цикл со стоимостю $|V|$, т. е.
$$\mathrm{Opt}_{\mathrm{TSP}}(K_{|V|}, c) = |V|.$$

(b) Каждый Гамильтонов цикл графа $K_{|V|}$, содержащий по крайней мере одно ребро множества $E' - E$, имеет стоимость не менее чем

$$|V| - 1 + (d-1) \cdot |V| + 2 = d \cdot |V| + 1 > d \cdot |V|.$$

Покажем, что $G = (V, E) \in$ HC тогда и только тогда, когда выходом алгоритма B является некоторое решение, имеющее стоимость $|V|$.

Пусть $G = (V, E) \in$ HC – т. е. граф G содержит некоторый Гамильтонов цикл (пусть цикл C). Согласно определению весовой функции c, стоимость этого цикла в $K_{|V|}$ равна $|V|$, поэтому

$$\mathrm{Opt}_{\mathrm{TSP}}(K_{|V|}, c) = |V|.$$

Если стоимость некоторого Гамильтонова цикл графа $K_{|V|}$ превышает значение $|V|$, то, согласно (b), эта стоимость не меньше, чем

$$d \cdot |V| + 1 > d \cdot |V|.$$

Следовательно, d-аппроксимационный алгоритм A должен вычислить возможное решение со стоимостью $|V|$ – т. е. B принимает G.

Обратно, пусть $G = (V, E)$ не входит в HC. В этом случае стоимость каждого возможного решения проблемы $(K_{|V|}, c)$ превышает значение $|V|$ – т. е. $cost(A(K_{|V|}, c)) > |V|$. Отсюда получаем, что B отклоняет G.

Итак, B – полиномиально-временной алгоритм, который решает NP-трудную проблему HC, – что противоречит нашему предположение P $\neq$ NP. □

В качестве возможного подхода к проблеме TSP мы скомбинируем концепцию аппроксимации с концепцией поиска множества простых частных случаев рассматриваемой проблемы. Для этого рассмотрим метрическую TSP, т. е. Δ-TSP, – которая допускает только частные случаи TSP, а именно те, которые удовлетворяют неравенству треугольника (пример 2.3). Это неравенство является естественным ограничением, которому удовлетворяют[11] многие конкретные приложения. Мы разработаем полиномиально-временной 2-аппроксимационный алгоритм решения Δ-TSP.

Алгоритм SB

Вход: Полный граф $G = (V, E)$ с весовой функцией $c : E \to \mathbb{N}^+$, которая для всех попарно различных вершин $u, v, w \in V$ удовлетворяет неравенству треугольника

$$c(\{u, v\}) \leq c(\{u, w\}) + c(\{w, v\}).$$

Этап 1. SB строит минимальное остовное дерево[12] T графа G относительно функции c.

Этап 2. SB выбирает произвольную вершину v множества V и выполняет поиск в глубину в дереве T – начиная с выбранной вершины v. При этом SB нумерует вершины дерева T в том порядке, в котором они встречаются в процессе работы алгоритма. Итоговую последовательность вершин, соответствующее этой нумерации, обозначим H.

Выход: Гамильтонов цикл $\overline{H} = H, v$.

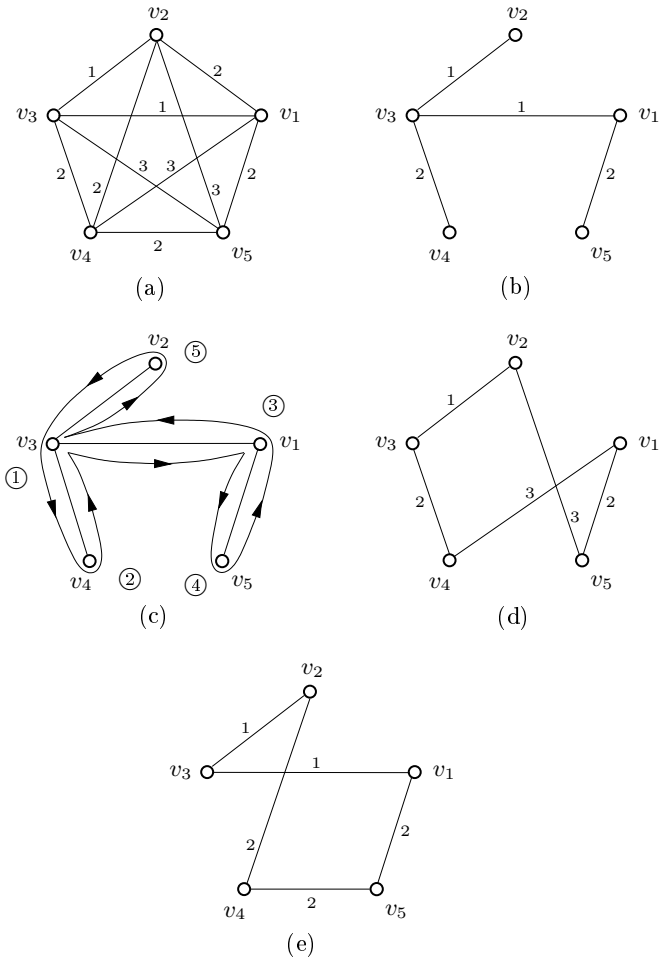


Рис. 7.2.

Проиллюстрируем работу алгоритма SB для приведённого на рис. 7.2 (a) частного случая B проблемы Δ-TSP. Минимальное остовное дерево

$$T = (\{v_1, v_2, v_3, v_4, v_5\}, \{\{v_1, v_3\}, \{v_1, v_5\}, \{v_2, v_3\}, \{v_3, v_4\}\}$$

графа G изображено на рис. 7.2(b). А рис. 7.2 (c) показывает процесс поиска в глубину для дерева T – начиная с вершины v_3. Заметим, что каждое ребро дерева T при поиске в глубину используется в точности 2 раза. Поиск в глубину определяет последовательность вершин

$$H = v_3, v_4, v_1, v_5, v_2$$

[11] Или – «почти удовлетворяют».

[12] Остовное дерево графа $G = (V, E)$ – это дерево $T = (V, E')$ с $E' \subseteq E$; здесь важно совпадение множества вершин у графов G и T. Если каждое ребро графа имеет некоторую стоимость, то стоимость графа равна сумме стоимостей всех его рёбер – поэтому стоимостью дерева T является сумма стоимостей всех рёбер множества E'.

– такую что последовательность

$$\overline{H} = v_3, v_4, v_1, v_5, v_2, v_3 = H, v_3$$

является выходом алгоритма SB (рис. 7.2 (d)). Стоимость последовательности $\overline{H}$ равна $2+3+2+3+1=11$. А оптимальным решением является последовательность

$$v_3, v_1, v_5, v_4, v_2, v_3,$$

имеющая стоимость $1+2+2+2+1=8$ (рис. 7.2 (e)).

Теорема 7.15. *Алгоритм* SB *является полиномиально-временны́м* 2*-аппроксимационным алгоритмом решения* Δ-TSP.

Доказательство. Сначала проанализируем временну́ю сложность алгоритма SB. Минимальное остовное дерево графа $G=(V,E)$ может быть найдено за время $O(|E|)$. Поиск в глубину в дереве $T=(V,E')$ выполняется за время $O(|V|)$. Следовательно,

$$\mathrm{Time}_{\mathrm{SB}}(G) \in O(|E|),$$

т. е. SB работает за линейное время.

Теперь покажем, что аппроксимационное отношение алгоритма SB не превосходит 2. Пусть H_{Opt} – оптимальный Гамильтонов цикл со стоимостью $cost(H_{\mathrm{Opt}}) = \mathrm{Opt}_{\Delta\text{-TSP}}(G)$ для $I=((V,E),c)$ – некоторого частного случая проблемы Δ-TSP. Пусть также $\overline{H}$ – выход $\mathrm{SB}(I)$ алгоритма SB для входа I, а $T=(V,E')$ – минимальное остовное дерево, построенное на первом этапе алгоритма SB. Отметим, что

$$cost(T) = \sum_{c\in E'} c(e) < cost(H_{\mathrm{Opt}}) \tag{7.1}$$

– поскольку после удаления ребра из графа H_{Opt} мы получаем остовное дерево графа G, а T – его оптимальное остовное дерево.

Рассмотрим W – путь, который соответствует поиску в глубину в дереве T. При этом W проходит в точности два раза через каждое ребро дерева T (по одному разу в каждом из двух направлений).[13] Если $cost(W)$ является суммой стоимостей всех рёбер W, то

$$cost(W) = 2 \cdot cost(T). \tag{7.2}$$

Из неравенства (7.1) и равенства (7.2) следует неравенство

$$cost(W) < 2 \cdot cost(H_{\mathrm{Opt}}). \tag{7.3}$$

Заметим, что путь $\overline{H}$ может быть получен из пути W путём замены некоторого его подпути $u, v_1, \ldots, v_k, v$, принадлежащего W, на ребро $\{u,v\}$ (т. е. путём прямого соединения вершин u и v).[14] Фактически это может быть сделано простой операцией,

[13] Заметим, что W может быть рассмотрен как Эйлеров цикл мультиграфа T_2. Последний мультиграф строится на основе дерева T удвоением каждого его ребра.

[14] Это происходит, когда вершины $v_1, \ldots, v_k$ уже были предварительно посещены нашим алгоритмом – перед вершиной u. Но при этом в пути W вершина v должна быть посещена в первый раз.

заменяющей подпоследовательность из трёх вершин u, w, v в W на подпоследовательность u, v (т. е. путём прямого соединения вершин u и v) – эта замена происходит, если вершина w уже встречалась в префиксе пути W. Такая операция не увеличивает стоимости пути – вследствие выполнения неравенства треугольника

$$c(\{u, v\}) \leq c(\{u, w\}) + c(\{w, v\}).$$

Следовательно,

$$cost(\overline{H}) \leq cost(W). \tag{7.4}$$

Неравенства (7.3) и (7.4) дают

$$cost(\overline{H}) \leq cost(W) < 2 \cdot cost(H_{\mathrm{Opt}}),$$

а отсюда

$$\frac{\mathrm{SB}(I)}{\mathrm{Opt}_{\Delta\text{-TSP}}(I)} = \frac{cost(\overline{H})}{cost(H_{\mathrm{Opt}})} < 2.$$

□

Упражнение 7.16. Для любого натурального $n \geq 3$ рассмотрим полный граф K_n, состоящий из n вершин. Найдите отличную от константы весовую функцию для графа K_n, такую что алгоритм SB всегда вычисляет оптимальное решение.

Упражнение 7.17.* Для каждого натурального $n \geq 4$ опишите частный случай I_n проблемы Δ-TSP, такой что

$$\frac{\mathrm{SB}(I_n)}{\mathrm{Opt}_{\Delta\text{-TSP}}(I_n)} \geq \frac{2n-2}{n+1}.$$

7.4 Алгоритмы локального поиска

Локальный поиск – это специальная технология разработки алгоритмов решения оптимизационных проблем. Её основными принципами являются:

- вычисление некоторого допустимого решения α для данного входа x;
- последовательное улучшение α путём его небольших (локальных) изменений.

Смысл термина «небольшие изменения» определяется с помощью следующего понятия окрестности.

Определение 7.18. *Пусть $\mathcal{U} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, cost, goal)$ – оптимизационная проблема. Для каждого $x \in L$ определим* **окрестность на множестве $\mathcal{M}(x)$** *как некоторое отображение $f_x : \mathcal{M}(x) \to \mathcal{P}(\mathcal{M}(x))$, удовлетворяющее следующим свойствам:*

(a) $\alpha \in f_x(\alpha)$ для каждого $\alpha \in \mathcal{M}(x)$.
{Любое α всегда находится в своей окрестности (в окрестности α).}
(b) Если $\beta \in f_x(\alpha)$ для некоторых $\alpha, \beta \in \mathcal{M}(x)$, то $\alpha \in f_x(\beta)$.
{Если β находится в окрестности α, то и α находится в окрестности β.}

(c) для всех $\alpha, \beta \in \mathcal{M}(x)$ *существуют натуральное* k *и* $\gamma_1, \gamma_2, \ldots, \gamma_k \in \mathcal{M}(x)$, *такие что*

$$\gamma_1 \in f_x(\alpha), \quad \gamma_{i+1} \in f_x(\gamma_i) \text{ для } i = 1, \ldots, k-1, \quad \beta \in f_x(\gamma_k).$$

{Для всех допустимых решений α *и* β *можно, начиная с* α, *достигнуть решения* β *– путём перемещения по другим допустимым решениям. При этом каждый шаг такого перемещения является переходом на какой-нибудь элемент множества* $\mathcal{M}(x)$, *входящий в окрестность предыдущего допустимого решения.}*

Если $\alpha \in f_x(\beta)$ *для некоторых* $\alpha, \beta \in \mathcal{M}(x)$, *то будем говорить, что* **α и β являются соседями на $\mathcal{M}(x)$**. *Множество* $\boldsymbol{f_x(\alpha)}$ *называется* **окрестностью α на $\mathcal{M}(x)$**. *Допустимое решение* $\alpha \in \mathcal{M}(x)$ *называется* **локальным оптимумом для x относительно окрестности f_x**, *если*

$$cost(\alpha) = goal\{cost(\beta) \mid \beta \in f_x(\alpha)\}.$$

Пусть для каждого $x \in L$ *имеется некоторая функция* f_x, *являющаяся окрестностью на* $\mathcal{M}(x)$. *Для всех* $x \in L$ *и всех* $\alpha \in \mathcal{M}(x)$ *определим функцию*

$$f : \bigcup_{x \in L} (\{x\} \times \mathcal{M}(x)) \to \bigcup_{x \in L} \mathcal{P}(\mathcal{M}(x))$$

с помощью равенства

$$f(x, \alpha) = f_x(\alpha).$$

Назовём эту функцию **окрестностью проблемы $\mathcal{U}$**.[15]

Согласно условию (b) определения 7.18, любая окрестность на множестве $\mathcal{M}(x)$ может быть рассмотрена в качестве соответствующего симметричного бинарного отношения на этом множестве. Но если мы хотим определить окрестность на $\mathcal{M}(x)$ для некоторого конкретного приложения, то нам неудобно работать с формализмами, использующими функции и бинарные отношения. Типичный путь введения на множестве $\mathcal{M}(x)$ окрестности – использование т. н. **локальных преобразований** на этом множестве. Термин «локальный» здесь является весьма важным. Смысл локального преобразования решения α заключается в том, что для получения какого-либо нового допустимого решения может быть изменена только некоторая, *локальная* часть описания решения α.

Приведём пример. Возможным локальным преобразованием для проблемы MAX-SAT[16] является переприсваивание Булевого значения ровно одной переменной. В этом случае окрестность допустимого решения α – это, во-первых, само α, а также все допустимые решения, которые могут быть получены путём применения этого локального преобразования. Например, для формулы из 5 переменных окрестностью допустимого решения $\alpha = 01100$ относительно локального преобразования, изменяющего (переприсвающего) Булево значение одной из переменных, является множество

$$\{01100, 11100, 00100, 01000, 01110, 01101\}.$$

[15] Мы применяем этот термин как для окрестности некоторого допустимого решения, так и для объединения всех таких множеств – т. е. окрестности всей проблемы. Мы надеемся, что такая несколько неудачная терминология (она принята и в английском языке) проблем всё же не вызовет – каждый раз из контекста будет понятно, какая именно окрестность имеется в виду. *(Прим. перев.)*

[16] А также многих других проблем, допустимые решения которых являются множествами значений нескольких Булевых переменных.

Упражнение 7.19. Докажите, что локальное преобразование замены бита для проблемы MAX-SAT удовлетворяет определению окрестности.

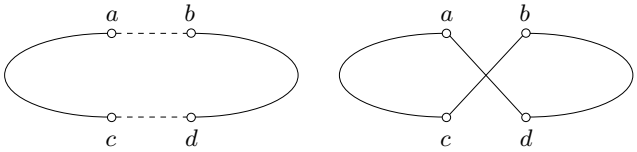


Рис. 7.3.

Наиболее известной окрестностью для TSP является т. н. окрестность **2-Exchange** (рис. 7.3). Простейший путь её определения – описание соответствующего локального преобразования; оно состоит из:

- удаления двух рёбер $\{a,b\}$ и $\{c,d\}$ с $|\{a,b,c,d\}| = 4$ [17] данного Гамильтонова тура α, который посещает эти 4 вершины в порядке a,b,c,d;
- добавление в α двух рёбер $\{a,d\}$ и $\{b,c\}$.

Заметим (см. рис. 7.3), что итоговый объект – снова Гамильтонов тур, и поэтому рёбра $\{a,b\}$ и $\{c,d\}$ могут быть заменены только на $\{a,d\}$ и $\{b,c\}$ – поскольку мы должны получить новый Гамильтонов тур.

Упражнение 7.20. Удовлетворяет ли окрестность 2-Exchange условиям определения 7.18? Приведите формальное обоснование ответа.

Упражнение 7.21. Пусть H – Гамильтонов тур графа G. Удалим три ребра – $\{a,b\}$, $\{c,d\}$ и $\{e,f\}$, такие что $|\{a,b,c,d,e,f\}| = 6$, а H посещает эти 6 вершины в порядке a,b,c,d,e,f. Укажите все возможные тройки рёбер, добавление которых к H снова даёт Гамильтонов тур. Сколько существует таких троек, если для некоторого $k \geq 3$ было удалено k рёбер, формирующих некоторое паросочетание?

Локальный поиск относительно окрестности является итерационным передвижением во множестве $\mathcal{M}(x)$ – от *некоторого* допустимого решения к *лучшему* допустимому решению. Эта итерационная процедура заканчивает работу в том случае, когда она достигает такого допустимого решения β, чьи соседи не лучше, чем β. Итак, мы можем представить схему локального поиска следующим образом.

Схема локального поиска, соответствующая окрестности Neigh
LS (Neigh)

Вход: Частный случай x оптимизационной проблемы $\mathcal{U}$.
Этап 1. Найти допустимое решение $\alpha \in \mathcal{M}(x)$.
Этап 2.

```
while α не является локальным оптимумом,
      соответствующим Neigh_x
  do  begin
      найти некоторое β ∈ Neigh_x(α), такое что:
```

[17] Это означает, что все 4 рассматриваемые вершины различны.

$cost(\beta) < cost(\alpha)$, если $\mathcal{U}$ – проблема минимизации, и
$cost(\beta) > cost(\alpha)$, если $\mathcal{U}$ – проблема максимизации;
end

Выход: α.

Заметим, что LS(Neigh) всегда вычисляет локальный оптимум относительно окрестности Neigh. Если все локальные оптимумы являются одновременно глобальными, то алгоритм локального поиска гарантирует получение решения оптимизационной проблемы. Например, таковой является проблема построения минимального остовного дерева – где окрестность определяется путём локального преобразования, заключающегося в замене ребра.

Если стоимости локальных оптимумов мало отличаются от стоимости оптимального решения, то локальный поиск можно использовать для разработки аппроксимационных алгоритмов. Примером является проблема максимального разреза (MAX-CUT), определённая в разделе 6.6. Для определения окрестности рассмотрим локальное преобразование, перемещающее некоторую вершину из одного подмножества в другое. Применительно к схеме локального поиска для такой окрестности мы получаем следующий алгоритм.

Алгоритм LS-CUT

Вход: Граф $G = (V, E)$.
Этап 1. $S = \emptyset$.

{В процессе вычисления мы рассматриваем разрез $(S, V - S)$. Поэтому в начале работы алгоритма разрез таков: $(\emptyset, V)$.}

Этап 2.

while существует вершина $v \in V$, такая что
$cost(S \cup \{v\}, V - (S \cup \{v\})) > cost(S, V - S)$ или
$cost(S - \{v\}, (V - S) \cup \{v\}) > cost(S, V - S)$
do begin
переместить v в другую часть разреза;
end

Выход: $(S, V - S)$.

Теорема 7.22. LS-CUT *является* 2*-аппроксимационным алгоритмом для проблемы* MAX-CUT.

Доказательство. Очевидно, что алгоритм LS-CUT вычисляет допустимое решение для проблемы MAX-CUT. Осталось показать, что для каждого графа $G = (V, E)$ выполнено неравенство

$$R_{\text{LS-CUT}}(G) \leq 2.$$

Пусть (Y_1, Y_2) – выход проблемы LS-CUT. Отметим, что (Y_1, Y_2) – локальный минимум относительно перемещения вершины из одной части разреза в другую.[18] Поэтому

[18] Ранее в этом разделе обычно применялся термин «минимум относительно некоторой окрестности» – а здесь применяется «минимум относительно некоторого локального алгоритма». Однако в данном случае (и аналогичных) на основе описанного локального алгоритма очевидным образом определяется функция f_x, использующаяся в определении 7.18, – т. е. мы действительно знаем окрестность. *(Прим. перев.)*

каждая вершина $v \in Y_1$ $[Y_2]$ имеет не меньше рёбер, идущих в вершины множества Y_2 $[Y_1]$, чем число рёбер, идущих из неё же в вершины множества Y_1 $[Y_2]$.[19] Этот простой «вычислительный» аргумент убеждает, что разрез (Y_1, Y_2) содержит по крайней мере половину рёбер графа G. Поскольку $\mathrm{Opt}_{\text{MIN-CUT}}(G)$ не может превышать $|E|$, мы получаем, что

$$\mathrm{R}_{\text{LS-CUT}}(G) = \frac{\mathrm{Opt}_{\text{MIN-CUT}}(G)}{cost((Y_1, Y_2))} \leq \frac{|E|}{|E|/2} = 2.$$

□

Упражнение 7.23. Докажите, что LS-CUT является полиномиально-временны́м алгоритмом.

Упражнение 7.24. Рассмотрим т. н. проблему максимального взвешенного разреза (MAX-WEIGHT-CUT), которая является следующим обобщением проблемы MAX-CUT. Её входом являются граф $G = (V, E)$ и весовая функция $c : E \to \mathbb{N}_0$, ставящая в соответствие каждому ребру e некоторый вес $c(e)$. Стоимостью разреза является сумма весов всех его рёбер. Цель заключается в том, чтобы найти разрез с максимальный стоимостью.

Очевидно, что MAX-WEIGHT-CUT является целочисленной проблемой. Разработайте псевдополиномиальный 2-аппроксимационный алгоритм локального поиска для проблемы MAX-WEIGHT-CUT.

Алгоритмы, основанные на локальном поиске, называются также локальными алгоритмами; они полностью (или почти полностью) определяются путём описания окрестности. А «свободными параметрами» схемы локального поска могут быть только такие:

- стратегия поиска более удачных соседних элементов;
- решение о том, принимать ли найденный более удачный соседний элемент в качестве следующего допустимого решения для итерационного поиска – или посчитать его одним из лучших решений в окрестности, но продолжить итеративный поиск ещё более удачного допустимого решения.

Если верно предположение P $\neq$ NP, то не существует полиномиально-временны́х локальных алгоритмов для NP-трудных оптимизационных проблем. Заметим, что временна́я сложность локального алгоритма может быть приблизительно оценена как

$$(\textit{время поиска в окрестности}) \times (\textit{число итеративных улучшений}).$$

Далее рассмотрим следующий вопрос:

для каких NP*-трудных оптимизационных проблем существует окрестность* Neigh *полиномиального размера – такая, что алгоритм* LS(Neigh) *всегда вычисляет оптимальное решение?*

То есть мы готовы согласиться с тем, что число итеративных улучшений в худшем случае экспоненциально, – если каждая из итераций выполняется за полиномиальное время, и при этом гарантирована сходимость к оптимальному решению. Смысл здесь заключается в том, что при увеличении размера окрестности мы, хоть и увеличиваем временну́ю сложность каждой итерации, но при этом уменьшаем вероятность

[19] В противном случае рассматриваемый алгоритм должен был переместить вершину v в другую часть разреза.

«застревания» в плохом локальном оптимуме. Вопрос состоит в том, существует ли окрестность приемлемого размера – такая, что каждый локальный оптимум является глобальным. Поставленный вопрос может быть оформлен следующим образом.

Определение 7.25. *Пусть* $\mathcal{U} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, cost, goal)$ *– оптимизационная проблема, а* Neigh *– окрестность для её решения. Назовём* Neigh **точной**, *если для каждого входа* $x \in L$ *каждый локальный оптимум для* x *относительно* Neigh_x *является оптимальным решением для рассматриваемого* x.
Окрестность Neigh *назовём* **достижимой за полиномиальное время**[20], *если существует полиномиально-временной алгоритм*[21], *который для каждых* $x \in L$ *и* $\alpha \in \mathcal{M}(x)$ *находит одно из лучших допустимых решений*[22] *в* $\mathrm{Neigh}_x(\alpha)$.

Итак, для оптимизационной проблемы $\mathcal{U} \in$ NPO наш вопрос может быть переформулирован на основе терминологии определения 7.25 следующим образом:

> *существует ли точная окрестность для проблемы* $\mathcal{U}$, *достижимая за полиномиальное время?*

Положительный ответ на этот вопрос означает, что трудность задачи – с точки зрения концепции локального поиска – состоит в количестве итеративных улучшений, необходимых для достижения оптимального решения. И во многих случаях из положительного ответа следует, что для $\mathcal{U}$ алгоритмы локального поиска могут быть применимы на практике. Например, если $\mathcal{U}$ является целочисленной оптимизационной проблемой, то существование точной окрестности Neigh, достижимой за полиномиальное время, обычно означает, что LS(Neigh) является псевдополиномиальным алгоритмом решения проблемы $\mathcal{U}$.[23] Известным положительным примером является симплекс-метод для линейного программирования. Он основан на существовании точной окрестности, достижимой за полиномиальное время, – но при этом нельзя исключать возможности экспоненциального числа итерационных улучшений.

А отрицательный ответ на поставленный выше вопрос означает, что никакая окрестность, достижимая за полиномиальное время, не может гарантировать успешного применения локальных алгоритмов в задаче поиска оптимального решения. Следовательно, в этом случае при разработке конкретного приложения можно только *попытаться* получить локальный оптимум за полиномиальное время.

Упражнение 7.26. Пусть k-Exchange – некоторая окрестность для проблемы TSP, в которой при изменении одного Гамильтонова тура на некоторый другой могут быть заменены k рёбер. Чему равна мощность окрестности $k\text{-Exchange}_G(H)$ Гамильтонова тура H – если граф G имеет n вершин?

Упражнение 7.27.* Докажите, что не существует натурального k, для которого окрестность k-Exchange являлась бы точной окрестностью для проблемы TSP.

---

[20] Если некоторая окрестность является достижимой за полиномиальное время, то это не обязательно означает, что данная окрестность имеет полиномиальных размер.

[21] Мы рассматриваем оптимизационные задачи только из класса NPO – следовательно любой алгоритм, работающий за полиномиальное время относительно $|\alpha|$, является также полиномиально-временны́м алгоритмом относительно $|x|$.

[22] Лучшее допустимое решение не обязательно единственно.

[23] Поскольку для любой целочисленной задачи каждый итеративный шаг улучшает стоимость решения не менее чем на 1, а цена любого решения обычно ограничена суммой всех входных значений.

Наша следующая цель – представить метод, который можно успешно применять для конкретных оптимизационных проблем – для доказательства несуществования у них точных окрестностей, достижимых за полиномиальное время. Мы увидим, что понятие NP-трудности применимо и для этой цели.

Определение 7.28. *Пусть* $\mathcal{U} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, cost, goal)$ *– некоторая целочисленная оптимизационная проблема. Будем говорить, что* $\mathcal{U}$ *является* **ограниченной по стоимости**, *если для каждого частного случая проблемы* x, *такого что* $\mathrm{Int}(x) = (i_1, i_2, \ldots, i_n)$, $i_j \in \mathbb{N}_0$ *для* $j = 1, 2, \ldots, n$,[24] *условие*

$$cost(\alpha) \in \left\{ 1, 2, \ldots, \sum_{j=1}^{n} i_j \right\}$$

выполнено для каждого допустимого решения $\alpha \in \mathcal{M}(x)$.

Отметим, что почти все известные целочисленные оптимизационные задачи являются ограниченными по стоимости – следовательно, это требование не ставит каких-либо серьёзных ограничений на применимость описываемого далее метода.

Теорема 7.29. *Пусть* $\mathcal{U} \in$ NPO *– ограниченная по стоимости целочисленная оптимизационная проблема. Если* P $\neq$ NP *и* $\mathcal{U}$ *– сильно* NP*-трудная, то у проблемы* $\mathcal{U}$ *не существует точной окрестности, достижимой за полиномиальное время.*

Доказательство. Прежде всего отметим, что, поскольку $\mathcal{U} \in$ NPO, начальное допустимое решение может быть вычислено за полиномиальное время. Далее будем рассматривать доказательство от противного.

Пусть $\mathcal{U}$ обладает точной окрестностью Neigh, достижимой за полиномиальное время. Тогда для любого входа x один шаг итерации алгоритма LS(Neigh$_x$) может быть выполнен за полиномиальное время относительно значения $p(|x|)$ некоторого полинома p. На каждом шаге итерации LS(Neigh$_x$) стоимость текущего допустимого решения улучшается по крайней мере на 1 – поскольку стоимости допустимых решений являются целыми числами. Более того, все возможные стоимости принадлежат целочисленному интервалу

$$\text{от } 0 \text{ до } \sum_{j \in \mathrm{Int}(x)} j \leq |x| \cdot \mathrm{MaxInt}(x)$$

– поскольку $\mathcal{U}$ ограничена по стоимости. Следовательно, количество итеративных улучшений ограничено значением

$$|x| \cdot \mathrm{MaxInt}(x).$$

Поэтому суммарная временная сложность алгоритма LS(Neigh) находится в пределах

$$O(p(|x|) \cdot |x| \cdot \mathrm{MaxInt}(x)).$$

Поскольку Neigh является точной окрестностью, алгоритм LS(Neigh) вычисляет оптимальное решение для входа x – и, следовательно, является псевдополиномиальным. А так как $\mathcal{U}$ является сильно NP-трудной проблемой, существование для неё псевдополиномиального алгоритма противоречит сделанному предположению P $\neq$ NP (теорема 7.7). □

[24] Обозначение Int(x) было введено в разделе 7.2.

В главе 6 мы уже фактически доказали, что проблемы TSP и MAX-CL являются сильно NP-трудными. Заметим, что обе эти проблемы являются ограниченными по стоимости целочисленными оптимизационными задачами. Следовательно, у них нет точных окрестностей, достижимых за полиномиальное время. Более того, для TSP можно доказать несуществование соответствующей точной окрестности размера $2^{\sqrt[3]{n}}$.

7.5 Алгоритм имитационной нормализации

В этом разделе мы познакомимся с алгоритмом имитационной нормализации (simulated annealing)[25] – как эвристикой для решения трудных оптимизационных задач. Термин «эвристика» в этой области комбинаторной оптимизации неоднозначен – он используется в нескольких различных значениях.

Во-первых, в общем смысле эвристические агоритмы являются непротиворечивыми алгоритмами для решения оптимизационных задач. Они основаны на какой-нибудь ясной, причём обычно достаточно простой идее поиска во множестве всех допустимых решений – которая, однако, не гарантирует, что какое-нибудь оптимальное решение найдётся. В этой ситуации мы обычно говорим об эвристиках локального поиска – или «жадных» эвристиках – даже если эти эвристики предусматривают аппроксимационные алгоритмы.

А в узком смысле, который, в основном, и рассматривается здесь, эвристика является методом получения непротиворечивого алгоритма, который в типичных частных случаях рассматриваемых оптимизационных проблем действительно даёт допустимые решения приемлемого качества за приемлемое время – например, за полиномиальное; но *доказать* этот факт обычно невозможно. Однако, несмотря на этот недостаток – отсутствие какого-либо общего утверждения относительно приемлемого поведения – эвристический подход стал очень популярным и широко используемым. Две основные причины этого таковы.

1. Эвристики обычно являются простыми – следовательно, каждую из них легко реализовать и проверить. Поэтому и затраты на разработку эвристических алгоритмов решения рассматриваемой проблемы обычно намного меньше, чем на разработку технически сложных алгоритмов.
2. Эвристики являются устойчивыми – это означает, что одна эвристика может успешно работать для целого класса похожих задач – даже если эти задачи имеют разную комбинаторную структуру. Следовательно, изменение условий задачи в течение процесса разработки алгоритма не послужит причиной для каких-либо серьёзных проблем: обычно достаточно изменить или исправить несколько параметров эвристики. А при разработке специальных проблемно-ориентированных оп-

[25] В русской литературе, прежде всего – в переводах с английского языка, обычно применяется другой (по-видимому, неудачный) вариант перевода этого термина – «эмуляция отжига». См. напр., книгу *[С. Рассел, П. Норвиг. «Искусственный интеллект. Современный подход», Вильямс, 2006]*.

Применяя наш термин – имитационная нормализация – мы «убиваем двух зайцев». Во-первых, он не менее правилен с точки зрения физики. А «вторым зайцем» является то, что этот термин, по-видимому, значительно более удачен как для математической, так и для программистской литературы – т. е. более понятен специалистам в этих областях. *(Прим. перев.)*

тимизационных алгоритмов изменение условий задачи часто сопровождаются изменением комбинаторной структуры – что требует возвращения назад, к самому началу разработки алгоритма.

Если для решения трудной задачи применять локальный поиск, но при этом не иметь возможности анализировать поведение итогового алгоритма, то такой вариант применения локального поиска можно рассматривать как эвристику. Подобные эвристики имеют особое свойство устойчивости – поскольку они могут быть применены практически в любой оптимизационной задаче, и у нас нет сомнений в их простоте. Главный же недостаток локального поиска заключается в том, что он заканчивает работу в некотором локальном оптимуме – не учитывая, однако, того, является этот оптимум «хорошим» или «плохим». Наша следующая цель состоит в том, чтобы улучшить метод локального поиска путём возможного отступления от локальных оптимумов – для поиска более удачного допустимого решения. Возможная аналогия здесь прослеживается в том, что мы имитируем физическую оптимизацию, основанную на законах термодинамики.

В физике конденсированного вещества нормализацией называется перевод вещества, находящегося в твёрдом состоянии с повышенной энергией, в более низкоэнергетическое состояние – за счёт нагрева выше температуры фазового перехода, выдержки при заданной температуре и достаточно медленного охлаждения. Этот процесс может рассматриваться как процесс оптимизации – в следующем смысле. В начале работы имеется твёрдое тело со множеством недостатков в её кристалической структуре. Целью является получение состояния твёрдого тела с более низкой энергией,[26] а в идеальном случае – с минимально возможной.

Физический процесс нормализации состоит из следующих двух шагов.

1. Температура увеличивается до максимального значения, при которой твёрдое тело плавится.[27] Благодаря этому – опять же в идеальном случае – все частицы сами размещаются случайным образом.
2. Температура тепловой ванны медленно уменьшается согласно заданной схеме охлаждения – вплоть до достижения низкоэнергетического состояния твёрдого тела (идеальной кристалической структуры).

Для нас важно, что описанный процесс оптимизации может быть успешно смоделирован с помощью алгоритма Метрополиса – который можно рассматривать как рандомизированный алгоритм локального поиска. Ниже для заданного состояния s твёрдого тела запись $\mathrm{E}(s)$ обозначает энергию этого состояния; константа Больцмана обозначается $\mathrm{c_B}$.

Алгоритм Метрополиса

[26] Точнее, это является не столько целью, сколько средством для достижения других целей, получения твёрдым телом определенных свойств – например, повышения пластичности, снижения хрупкости и др. *(Прим. перев.)*

[27] Точнее, до температуры плавления процесс обычно не доходит: применяется нагрев выше температуры фазовых превращений (температуры перехода в другое кристаллическое состояние) и выдержке при этой температуре. Всё это необходимо для того, чтобы «раскачать» структуры, избавиться от их дефектов «в прошлой жизни». *(См., напр., следующую публикацию: М.Криштал. Термодинамика релаксационных процессов... Известия РАН. Серия физическая. 2005, том 69, № 9. Прим. перев.)*

Вход: Состояние s твёрдого тела с энергией $\mathrm{E}(s)$.
Этап 1. Выбрать начальную температуру T тепловой ванны.
Этап 2. Сгенерировать состояние q из имеющегося состояния s. Это делается путём применения механизма возмущения, который осуществляет перевод состояния s в состояние q с помощью небольшого случайного искажения – например, с помощью случайного перемещения небольшой частицы.
`if` $\mathrm{E}(q) \le \mathrm{E}(s)$ `then` $s := q$ {принять q как новое состояние}
`else` принять q как новое состояние с вероятностью

$$\mathrm{prob}(s \to q) = \mathrm{e}^{-\frac{\mathrm{E}(q)-\mathrm{E}(s)}{c_\mathrm{B}\cdot T}}$$

{т. е. принять состояние s с вероятностью $1 - \mathrm{prob}(s \to q)$}
Этап 3. Соответствующим образом уменьшить T.
`if` T существенно отличается от 0 `then goto` этап 2;
`else output`(s);

Отметим большое сходство между алгоритмом Метрополиса и схемой локального поиска. Чтобы перейти от текущего состояния s в новое, мы рассматриваем только небольшое, локальное изменение в описании s – и таким образом генерируем состояние q. Если сгенерированное состояние q является по крайней мере столь же хорошим, как s (или даже лучше), то далее q рассматривается в качестве нового состояния.

А основные отличия алгоритма Метрополиса от схемы локального поиска состоят в следующем.

- Алгоритм Метрополиса может принять вариант, ухудшающий состояние, – с вероятностью
$$\mathrm{prob}(s \to q) = \mathrm{e}^{-\frac{\mathrm{E}(q)-\mathrm{E}(s)}{c_\mathrm{B}\cdot T}}.$$
- Окончание работы алгоритма Метрополиса определяется значением параметра T – в то время как критерием остановки схемы локального поиска является локальная оптимальность.

Вероятность $\mathrm{prob}(s \to q)$ удовлетворяет законам термодинамики – которые утверждают, что при температуре T вероятность $\mathrm{prob}(\Delta\mathrm{E})$ увеличения энергии не менее чем на $\Delta\mathrm{E}$ определяется формулой

$$\mathrm{prob}(\Delta\mathrm{E}) = \mathrm{e}^{\frac{-\Delta\mathrm{E}}{c_\mathrm{B}\cdot T}}.$$

Эта вероятность является необходимой для доказательства сходимости алгоритма Метрополиса к оптимальному состоянию. Для наших прикладных задач наиболее важным свойствами $\mathrm{prob}(s \to q)$ являются следующие.

- Вероятность $\mathrm{prob}(s \to q)$ перехода из состояния s в состояние q уменьшается с увеличением $\mathrm{E}(q) - \mathrm{E}(s)$ – т. е. бóльшие ухудшения менее вероятны, чем меньшие.
- Вероятность $\mathrm{prob}(s \to q)$ увеличивается вместе с T – т. е. большúе ухудшения более вероятны в начале работы алгоритма (когда значение T велико) чем впоследствии (когда T становится всё меньше и меньше).

Важно отметить, что для получения возможности оставить локальный оптимум мы иногда допускаем ухудшения: не допуская их, мы не можем гарантировать сходимости к оптимуму. Неформально – в начале работы алгоритма (т. е. когда T является большим) возможно преодолеть «высокие холмы» вокруг очень низкого локального

оптимума – чтобы найти несколько «низких долин», где только позднее будут найдены несколько «не слишком глубоких» локальных оптимумов. Интуитивно этот подход к оптимизации может рассматриваться как рекурсивная процедура – в следующем смысле. Сначала мы «взбираемся на вершину очень высокой горы» и ищем более многообещающие области («низкие долины»). Затем идем к одной из таких областей – и рекурсивно продолжаем искать минимум только в этой области.

Для применения такой стратегии алгоритма Метрополиса в комбинаторной оптимизации надо использовать следующее взаимно-однозначное соответствие между понятиями термодинамической оптимизации и комбинаторной.

$$\begin{aligned}
\text{множество состояний системы} &\mathrel{\widehat{=}} \text{множество допустимых решений}\\
\text{энергия состояния} &\mathrel{\widehat{=}} \text{стоимость допустимого решения}\\
\text{механизм возмущений} &\mathrel{\widehat{=}} \text{случайный выбор соседнего элемента}\\
\text{оптимальное состояние} &\mathrel{\widehat{=}} \text{оптимальное допустимое решение}\\
\text{температура} &\mathrel{\widehat{=}} \text{парметр управления}
\end{aligned}$$

Алгоритм имитационной нормализации решения оптимизационных проблем является алгоритмом локального поиска, основанным на аналогии с алгоритмом Метрополиса. Если для минимизационной задачи $\mathcal{U} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, cost, minimum)$ зафиксировать некоторую окрестность f, то алгоритм имитационной нормализации может быть описан следующим образом.

Имитационная нормализация для проблемы $\mathcal{U}$ относительно f
SA(f)

Вход: Частный случай проблемы $x \in L$.
Этап 1. Вычислить допустимое решение $\alpha \in \mathcal{M}(x)$.

Выбрать начальное значение T («начальную температуру»).

Выбрать функцию g («функцию уменьшения температуры») с двумя параметрами.
Этап 2.

```
I := 0;
while T > 0 (или T существенно отличается от 0) do
  begin
    случайно выбрать некоторое β из f_x(α);
    if cost(β) ≤ cost(α) then α := β
    else  begin
      сгенерировать случайное r в интервале [0, 1];
      if r < e^(-(cost(β)-cost(α))/T) then α := β;
    end;
  I := I + 1;
  T := g(T, I);
end
```

Выход: α.

Выбрав приемлемые окрестность, «температуру» T и функцию её уменьшения g, можно доказать, что алгоритм SA(f) сходится к оптимуму. Но при этом имеется следующая проблема: количество итераций, необходимых для достижения оптимума, не

может быть заранее ограничено каким-либо значением. Все попытки получить гарантию хорошего коэффициента аппроксимации с помощью алгоритма $\mathrm{SA}(f)$ не приводят к успеху: возможны ситуации, в которых для получения приемлемого коэффициента аппроксимации требуется больше итераций алгоритма $\mathrm{SA}(f)$, чем $|\mathcal{M}(x)|$. Однако существует немало конкретных прикладных задач, для которых алгоритм имитационной нормализации даёт допустимые решения – и при этом он, благодаря небольшим вычислительным затратам, предпочтительнее других методов. Другое положительное свойство имитационной нормализации состоит в том, что выбор параметров T и g, а также критерия остановки программы, может быть передан пользователю. Следовательно, пользователь может сам расставлять приоритеты, связанные с выбором компромисса между временем выполнения алгоритма и качеством решения.

7.6 Заключение

Технологии разработки алгоритмов очень важны для успеха в решении трудных оптимизационных проблем – поскольку требования к вычислительным ресурсам, выдвигаемые количественным «прыжком»,[28] не могут быть удовлетворены путём какого-либо улучшения аппаратного обеспечения («железа»). Поэтому для получения эффективных алгоритмов решения некоторой трудной проблемы мы должны «заплатить» на уровне требований, относящихся к спецификации этой проблемы. При этом мы либо сокращаем множество возможных входов до некоторого подкласса типичных входов[29] – либо решаем нашу проблему без гарантии получения корректного или оптимального решения. Целью разработок эвристических алгоритмов является получение большой выгоды на уровене эффективности – путём «оплаты небольшой скидки» на уровне постановки этой проблемы (уровне требований).

Псевдополиномиальные алгоритмы выполняются за полиномиальное время в частных случаях целочисленных проблем – например, мы можем разработать псевдополиномиальный алгоритм для проблемы рюкзака. Для доказательства несуществования псевдополиномиальных алгоритмов решения конкретных проблем здесь полезна концепция NP-полноты – при дополнительном предположении $\mathrm{P} \neq \mathrm{NP}$.

А аппроксимационные алгоритмы дают допустимые решения, стоимость которых мало отличаются от стоимости оптимального решения. Мы можем разработать аппроксимационные алгоритмы для различных оптимизационных проблем – например, для метрической TSP, MAX-VC, и MAX-CUT. Предположив $\mathrm{P} \neq \mathrm{NP}$, мы можем доказать несуществование аппроксимацонных алгоритмов, решающих общую TSP; этот отрицательный результат может быть получен и путём применения концепции NP-трудности.

Схема локального поиска некоторой оптимизационной проблемы начинает работу с некоторого произвольного допустимого решения – после чего пытается получить более хорошее решение с помощью последовательного улучшением текущего. Одна итерация этой схемы состоит в поиске соседнего элемента текущего решения α – который лучше, чем само α. Соседние элементы допустимого решения α определяются путём применения к нему некоторого локального преобразования – которое может изменять

[28] Например, при переходе от экспоненциальной сложности к полиномиальной.

[29] Типичных относительно рассматриваемого приложения. Таким образом, мы *не* решаем проблему в её общей формальной постановке.

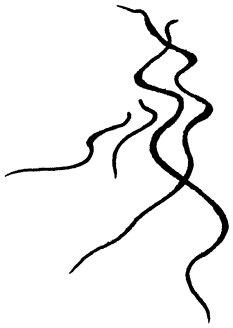
только локальную часть спецификации решения α. Локальные алгоритмы всегда заканчиваются в некотором локальном оптимуме относительно окрестности, допускающей локальные преобразования. Поскольку стоимости локальных оптимумов могут существенно отличаться от стоимости оптимального решения, локальный поиск для многих проблем не даёт гарантии качества решений.

Имитационная нормализация является эвристикой, основанной на локальном поиске, – но позволяющей и отход от локальных оптимумов. Она обладает свойством устойчивости (т. е. может быть применена в большинстве задач оптимизации и практически не меняется при изменении входных спецификаций задачи) – и в то же время легка для реализации; поэтому она широко используется во многих приложениях.

Концепции применения псевдополиномиальных алгоритмов в целочисленных проблемах, а также строгой NP-трудности, были введены Гэри и Джонсоном в [19]. Псевдополиномиальный алгоритм решения проблемы рюкзака был разработан Ибаррой и Кимом – см. [34]. Первый аппроксимационный алгоритм был предложен Грэхемом в [22]. Бок в [6] и Коэс в [15] представили первые алгоритмы локального поиска. Концепция точной окрестности, достижимой за полиномиальное время, была введена Пападимитриу и Стейглицем в [50]. Алгоритм Метрополиса для моделирования процесса нормализации был предложен самим Метрополисом – в соавторстве с А.В. и М.Н. Розенблютами, а также А.М. и Е. Теллерами в [46]. Идея применения алгоритма Метрополиса в комбинаторной оптимизации была независимо представлена Черны в [8] и Киркпатриком, Геллатом и Векки в [37].

Систематический обзор методов решения трудных проблем приведён в [30]. Для дальнейшего чтения по различным технологиям разработки алгоритмов мы настоятельно рекомендуем книги Пападимитриу и Стейглица [50], Кормена, Лейзерсона и Ривеста [14], а также Шёнинга [63]. Много информации, связанной с теорией аппроксимационных алгоритмов, имеется в книгах Аузелло, Крешенси, Гамбоси, Канна, Маркетти-Спаккамелы и Протаси [1], Хохбаума [26], Майра, Прёмеля и Штегера [45], а также Вазирани [69].

Вся ткань этого мира
построена из необходимости и случайности;
человеческий интеллект находится между ними –
и может ими управлять;
он рассматривает необходимость
в качестве причины существования мира;
он знает, как можно управлять случайностью,
контролировать и использовать её...
И. В. Гёте



8

Рандомизация

8.1 Цели и задачи главы

Понятие «случайность» является одним из основных и наиболее обсуждаемых терминов в науке – причём далеко не только в математике и информатике. Основной вопрос здесь такой: действительно ли существует случайность – или мы просто используем этот термин для моделирования объектов и событий с неизвестными нам законами? Учёные и философы обсуждали эти вопросы с античных времён. По мнению Демокрита,

случайность есть неизвестное,
и поэтому природа в своей основе детерминирована.

Итак, Демокрит утверждал, что весь мир подчиняется некоторому порядку, и что этот порядок управляется однозначными законами. А Эпикур, в противоположность Демокриту, заявлял, что:

случайность – объективная реальность,
она является свойством природы событий.

До 20-го века мирское представление людей о понятии «случайность» было «детерминированным» – т. е. основанным на причинно-следственной связи. Эта вера в детерминизм имела и эмоциональные корни – потому что случайность, относимая людьми к хаосу и неуверенности, всегда связывались с обычным страхом, в том числе религиозным, – и, таким образом, возможность случайных событий часто просто не допускалась. Всё это даже приносило некоторые успехи в развитии естественных и инженерно-технических наук.

Итак, случайность чаще всего сводилась к детерминизму. Даже Альберт Эйнштейн «возлагал ответственность» за случайность на наши недостаточные знания, утверждая, что каждая рандомизированная модель физической действительности может быть заменена на детерминированную – когда соответствующее знание будет обнаружено. Развитие науки (особенно физики и биологии) в 20-м веке возвратило мир к взгляду Эпикура относительно случайности (хаотичности). Экспериментальная физика подтвердила теорию квантовой механики – в основе которой лежат именно случайные события. А эволюционная биология сегодня рассматривает случайную мутацию ДНК как главный инструмент эволюции.

Хорошее, современное мнение относительно случайности было высказано венгерским математиком Альфредом Рени:

> *Хаотичность и порядок не противоречат друг другу; оба этих понятия могут быть более-менее верными одновременно. Случайность управляет миром – и поэтому в мире есть порядок, и законы, которые могут быть выражены с точки зрения случайных событий, следуют из законов теории вероятностей.*

И действительно, всё изложенное здесь связано с математикой и информатикой – прежде всего с теми их областями, которые связаны с теорией вероятностей. Например, при изучении многих областей теоретической информатики важно понимать, что часто может быть очень выгодно разрабатывать и реализовывать *рандомизированные алгоритмы*[1] и программные системы – вместо полностью детерминированных. Такую реализацию можно назвать принятием в качестве нашего учителя Природы.[2] По-видимому, она всегда использует самый эффективный и простой способ при достижении своей цели – а рандомизация в качестве части управления жизнью является существенным понятием её стратегии.

Практическое применение теоретической информатики подтверждает эту точку зрения. Во многих используемых ежедневно программных приложениях простые рандомизированные алгоритмы и системы, написанные на их основе, выполняют требуемую работу эффективно и с высокой степенью надежности – но мы не знаем детерминированных алгоритмов, способных сделать то же самое с сопоставимой эффективностью. Известны многие примеры, в которых использование детерминированных копий некоторых удачных рандомизированных алгоритмов находится за границами физических возможностей компьютера. Это является одной из причин того, что в настоящее время мы соединяем удобство манипулирования – как одно из свойств математической модели в классе фактически разрешимых проблем – не с детерминированным полиномиальным временем, а с рандомизированным.

В этой главе мы не стремимся представить основные принципы проектирования и анализа рандомизированных алгоритмов, а также теорию сложности рандомизированных вычислений – поскольку это потребовало бы слишком больших знаний в теории вероятностей, теории сложности, алгебре и теории чисел. Вместо этого мы предпочли привести три простых примера рандомизированных алгоритмов – которые просто и ясно объясняют понятие рандомизации, и, более того, помогают читателю понять, почему рандомизированный алгоритм может быть более мощным средством, чем любой детерминированный.

Глава организована следующим образом. Некоторые основные понятия теории вероятностей рассматриваются в разделе 8.2. В разделе 8.3 мы разработаем рандоми-

[1] Как уже было отмечено в предисловии, в русской литературе применяется также термин «вероятностные алгоритмы». *(Прим. перев.)*

[2] Приведём по этому поводу мысль из книги «Нечёткое мышление» математика и философа – также венгерского – Барта Коско:

> *Два тысячелетия назад человечество сделало роковую ошибку, заложив в фундамент науки не зыбкую поэтику ранних восточных философий, а выхолощенную двоичную логику Аристотеля. И с тех пор классическая «чёрно-белая» бинарная логика всё более отдаляется от реального многоцветного мира, где нет ничего абсолютного, а всё самое интересное происходит в туманной области между «да» и «нет».*

(Прим. перев.)

зированный коммуникационный протокол для сравнения содержимого двух больших баз данных; этот протокол существенно более эффективен любого детерминированного протокола, предназначенного для тех же целей. В разделе 8.4 подобный протокол используется для введения метода избытка свидетельств (abundance of witnesses) – как основы проекта рандомизированных алгоритмов. Мы применим этот метод ещё раз – с целью получения рандомизированного алгоритма проверки некоторого заданного числа на простоту; отметим, что данная проблема является одной из самых важных текущих практических задач, и что мы не знаем эффективных детерминированных алгоритмов её решения.[3] В разделе 8.5 представлен «дактилоскопический» метод (fingerprinting)[4] – как специальный случай метода избытка свидетельств; мы применяем его для эффективного решения вопроса об эквивалентности двух полиномов.

8.2 Основы теории вероятностей

Если некоторое событие является неизбежным следствием другого, то мы говорим о причинно-следственной связи – или детерминизме. Но, как уже было упомянуто во введении, могут существовать события, которые не являются детерминированными (не полностью определены). Теория вероятностей была создана именно для моделирования и анализа таких неопределённых ситуаций и экспериментирования с подобными неоднозначными результатами. Простыми примерами таких экспериментов является бросание монеты (игра в орлянку) или игрального кубика (игра в кости). Если нет очевидной возможности предсказания результата таких экспериментов, то мы говорим о **случайных событиях**.[5]

При моделировании вероятностного эксперимента мы рассматриваем все возможные его результаты, называемые **элементарными событиями**. С философской точки зрения важно, что эти элементарные события являются «атомарными», неделимыми; это означает, что никакое элементарное событие не может быть рассмотрено в качестве совокупности некоторых других[6] событий, результатов экспериментов – таким образом, одно элементарное событие исключает любое другое. Для бросания монеты элементарными событиями являются «орёл» и «решка». а для игрального кубика – «1», «2», «3», «4», «5» и «6». **Событие** – это некоторое множество элементарных событий (т. е. некоторое подмножество множества всех элементарных событий). Например, $\{2, 4, 6\}$ является событием для бросания игрального кубика, которое соответствует выпаданию некоторого чётного числа. Поскольку элементарные события также могут рассматриваться в качестве событий, мы в этом случае представляем их в качестве одноэлементных множеств.

---

[3] Заметим, что недавно ([44]) был описан первый детерминированный полиномиально-временной алгоритм проверки числа на простоту. Этот результат имеет большое теоретическое значение, однако его временна́я сложность есть $O(n^{12})$ – что не позволяет применять его на практике.

[4] Русское название термина не устоялось – но, несмотря на это, далее в данной главе будем писать название этого метода без кавычек. *(Прим. перев.)*

[5] В уже упомянутой книге *[С.Рассел, П.Норвиг. «Искусственный интеллект. Современный подход», Вильямс, 2006]* (стр. 86) приведено подробное обсуждение терминологии, применяемой для различных вариантов случайных событий – в различных «недетерминированных» ситуациях. *(Прим. перев.)*

[6] Даже «более элементарных».

Далее мы будем рассматривать эксперименты только с конечным числом элементарных событий – этого для наших целей достаточно, а весь следующий материал данного раздела становится при этом более ясным. Нашей целью является создание теории, которая естественным образом ставит в соответствие любому событию его вероятность. Достижение такой цели было далеко не простым: теории вероятностей потребовалось почти 300 лет для того, чтобы продвинуться от работ Паскаля, Ферма и Гюйгенса, выполненных в середине 17-го века, до общепринятой в настоящее время аксиоматики Колмогорова. Накладывая ограничение, чтобы множество элементарных событий S было конечным, мы избавляемся от технических сложностей, связанных с возможной несчётностью множества S в общем определении Колмогорова. Основная идея заключается в том, чтобы определить вероятность некоторого события E как

> *отношение между суммой вероятностей (благоприятных) элементарных событий, входящих в E, к сумме вероятностей всех возможных элементарных событий.* (8.1)

Зафиксировав таким образом вероятности событий, мы в общем случае накладываем на значения вероятностей некоторые стандарты – например, вероятность 1 соответствует достоверному событию, а вероятность 0 – невозможному.[7]

Ещё один важный момент заключается в том, что вероятности элементарных событий однозначно определяют вероятности всех событий. Например, для т. н. симметричных экспериментов – вроде бросания монеты или игрального кубика – для всех элементарных событий назначается одна и та же вероятность.

Пусть $\mathrm{Prob}(E)$ – вероятность события E. В нашей модели результат эксперимента должен быть одним из элементарных событий, поскольку мы полагаем $\mathrm{Prob}(S) = 1$ для множества S всех элементарных событий. Поэтому для бросания игрального кубика мы получаем, что

$$\begin{aligned}\mathrm{Prob}(\{2,4,6\}) &= \frac{\mathrm{Prob}(\{2\}) + \mathrm{Prob}(\{4\}) + \mathrm{Prob}(\{6\})}{\mathrm{Prob}(S)} \\ &= \mathrm{Prob}(\{2\}) + \mathrm{Prob}(\{4\}) + \mathrm{Prob}(\{6\}) \\ &= \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2},\end{aligned}$$

т. е. вероятность выпадения чётного числа в точности равна $1/2$. Следуя концепции измерения вероятности (8.1), мы получаем, что для всех пар несовместных (взаимоисключающих) событий X и Y выполнено равенство

$$\mathrm{Prob}(X \cup Y) = \frac{\mathrm{Prob}(X) + \mathrm{Prob}(Y)}{\mathrm{Prob}(S)} = \mathrm{Prob}(X) + \mathrm{Prob}(Y).$$

Эти рассуждения приводят к следующему определению вероятностей.

Определение 8.1. *Пусть S – множество всех элементарных событий вероятностного эксперимента.* **Распределение вероятностей на S** *– функция*

$$\mathrm{Prob} : \mathcal{P}(S) \to [0,1],$$

удовлетворяющая сдедующим условиям (аксиомам вероятности):

[7] Достоверное событие состоит из всех элементарных событий. Невозможное событие называется также пустым.

- $\mathrm{Prob}(\{x\}) \geq 0$ *для каждого элементарного события* x*;*
- $\mathrm{Prob}(S) = 1$*;*
- $\mathrm{Prob}(X \cup Y) = \mathrm{Prob}(X) + \mathrm{Prob}(Y)$ *для всех пар событий* $X, Y \subseteq S$*, для которых* $X \cap Y = \emptyset$.

$\mathrm{Prob}(X)$ *называется* **вероятностью события** $\boldsymbol{X}$. *Пара* (S, Prob) *называется* **вероятностным пространством**. *Если для всех* $x, y \in S$

$$\mathrm{Prob}(\{x\}) = \mathrm{Prob}(\{y\}),$$

то Prob *называется* **равномерным распределением вероятностей на** $\boldsymbol{S}$.

Упражнение 8.2. Докажите, что следующие свойства выполняются для каждого вероятностного пространства (S, Prob):

- $\mathrm{Prob}(\emptyset) = 0$;
- $\mathrm{Prob}(S - X) = 1 - \mathrm{Prob}(X)$ для каждого подмножества $X \subseteq S$;
- $\mathrm{Prob}(X) \leq \mathrm{Prob}(Y)$, для всех $X, Y \subseteq S$, таких что $X \subseteq Y$;
- $\mathrm{Prob}(X \cup Y) = \mathrm{Prob}(X) + \mathrm{Prob}(Y) - \mathrm{Prob}(X \cap Y)$
  $\leq \mathrm{Prob}(X) + \mathrm{Prob}(Y)$ для всех $X, Y \subseteq S$;
- $\mathrm{Prob}(X) = \sum_{x \in X} \mathrm{Prob}(x)$ для всех $X \subseteq S$.

Заметим, что все свойства определения 8.2 соответствуют нашим интуитивным понятиям о вероятности – и, следовательно, неформальной концепции (8.1). Итак, подобное введение вероятностей даёт нам тот факт, что вероятность нескольких попарно-независимых (взаимоисключающих) событий является суммой вероятностей этих событий.

А чему соответствует произведение двух вероятностей? Рассмотрим два вероятностных эксперимента, которые являются независимыми – в том смысле, что результат одного из них не зависит от результата другого. Примером такой ситуации является двойное бросание кубика. При этом не имеет значения, бросаем ли мы два кубика один раз или дважды используем один и тот же – поскольку в обоих случаях результаты независимы. Например, элементарное событие «3» первого кубика не связано с каким-либо результатом выпадения второго. Мы знаем, что $\mathrm{Prob}(i) = \frac{1}{6}$ для обоих экспериментов и для всех $i \in \{1, 2, \ldots, 6\}$.

Теперь рассмотрим объединение обоих вероятностных экспериментов в один. Множеством элементарных событий этого объединённого эксперимента является

$$S_2 = \{(i, j) \mid i, j \in \{1, 2, \ldots, 6\}\}$$

– где для некоторого элементарного события (i, j) этого множества число i является результатом первого кубика, а j – второго. Что является распределением вероятностей Prob_2 на множестве S_2 – которое может быть определено на основе эксперимента $(\{1, 2, \ldots, 6\}, \mathrm{Prob})$? Рассмотрим гипотезу о том, что вероятность некоторого события, состоящего из двух полностью независимых событий, равна произведению их вероятностей; согласно этой гипотезе для всех $i, j \in \{1, 2, \ldots, 6\}$ должно выполняться равенство

$$\mathrm{Prob}_2(\{(i, j)\}) = \mathrm{Prob}(\{i\}) \cdot \mathrm{Prob}(\{j\}) = \frac{1}{6} \cdot \frac{1}{6} = \frac{1}{36}.$$

Проверим корректность этой гипотезы. Множество S_2 содержит в точности 36 элементарных событий – и каждое из них равновероятно. Следовательно, для всех $(i, j) \in S_2$

$$\mathrm{Prob}_2(\{(i,j)\}) = \frac{1}{36}.$$

Упражнение 8.3. Пусть $k \in \mathbb{N}$, $S = \{0, 1, 2, \ldots, 2^k - 1\}$. Опишите вероятностное пространство (S, Prob) из k экспериментов, заключающихся в бросании монеты.

Нам осталось объяснить, как теория вероятностей может быть использована для моделирования, разработки и анализа рандомизированных алгоритмов. Одна из возможностей – начать с моделирования недетерминированной машины Тьюринга с конечными вычислениями, а потом заменить каждый её недетерминированный переход (недетерминированный выбор) на некоторый случайный эксперимент. Эта замена означает, что при наличии недетерминированного выбора из k возможностей мы рассматриваем их как k элементарных событий, причём каждое – с вероятностью $1/k$. В таком случае вероятность некоторого конкретного вычисления является произведением вероятностей всех случайных решений этого вычисления. Пусть $S_{A,x}$ – множество всех вычислений некоторой НМТ (недетерминированной программы) A со входом x. Тогда мы каждому вычислению C из множества $S_{A,x}$ вышеописанным путём ставим в соответствие вероятность $\mathrm{Prob}(C)$ – получая при этом требуемое вероятностное пространство $(S_{A,x}, \mathrm{Prob})$.

Упражнение 8.4. Докажите, что $(S_{A,x}, \mathrm{Prob})$ является вероятностным пространством.

Для некоторого заданного входа x сумма вероятностей вычислений из множества $S_{A,x}$, дающих неверный выход, называется **вероятностью ошибки алгоритма A над входом x** и обозначается $\mathbf{Error}_A(x)$. **Вероятность ошибки алгоритма A** определяется как функция вида $\mathrm{Error}_A : \mathbb{N}_0 \to \mathbb{N}_0$, такая что

$$\mathbf{Error}_A(n) = \max\{\mathrm{Error}_A(x) \mid |x| = n\}.$$

Моделирование рандомизированных алгоритмов с помощью вероятностных экспериментов может быть также использовано для анализа вероятности того, что для заданной функции t вычисление алгоритма A на входе x заканчивается не более чем за $t(n)$ шагов[8] – а отсюда можно получить численное значение эффективности рандомизированого алгоритма.

Другая простая возможность моделирования рандомизированных алгоритмов – это рассмотрение каждого из них как вероятностного распределения над множеством детерминированных алгоритмов. Для этого машине Тьюринга A (т. е. детерминированному алгоритму) в качестве дополнительных входных данных добавляется ещё одна лента, содержащая последовательность случайных битов. При этом каждая такая последовательность однозначно определяет (детерминированное) вычисление алгоритма A над заданным входом x. Такие случайные последовательности битов могут рассматриваться как элементарные события, соответствующие множеству $S_{A,x}$ всех вычислений машины A над входом x. И, поскольку обычно все случайные последовательности имеют одну и ту же вероятность, рандомизированный алгоритм является равномерным распределением вероятностей над всеми вычислениями множества $S_{A,x}$. Примеры

[8] Эта вероятность является суммой вероятностей всех вычислений, длина которых не превышает $t(|x|) + 1$.

рандомизированных алгоритмов, приведённые в следующем разделе, основаны на простом моделировании подобного случайного управления. В этих примерах последовательности случайных битов рассматриваются как случайные простые числа – которые и определяют выбор детерминированной стратегии для решения данной проблемы.

8.3 Рандомизированный протокол связи

Главная цель этого раздела – показать, что рандомизированные алгоритмы могут быть намного эффективнее своих самых лучших детерминированных аналогов. Рассмотрим следующую задачу. Пусть есть два компьютера, C_{I} и C_{II}, находящиеся очень далеко друг от друга – например, один в Европе, а другой в Америке. Сначала оба компьютера содержали одну и ту же базу данных; потом на каждом из них проводилась независимая работа с целью получения полной информации о предмете базы данных – например, о последовательности аминокислот некоторого генома. По прошествии некоторого времени мы захотели определить, успешно ли закончился этот процесс – т. е. действительно ли C_{I} и C_{II} содержат одни и те же данные.

Пусть n – размер базы данных в битах. Например, n может иметь порядок 10^{16} – это является реальным значением для биологических приложений. Наша цель состоит в том, чтобы разработать коммуникационный протокол между C_{I} и C_{II}, который дал бы возможность определить, действительно ли совпадают данные на этих компьютерах. Сложность коммуникационного протокола – это число битов, которые нужно переслать между C_{I} и C_{II} с целью получения ответа на данный вопрос. Очевидно, что мы стараемся минимизировать эту сложность.

Можно доказать, что любой детерминированный коммуникационный протокол для решения этой задачи должен переслать между C_{I} и C_{II} не менее n битов[9] – т. е. не существует детерминированных протоколов, которые решали бы эту проблему путём пересылки $n-1$ или меньшего числа битов. Надо переслать 10^{16} битов, дополнительно убедиться, что все они успешно дошли ... [10] Итак, вряд ли стоит идти по этому пути.

Более рациональное решение этой проблемы может быть получено путём применения следующего рандомизированного протокола, основанного на теореме о простых числах (теореме 2.59).

$R = (C_{\mathrm{I}}, C_{\mathrm{II}})$ (Рандомизированный коммуникационный протокол проверки равенства данных)

Входная ситуация:

C_{I} содержит последовательность x из n битов $x = x_1 \dots x_n$,

C_{II} содержит последовательность y из n битов $y = y_1 \dots y_n$.

Цель: Определить, действительно ли $x = y$.

Этап 1. Компьютер C_{I} случайно выбирает некоторое простое число p из интервала $[2, n^2]$.

{Заметим, что в этом интервале имеется примерно $Prim\left(n^2\right) \sim n^2/\ln n^2$ простых чисел. Следовательно, для такого выбора необходимо примерно $\lceil \log_2 n^2 \rceil \leq 2 \cdot \lceil \log_2 n \rceil$ случайных битов.}

[9] Это означает, что оптимальная коммуникационная стратегия – это пересылка *всех* данных от C_{I} к C_{II} для их последующего сравнения.

[10] Без ошибок в пересылке хотя бы одного из них – это тоже является нетривиальной задачей.

Этап 2. C_{I} вычисляет значение

$$s = Number(x) \bmod p$$

и посылает двоичное представление s и p на компьютер C_{II}.

{Поскольку двоичные представления чисел s и p состоят не более чем из $\lceil \log_2 n^2 \rceil$ битов ($s \le p < n^2$), длина такого сообщения не превосходит $4 \cdot \lceil \log_2 n \rceil$. }

Этап 3. После прочтения s и p компьютер C_{II} вычисляет значение

$$q = Number(y) \bmod p.$$

Если $q \neq s$, то C_{II} выводит «$x \neq y$».
Если $q = s$, то C_{II} выводит «$x = y$».

Проанализируем работу протокола $R = (C_{\mathrm{I}}, C_{\mathrm{II}})$. Во-первых, мы посмотрим на сложность – измеряемую числом битов, необходимых для такой коммуникации, – а затем проанализируем надёжность (вероятность ошибки) протокола R.

Единственная информация, передаваемая этим протоколом – двоичные представления натуральных чисел s и p. Как мы уже заметили, $s \le p < n^2$, следовательно длина сообщения не превышает

$$2 \cdot \lceil \log_2 n^2 \rceil \le 4 \cdot \lceil \log_2 n \rceil.$$

Например, для рассматриваемого нами значения $n = 10^{16}$ двоичная длина сообщения не превышает $4 \cdot 16 \cdot \lceil \log_2 10 \rceil = 256$. Это – очень короткое сообщение, оно может быть передано без каких-либо проблем.

Далее покажем не только то, что для большинства входов (начальных ситуаций) эта рандомизированная стратегия действительно работает – но также и то, что для каждого входа вероятность получения правильного ответа весьма высока. Анализируя вероятность ошибки, мы будем различать две возможности – в зависимости от того, равны ли значения x и y.

- Пусть $x = y$. Тогда

  $$Number(x) \bmod p = Number(y) \bmod p$$

  для всех простых чисел p. Поэтому компьютер C_{II} обязательно выводит ответ «равно» – т. е. вероятность ошибки равна 0.
- Пусть $x \neq y$. Тогда мы получаем *не*правильный ответ «равно» только если компьютер C_{I} выбрал такое простое p, что

  $$z = Number(x) \bmod p = Number(y) \bmod p.$$

  Это означает, что для некоторых натуральных чисел x' и y'

  $$Number(x) = x' \cdot p + z \quad \text{и} \quad Number(y) = y' \cdot p + z.$$

  Из последнего следует, что

  $$Number(x) - Number(y) = x' \cdot p - y' \cdot p = (x' - y') \cdot p$$

– т. е. p является делителем числа

$$w = |\, Number(x) - Number(y)|.$$

Итак, протокол $R = (C_{\mathrm{I}}, C_{\mathrm{II}})$ выводит неверный ответ только если выбранное простое p является делителем w. Это p было случайно выбрано из $Prim\left(n^2\right)$ простых чисел, принадлежащих множеству $\{2, 3, \ldots, n^2\}$ – с помощью равномерного распределения. Поэтому для вычисления вероятности ошибки достаточно посчитать, как много простых чисел среди данных (т. е. среди $Prim\left(n^2\right) \sim n^2/\ln n^2$ чисел) являются делителями числа w. Поскольку длина двоичного представления чисел x и y равна n, мы получаем

$$w = |\, Number(x) - Number(y)| < 2^n.$$

Очевидно, что мы можем разложить на множители число w, получая

$$w = p_1^{i_1} p_2^{i_2} \ldots p_k^{i_k},$$

где $p_1 < p_2 < \cdots < p_k$ – простые, а $i_1, i_2, \ldots, i_k$ – некоторые натуральные числа.[11] Наша цель – доказать, что

$$k \leq n - 1;$$

докажем это методом от противного.
Предположим, что $k \geq n$. Тогда

$$w = p_1^{i_1} p_2^{i_2} \ldots p_k^{i_k} \geq p_1 p_2 \ldots p_n > 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n = n! > 2^n,$$

что противоречит неравенству $w < 2^n$.
Теперь мы должны доказать, что число w имеет не более $n - 1$ различных простых делителей. Поскольку мы могли выбрать любое простое число из множества $\{2, 3, \ldots, n^2\}$ с одной и той же вероятностью, то для достаточно больших n вероятность выбора простого p, являющегося делителем w, не превышает значения

$$\frac{n-1}{Prim\left(n^2\right)} \leq \frac{n-1}{n^2/\ln n^2} \leq \frac{\ln n^2}{n}.$$

Итак, вероятность ошибки протокола R для двух отличных друг от друга входов x и y не превышает значения

$$\frac{\ln n^2}{n},$$

которое, в свою очередь, для $n = 10^{16}$ не превышает $0.36892 \cdot 10^{-14}$.

Вероятность ошибки такого уровня не является реальным риском – но предположим, что какой-нибудь пессимист ею не удовлетворён, и хочет её существенно понизить. В таком случае мы можем выполнить работу протокола $R = (C_{\mathrm{I}}, C_{\mathrm{II}})$, например, 10 раз – всегда независимо выбирая требуемое для его работы простое число.

Протокол $\mathbf{R_{10}}$

[11] Т. к. из элементарной теории чисел нам известно, что любое натуральное число может быть единственным образом разложено на множители.

Входная ситуация:

C_{I} содержит последовательность x из n битов $x = x_1 \dots x_n$,

C_{II} содержит последовательность y из n битов $y = y_1 \dots y_n$.

Цель: Определить, действительно ли $x = y$.

Этап 1. Компьютер C_{I} с помощью одного и того же алгоритма выбирает 10 случайных простых чисел

$$p_1, p_2, \dots, p_{10}$$

из множества $\{2, 3, \dots, n^2\}$.

Этап 2. C_{I} вычисляет

$$s_i = \mathit{Number}(x) \bmod p_i$$

для $i = 1, 2, \dots, 10$ и посылает двоичное представление чисел

$$p_1, p_2, \dots, p_{10}, s_1, s_2, \dots, s_{10}$$

на компьютер C_{II}.

Этап 3. После получения этих чисел компьютер C_{II} вычисляет значения

$$q_i = \mathit{Number}(y) \bmod p_i$$

для $i = 1, 2, \dots, 10$. Если существует некоторое $i \in \{1, 2, \dots, 10\}$, такое что $q_i \neq s_i$, то C_{II} выводит «$x \neq y$». Иначе (т. е. если $q_j = s_j$ для всех $j \in \{1, 2, \dots, 10\}$), C_{II} выводит «$x = y$».

Заметим, что вычислительная сложность алгоритма R_{10} в 10 раз больше, чем R. Но для $n = 10^{16}$ сообщение состоит не более чем из 2560 битов – поэтому последний факт не имеет значения.

Очень важен ответ на следующий вопро:

в чём здесь выгода – относительно вероятности ошибки?

Если $x = y$, то мы снова получаем ситуацию, когда протокол R_{10} всегда (т. е. с вероятностью 1) даёт правильный ответ «$x = y$» – поэтому вероятность ошибки равна 0.

Однако если $x \neq y$, то протокол R_{10} выводит неправильный ответ «$x = y$» только если все 10 выбранных простых чисел принадлежат тем максимальным $n-1$ числам, которые являются делителями значения

$$w = |\mathit{Number}(x) - \mathit{Number}(y)|.$$

Поскольку 10 простых чисел выбирались в 10 независимых экспериментах, вероятность ошибки не превышает

$$\left(\frac{n-1}{\mathit{Prim}\,(n^2)}\right)^{10} \leq \left(\frac{\ln n^2}{n}\right)^{10} = \frac{2^{10} \cdot (\ln n)^{10}}{n^{10}}.$$

Поэтому для $n = 10^{16}$ вероятность ошибки меньше, чем

$$0.4717 \cdot 10^{-141}.$$

Если мы примем во внимание, что число микросекунд, прошедших с момента Большого Взрыва, записывается 24 цифрами, а число протонов в известной части Вселенной – 79 цифрами, то реализация события, имеющего вероятность менее 10^{-141}, должно вызвать удивление. Заметим ещё, что даже если бы мог быть осуществлён детерминированный коммуникационный протокол с 10^{16} битами, сравнительная сложность ясно говорила бы в пользу рандомизированного протокола.

Можно многому научиться в процессе разработки несложного протокола R_{10}, состоящего из нескольких независимых повторений протокола R. Мы видим, что вероятность ошибки некоторого алгоритма может быть существенно понижена путём нескольких независимых его выполнений. В таких случаях, как вышеприведённый коммуникационный протокол, даже несколько повторений ведут к существенному уменьшению вероятности ошибки.

Упражнение 8.5. Пусть k – некоторое натуральное число. Рассмотрим протокол R_k, который основан на выборе k простых чисел – т. е. на k независимых выполнений протокола R. Оцените вероятность ошибки R_k как функцию аргумента k.

Упражнение 8.6. Другой подход к уменьшению вероятности ошибки – заменить R на следующий протокол Q_r, где $r \in \mathbb{N} - \{1, 2\}$. Протокол Q_r работает в точности как R – за исключением того, что некоторое требуемое простое число p выбирается из множества $\{2, 3, \ldots, n^r\}$ вместо $\{2, 3, \ldots, n^2\}$. Оцените коммуникационную сложность и вероятность ошибки протокола Q_r для каждого натурального $r \geq 2$.

Упражнение 8.7. Пусть $\delta > 1$ – некоторое натуральное число. Разработайте рандомизированный протокол для сравнения двух баз данных размера n, который работает с вероятностью ошибки, не превышающей $1/\delta$. Который из двух подходов – из упражнения 8.5 или 8.6 – является более эффективным с точки зрения коммуникационной сложности? Что лучше выбирать – несколько малых простых чисел или одно большое?

8.4 Избыток свидетельств и проверка простоты числа

В этом разделе мы ответим на вопрос, почему рандомизированный протокол R эффективнее любого детерминированного протокола, решающего ту же самую задачу – и, более того, почему отношение сложностей протоколов при этом экспоненциально. Мы покажем, что протокол R является одним из приложений метода избытка свидетельств. Для этого сначала опишем сам метод более подробно.

Рассмотрим произвольную проблему принадлежности – в которой вопрос состоит в том, обладает ли заданный вход некоторым особым свойством. Предположим, что не существует эффективных алгоритмов решения этой[12] проблемы – или, по крайней мере, что такие алгоритмы ещё не разработаны.

Начнём с подходящей формулировки понятия «свидетельство». Это понятие (см. также определение 6.43) должно представлять собой некоторую информацию, дополняющую входные данные; такая информация полезна для эффективного доказательства того, что заданный вход обладает необходимым нам свойством – или, наоборот, не обладает им. В нашем протоколе R в качестве свидетельства того, что $x \neq y$, можно рассматривать простое число p – в том случае, если

[12] Конкретной, рассматриваемой нами.

$$Number(x) \bmod p \neq Number(y) \bmod p.$$

Если бы у нас был некоторый оракул[13], дающий необходимое нам простое p (если оно существует) – то мы могли бы эффективно проверить выполнение условия «x отличается от y». Однако очевидно, что такого оракула у нас нет. Другими словами – мы не можем эффективно вычислить (реализовать, запрограммировать) подобное свидетельство (свидетельство того, что $x \neq y$): из существования такого вычисления следовало бы наличие эффективного детерминированного алгоритма для труднорешаемых задач.

Итак, заранее указать *конкретное* свидетельство мы не можем. Но для проектирования эффективного *рандомизированного* алгоритма нам достаточно иметь множество свидетельств-*кандидатов* для любого входа. Отметим, что это множество кандидатов должно содержать достаточно много элементов.

В примере протокола R возможными свидетельствами являются

$$Prim\left(n^2\right) \sim n^2 / \ln n^2$$

простых чисел из интервала $[2, n^2]$. Из этих $Prim\left(n^2\right)$ свидетельств-кандидатов по крайней мере $Prim\left(n^2\right) - (n-1)$ «говорят, что $x \neq y$» – в случае различных входов x и y.

Следовательно, вероятность выбора свидетельства из множества свидетельств-кандидатов не менее

$$\frac{\frac{n^2}{\ln n^2} - (n-1)}{\frac{n^2}{\ln n^2}} \geq 1 - \frac{\ln n^2}{n}.$$

Такая оценка нам подходит – поскольку это значение очень близко к 1. Но даже в ситуации, когда вероятность выбора свидетельства составляет только 1/2, избыток свидетельств всё ещё более чем достаточен: при случайном выборе нескольких кандидатов вероятность получения по крайней мере одного свидетельства растёт достаточно быстро.

Теперь ответим на такой вопрос: почему мы всё же не способны эффективно найти свидетельство с помощью какого-нибудь вспомогательного детерминированного алгоритма – ведь во множестве свидетельств-кандидатов имеется очень много подходящих? Детерминированный подход состоял бы в том, чтобы последовательно проверять каждое из свидетельств-кандидатов – причём так, чтобы достаточно быстро найти нужное нам свидетельство. Но проблема состоит в том, что для различных входов распределение свидетельств среди кандидатов может быть совершенно разным. Поэтому для каждой заранее предложенной стратегии поиска всегда можно найти такие входы, где эта стратегия терпит неудачу – т. е., другими словами, недостаточно эффективна.

Рассмотрим наш предыдущий пример. Для него мы даже можем *доказать*, что не существует стратегии, позволяющей паре компьютеров C_{I} и C_{II} эффективно найти свидетельство для любого входа (x, y). Мы опускаем техническую часть доказательства этого факта, приведём интуитивное его описание.

Сначала рассмотрим простую стратегию поиска – в которой простые числа перебираются по порядку, от наименьшего до наибольшего. Очевидно, что после n попыток эта стратегия находит свидетельство – поскольку число «кандидатов-несвидетельств»

[13] См. раздел 3.5.

не превышает $n-1$. К сожалению, n попыток приводят к сложности алгоритма $n \cdot 4 \cdot \log_2 n$ – что превышает сложность посылки всех битов от C_{I} к C_{II}.

Почему же *не* существует гарантии того, что в результате применения этой стратегии мы можем найти свидетельство после применения небольшого числа попыток? Потому что для входа (x, y), такого что

$$Number(x) - Number(y) = p_1 \cdot p_2 \cdot \cdots \cdot p_k,$$

где $k = \frac{n}{2(\log n)^2}$, а $p_1 < p_2 \cdots < p_k$ являются k наименьшими простыми числами, этой стратегии для нахождения свидетельства требуется $k+1$ попытка. И очевидно, что для любой другой нумерации простых чисел можно найти такие входные данные, которые для нахождения свидетельства требуют такого же числа попыток.

Метод избытка свидетельств весьма успешно применяется при разработке рандомизированных алгоритмов. Эффективная рандомизированная проверка числа на простоту[14] является типичным практическим приложением этого метода. Наиболее известный детерминированный алгоритм для проверки числа на простоту потребовал бы миллиарды лет работы – для чисел, имеющих обычный для современных криптографических приложений размер. Но объяснять способ определения подходящего свидетельства для проверки числа на простоту – это слишком сложная задача для нашей книги. Вместо неё мы решим только одну вспомогательную задачу, необходимую для этой проверки – а именно, покажем, как разработать эффективный рандомизированный алгоритм, который проверяет простоту всех нечётных чисел с нечётным значением $\frac{n-1}{2}$.

Во-первых – что означает термин «эффективный» для задач теории чисел? Для некоторого числа n размер входа, равного n, составляет $\lceil \log_2 n \rceil$. Это означает, что полиномиально-временной алгоритм для проверки простоты числа должен иметь временну́ю сложность, являющуюся полиномом относительно значения $\log_2 n$. Во многих приложениях нам нужно проверить многие сотни цифр (например, при $\log_{10} n \approx 500$) – и, следовательно, мы не можем допустить ни экспоненциальной сложности, ни полиномиальной с достаточно большой степенью полинома.

Наивный детерминированный алгоритм, который проверяет, действительно ли некоторое число из множества $\{2, 3, \ldots, \lfloor \sqrt{n} \rfloor\}$ является делителем заданного n, имеет экспоненциальную сложность относительно $\log_2 n$ (сложность не менее $\sqrt{n} = 2^{\frac{\log_2 n}{2}}$). В этом подходе свидетельством факта «число p составное» является каждое натуральное число $m > 1$, не равное p и являющееся делителем n. Но таких свидетельств[15] для разработки эффективных алгоритмов, вообще говоря, недостаточно. Если $n = p \cdot q$ для двух простых чисел p и q, то существуют только два свидетельства того факта, что число n составное – это p и q; при этом число свидетельств-кандидатов есть по крайней мере $\Omega(\sqrt{n})$. Следовательно, в данном случае нам необходим другой подход к данной проблеме.

Теорема 8.8 (малая теорема Ферма). *Для любого простого p и любого натурального a, таких что НОД$(a, p) = 1$,*

$$a^{p-1} \mod p = 1.$$

[14] Заметим, что эта задача относится к самым фундаментальным алгоритмическим проблемам, имеющим огромную практическую важность.

[15] Они основаны на классическом определении простых чисел.

Доказательство. Будем использовать тот факт, что каждое натуральное число допускает единственное разложение на простые множители. Поскольку p – простое, мы получаем, что для всех натуральных c и d

$$c \cdot d \mod p = 0 \quad \Leftrightarrow \quad c \mod p = 0 \text{ или } d \mod p = 0. \tag{8.2}$$

Пусть a – произвольное натуральное число, такое что НОД$(a, p) = 1$. Рассмотрим числа

$$m_1 = 1 \cdot a,\ m_2 = 2 \cdot a,\ \ldots,\ m_{p-1} = (p-1) \cdot a.$$

Докажем от противного, что для всех $u, v \in \{1, \ldots, p-1\}$, таких что $u \neq v$,

$$m_u \mod p \neq m_v \mod p$$

Предположим, что

$$m_u \mod p = m_v \mod p$$

для некоторых $u, v \in \{1, \ldots, p-1\}$, причём $u > v$. Тогда p является делителем числа

$$m_u - m_v = u \cdot a - v \cdot a = (u - v) \cdot a.$$

Но это невозможно – потому что:

- $u - v < p$, поэтому p не может быть делителем числа $u - v$;
- НОД$(a, p) = 1$, следовательно, p не может быть делителем числа a.

Следовательно,

$$|\{m_1 \mod p,\ m_2 \mod p,\ \ldots,\ m_{p-1} \mod p\}| = p - 1.$$

Теперь покажем, что каждое из чисел $m_i \mod p$ отлично от 0 (ни одно m_i не делится на p). Предположим противное, т. е. пусть для некоторого u

$$m_u \mod p = (u \cdot a) \mod p = 0.$$

Применяя (8.2), мы получаем, что

$$u \mod p = 0 \quad \text{или} \quad a \mod p = 0.$$

Но рассматриваемое нами p не является делителем ни u, ни a – поскольку $u < p$ и НОД$(a, p) = 1$. Следовательно,

$$\{m_1 \mod p, m_2 \mod p, \ldots, m_{p-1} \mod p\} = \{1, 2, \ldots, p-1\}. \tag{8.3}$$

Теперь рассмотрим число

$$m = m_1 \cdot m_2 \cdot \cdots \cdot m_{p-1}.$$

По определению m_i, мы получаем, что

$$m = 1 \cdot a \cdot 2 \cdot a \cdot \cdots \cdot (p-1) \cdot a = 1 \cdot 2 \cdot \cdots \cdot (p-1) \cdot a^{p-1}. \tag{8.4}$$

Совпадение множеств (8.3) влечёт равенство

$$m \mod p = 1 \cdot 2 \cdot \cdots \cdot (p-1) \mod p. \tag{8.5}$$

Из равенств (8.4) и (8.5) следует, что

$$1 \cdot 2 \cdot \cdots \cdot (p-1) \cdot a^{p-1} \mod p = 1 \cdot 2 \cdot \cdots \cdot (p-1) \mod p,$$

т. е.

$$a^{p-1} \mod p = 1.$$

□

Следующее утверждение обобщает малую теорему Ферма.

$$p \text{ простое} \iff \text{для всех } a \in \{1, \ldots, p-1\} \quad a^{\frac{p-1}{2}} \mod p \in \{1, p-1\}.$$

Это утверждение позволяет ввести альтернативное определение простых чисел. Согласно ему, мы можем получить новое свойство свидетельства для составных чисел – а именно, такое:

число $a \in \{1, \ldots, n-1\}$ является свидетельством того, что некоторое n является составным, если

$$a^{\frac{n-1}{2}} \mod n \notin \{1, n-1\}.$$

Следующая теорема показывает, что для нечётных составных чисел n с нечётным значением $\frac{n-1}{2}$ у нас есть достаточно много (избыток) свидетельств необходимого нам факта – того, что n является составным.

Теорема 8.9. *Для каждого нечётного натурального n с нечётным значением $\frac{n-1}{2}$ (т. е. если $n \mod 4 = 3$) выполнено следующее:*

(a) если n – простое, то

$$a^{\frac{n-1}{2}} \mod n \in \{1, n-1\} \text{ для всех } a \in \{1, \ldots, n-1\};$$

(b) если n – составное, то

$$a^{\frac{n-1}{2}} \mod n \notin \{1, n-1\}$$

по крайней мере для половины чисел a из множества $\{1, \ldots, n-1\}$.

Упражнение 8.10.* Докажите теорему 8.9.

Итак, для любого составного числа m, такого что $m \mod 4 = 3$, вероятность выбора свидетельства того, что число m составное, составляет по крайней мере $1/2$. Но чтобы удовлетворить нашим требованиям избытка свидетельств, мы должны ещё объяснить, как именно можно эффективно вычислить значение $a^{\frac{n-1}{2}} \mod n$.

Очевидно, что нам нельзя делать это путём $\frac{n-1}{2}$ умножений на a – поскольку в этом случае временна́я сложность относительно $\lceil \log_2 n \rceil$ экспоненциальная. Однако при $b = 2^k$ вычислить значение $a^b \mod p$ можно путём k умножений – с помощью следующего метода последовательного возведения в квадрат:

$$\begin{aligned}
a^2 \bmod p &= a \cdot a \bmod p,\\
a^4 \bmod p &= (a^2 \bmod p) \cdot (a^2 \bmod p) \bmod p,\\
a^8 \bmod p &= (a^4 \bmod p) \cdot (a^4 \bmod p) \bmod p,\\
&\vdots\\
a^{2^k} \bmod p &= (a^{2^{k-1}} \bmod p)^2 \bmod p.
\end{aligned}$$

Теперь рассмотрим общий случай, когда для некоторых $k \in \mathbb{N}$ и $b_i \in \{0,1\}$ при $i = 1,\ldots,k$ число b представлено как

$$b = \sum_{i=1}^{k} b_i \cdot 2^{i-1}$$

– т. е. $b = \mathit{Number}(b_k b_{k-1} \ldots b_1)$. Мы можем представить a^b в виде

$$a^b = a^{b_1 \cdot 2^0} \cdot a^{b_2 \cdot 2^1} \cdot a^{b_3 \cdot 2^2} \cdot \ldots \cdot a^{b_k \cdot 2^{k-1}}.$$

Заметим, что для подсчёта значения $a^b \bmod p$ мы сначала вычисляем все числа

$$a_i = a^{2^i} \bmod p$$

методом последовательного возведения в квадрат. После этого мы вычисляем произведение по модулю p всех таких чисел a_i, для которых $b_i = 1$. Применение этого подхода для вычисления значения $a^{\frac{n-1}{2}} \bmod n$ (для некоторого $a \in \{1,\ldots,n-1\}$) имеет то преимущество, что в процессе всего вычисления мы работаем только с числами множества $\{0,1,\ldots,n-1\}$ – т. е. с такими, длина двоичного представления которых не превышает $\lceil \log_2 n \rceil$. Число умножений для этих чисел при вычислении значения $a^{\frac{n-1}{2}} \bmod n$ не превышает

$$2 \cdot \lceil \log_2 \frac{n-1}{2} \rceil \in O(\log_2 n).$$

Применяя логарифмическое измерение временной сложности, мы получаем, что сложность всего вычисления находится в пределах $O((\log_2 n)^2)$.

Таким образом, на основе описанной выше теории можно сформулировать следующий эффективный рандомизированный алгоритм проверки простоты числа.

Алгоритм Соловея–Штрассена

Вход: Нечётное число n с нечётным значением $\frac{n-1}{2}$.
Этап 1. Выбрать некоторое случайное $a \in \{1,2,\ldots,n-1\}$, применяя равномерное распределение.
Этап 2. $x := a^{\frac{n-1}{2}} \pmod n$.
Этап 3.

```
if x ∈ {1, n − 1}
then output («простое»)
else output («составное»)
```

Проанализируем вероятность ошибки данного рандомизированного алгоритма проверки простоты числа.

Если n простое, то, согласно теореме 8.9 (а), для всех $a \in \{1,\ldots,n-1\}$

$$a^{\frac{n-1}{2}} \pmod n \in \{1, n-1\}$$

– при этом алгоритм всегда выводит «простое», т. е. вероятность ошибки равна 0.

Если же n составное, то, согласно теореме 8.9 (b), случайно выбранное значение $a \in \{1, 2, \ldots, n-1\}$ является свидетельством того, что n составное, с вероятностью не менее $1/2$. Следовательно, вероятность ошибки не превышает $1/2$ – но очевидно, что она всё же слишком велика. Поэтому из множества $\{1, \ldots, n-1\}$ мы независимо выбираем, например, 20 случайных чисел $a_1, \ldots, a_{20}$ вместо одного – и выдаём ответ «простое» только если для всех $i \in \{1, \ldots, 20\}$ выполнены условия

$$a_i^{\frac{n-1}{2}} \bmod n \in \{1, n-1\}.$$

При этом мы понижаем вероятность ошибки до значения

$$\frac{1}{2^{20}} < 10^{-6}.$$

Упражнение 8.11. Как сильно можно понизить вероятность ошибки алгоритма Соловея–Штрассена путём $k \geq 2$ независимых выполнений? Приведите подробное обоснование своего ответа.

8.5 Дактилоскопические методы и эквивалентность двух полиномов

В разделе 8.3 мы применили метод избытка свидетельств для разработки эффективного рандомизированного коммуникационного протокола, предназначенного для сравнения двух больших чисел $Number(x)$ и $Number(y)$. Этот специальный случай метода избытка свидетельств называется дактилоскопическим методом; обобщённо он может быть описан следующим образом.

Дактилоскопический метод

Постановка задачи: Определить эквивалентность двух объектов O_1 и O_2, полные описания которых слишком велики. Пусть M – некоторое «подходящее» множество отображений, действующих из множества всех возможных полных описаний этих объектов во множество их частичных представлений («отпечатков пальцев»).

Этап 1. Случайным образом выбрать некоторое отображение h из множества M.

Этап 2. Вычислить $h(O_1)$ и $h(O_2)$.

(При этом $h(O)$ называется **отпечатками пальцев** объекта O.)

Этап 3.

```
if h(O1) = h(O2)
then output «O1 и O2 эквивалентны»
else output «O1 и O2 не эквивалентны»
```

Для рандомизированного протокола, разработанного в разделе 8.3, объектами O_1 и O_2 являлись два больших числа, состоящие из n битов ($n = 10^{16}$). Множество M при этом было

$$M = \{h_p \mid p \text{ простое}, p \leq n^2,\ h_p(m) = m \bmod p \text{ для всех } m \in \mathbb{N}_0\}.$$

Для случайно выбранного простого p значения

$$h_p(O_1) = O_1 \bmod p \quad \text{и} \quad h_p(O_2) = O_2 \bmod p$$

были отпечатками пальцев объектов O_1 и O_2.

Основная идея метода заключается в том, что представление $h_p(O)$ существенно короче представления самого объекта O – поэтому сравнение значений $h_p(O_1)$ и $h_p(O_2)$ существенно проще, чем непосредственное сравнение объектов O_1 и O_2. Но в полной мере применить эту идею можно только в том случае, когда $h_p(O_i)$ является *не*полным описанием объекта O_i – и поэтому в принципе возможно принятие неправильного решения.

Вторая идея, также необходимая для рассматриваемого нами дактилоскопического метода, основана на методе избытка свидетельств. Пусть M – множество свидетельств-кандидатов того факта, что рассматриваемые входные объекты O_1 и O_2 *не* совпадают. Если для любой такой пары объектов во множестве M существует достаточно много[16] свидетельств факта $O_1 \neq O_2$, то мы можем уменьшить вероятность ошибки до сколь угодно малого значения.

Искусство применения дактилоскопического метода основано на подходящем выборе множества M. С одной стороны, сами «отпечатки пальцев» должны быть как можно более короткими – для того, чтобы сделать их сравнение как можно более эффективным. А с другой стороны, они должны содержать как можно больше информации о сравниваемых объектах[17] – для того, чтобы ограничить вероятность потери той части необходимой информации, которая и различает два объекта по их отпечаткам пальцев. Следовательно, разработчики алгоритмов при выборе конкретного отображения из O в $h(O)$ должны «соблюдать равновесие» между степенью сжатия с одной стороны – и получающейся вероятностью ошибки с другой. В разделе 8.3 мы получили вероятность ошибки, стремящуюся к 0 при увеличении размера входа, – и при этом для соответствующего отображения из O в $h(O)$ добились логарифмического сжатия.

В качестве примера применения дактилоскопического метода рассмотрим следующую проблему эквивалентности. Она заключается в определении, эквивалентны ли для некоторого заданного простого p два полинома от нескольких переменных над конечным полем $\mathbb{Z}_p$.[18] Для этой проблемы неизвестно детерминированных полиномиально-временны́х алгоритмов – но она может быть эффективно решена дактилоскопическим методом.

На первый взгляд непонятно, почему не существует полиномиально-временны́х алгоритмов для данной проблемы: ведь сравнение двух полиномов может быть проведено путём простого сравнения коэффициентов соответствующих слагаемых, поскольку два полинома, P_1 и P_2, эквивалентны тогда и только тогда, когда все коэффициенты P_1 совпадают с соответствующими коэффициентами P_2. Но проблема заключается в том,

[16] Относительно $|M|$.

[17] Это и является причиной того, что данный метод получил название дактилоскопического – поскольку в криминалистике «настоящие» отпечатки пальцев рассмтариваются в качестве практически однозначной идентификации человека.

[18] Два полинома $P_1(x_1, \ldots, x_n)$ и $P_2(x_1, \ldots, x_n)$ называются эквивалентными над полем $\mathbb{Z}_p$, если для всех $(\alpha_1, \ldots, \alpha_n) \in (\mathbb{Z}_p)^n$ выполнено следующее:

$$P_1(\alpha_1, \ldots, \alpha_n) \equiv P_2(\alpha_1, \ldots, \alpha_n) \pmod p.$$

что для выполнения такого сравнения коэффициентов мы должны сначала перевести полином в т. н. нормальную форму; ею для полинома над n переменными $x_1, x_2, \ldots, x_n$, имеющего степень[19] d, является

$$\sum_{i_1=0}^{d} \sum_{i_2=0}^{d} \cdots \sum_{i_n=0}^{d} c_{i_1,i_2,\ldots,i_n} \cdot x_1^{i_1} \cdot x_2^{i_2} \cdot \ldots \cdot x_n^{i_n}.$$

Однако входные полиномы, для которых мы должны проверить эквивалентность, могут иметь произвольную форму – например, иметь вид

$$P(x_1, x_2, x_3, x_4, x_5, x_6) = (x_1 + x_2)^{10} \cdot (x_3 - x_4)^7 \cdot (x_5 + x_6)^{20}.$$

Применяя бином Ньютона

$$(x_1 + x_2)^n = \sum_{k=0}^{n} \binom{n}{k} \cdot x_1^k \cdot x_2^{n-k},$$

мы получаем, что $P(x_1, x_2, x_3, x_4, x_5, x_6)$ имеет в точности $(10+1) \cdot (7+1) \cdot (20+1) = 1848$ слагаемых с ненулевыми коэффициентами. Таким образом, нормальная форма полинома может быть экспоненциально длиннее его входного представления – поэтому мы, вообще говоря, не можем получить нормальную форму за полиномиальное время.

Итак, если мы хотим создать эффективный алгоритм, то должны найти путь для сравнения двух полиномов без создания их нормальной формы. Для этого мы и применяем дактилоскопический метод. Пусть $P_1(x_1, \ldots, x_n)$ и $P_2(x_1, \ldots, x_n)$ – полиномы над полем $\mathbb{Z}_p$ для заданного простого p. Будем говорить, что некоторое $\alpha = (\alpha_1, \ldots, \alpha_n) \in (\mathbb{Z}_p)^n$ является свидетельством того, что

$$P_1(x_1, \ldots, x_n) \not\equiv P_2(x_1, \ldots, x_n),$$

если

$$P_1(\alpha_1, \ldots, \alpha_n) \bmod p \neq P_2(\alpha_1, \ldots, \alpha_n) \bmod p.$$

«На языке отпечатков пальцев» отображение

$$h_\alpha(P_1) = P_1(\alpha_1, \ldots, \alpha_n) \bmod p$$

как раз и является отпечатками пальцев полинома P_1. Приведённое выше определение отпечатков пальцев (свидетельств неэквивалентности) приводит к следующему рандомизированному алгоритму.

[19] Здесь степенью полинома от нескольких переменных называется максимальная из степеней переменных, входящих в этот полином.

Алгоритм AQP

Вход: Простое p, натуральные n и d. Два полинома P_1 и P_2 над n переменными $x_1, \ldots, x_n$; степень обоих полиномов не превышает d.

Этап 1. Выбрать случайным образом некоторый набор значений $\alpha = (\alpha_1, \ldots, \alpha_n) \in (\mathbb{Z}_p)^n$. [20]

Этап 2. Вычислить «отпечатки пальцев» – $h_\alpha(P_1) = P_1(\alpha_1, \ldots, \alpha_n) \bmod p$ и $h_\alpha(P_2) = P_2(\alpha_1, \ldots, \alpha_n) \bmod p$.

Этап 3.

```
if h_α(P_1) = h_α(P_2)
then output («P_1 ≡ P_2»)
else output («P_1 ≢ P_2»)
```

Проанализируем вероятность ошибки алгоритма AQP. Если полиномы P_1 и P_2 над полем $\mathbb{Z}_p$ эквивалентны, то для всех наборов $(\alpha_1, \alpha_2, \ldots, \alpha_n) \in (\mathbb{Z}_p)^n$ выполнено равенство

$$P_1(\alpha_1, \ldots, \alpha_n) = P_2(\alpha_1, \ldots, \alpha_n) \pmod p.$$

Поэтому для входов P_1 и P_2 при $P_1 \equiv P_2$ вероятность ошибки равна 0.

Теперь пусть P_1 и P_2 – два полинома над полем $\mathbb{Z}_p$, не являющиеся эквивалентными. Покажем, что при $p > 2nd$ вероятность ошибки алгоритма AQP меньше $1/2$.

Сначала отметим следующий несложный факт. Вопрос, верно ли, что

$$P_1(x_1, \ldots, x_n) \equiv P_2(x_1, \ldots, x_n),$$

эквивалентен такому вопросу: верно ли, что

$$Q(x_1, \ldots, x_n) = P_1(x_1, \ldots, x_n) - P_2(x_1, \ldots, x_n) \equiv 0.$$

Это означает, что если полиномы P_1 и P_2 *не*эквивалентны, то полином Q *не* является тождественным нулём (нулевым полиномом).

Нашей основной целью является доказательство следующего утверждения: число корней рассматриваемого нами полинома Q (зависящего от n переменных, имеющего степень d и не являющегося тождественным нулём) ограничено. Из этого факта будет следовать, что для утверждения

$$Q(\alpha) \not\equiv 0 \pmod p \quad - \text{ т. е. } P_1(\alpha) \not\equiv P_2(\alpha) \pmod p$$

имеется достаточно много свидетельств – т. е. достаточно много соответствующих $\alpha \in (\mathbb{Z}_p)^n$.

Начнём с более простого утверждения – широко известной теоремы о количестве корней полинома от одной переменной.

Теорема 8.12. *Пусть полином $P(x)$ (над любым полем) зависит от одной переменной x и имеет степень d. Тогда P имеет не более d корней – либо он тождественно равен 0.*

Доказательство. Докажем это утверждение по индукции относительно степени d.

[20] Выбор осуществляется относительно равномерного распределения вероятностей над $(\mathbb{Z}_p)^n$. Иными словами – все варианты значений набора α считаются равновероятными.

- Пусть $d = 0$. Тогда $P(x) = c$ для некоторой константы c. Если $c \neq 0$ (в противном случае P является тождественным нулём), то P корней не имеет.
- Предположим, что условие теоремы выполнено для некоторого $d-1$ при $d \geq 1$; докажем такой же факт для значения d.
  Пусть $P(x) \not\equiv 0$, и пусть a – корень полинома P. Тогда

$$P(x) = (x-a) \cdot P'(x),$$

  где $P'(x) = \frac{P(x)}{(x-a)}$ – некоторый полином степени $d-1$. Согласно предположению индукции, $P'(x)$ имеет не более $d-1$ корней. Следовательно, $P(x)$ имеет не более d корней. □

Теперь мы готовы доказать, что для каждого достаточно большого простого p много свидетельств неэквивалентности различных полиномов P_1 и P_2 над полем $\mathbb{Z}_p$.

Теорема 8.13. *Пусть d, n и простое p – некоторые натуральные числа, а $Q(x_1, \ldots, x_n) \not\equiv 0$ – некоторый полином над полем $\mathbb{Z}_p$, зависящий от n переменных $x_1, \ldots, x_n$, – причём степень полинома Q не превышает d.*[21] *Тогда число корней полинома Q не превышает*

$$n \cdot d \cdot p^{n-1}.$$

Доказательство. Докажем теорему 8.13 по индукции относительно n.

- Пусть $n = 1$. Тогда, согласно теореме 8.12, число корней полинома $Q(x_1)$ не превышает d, которое при $n = 1$ можно записать в виде $d = n \cdot d \cdot p^{n-1}$.
- Предположим, что утверждение 8.13 выполняется для значения $n-1$ при некотором натуральном n. Докажем это же утверждение для n.
  Мы можем представить Q в виде

$$\begin{aligned} Q(x_1, x_2, \ldots, x_n) &= Q_0(x_2, \ldots x_n) + x_1 \cdot Q_1(x_2, \ldots, x_n) + \ldots \\ &\quad + x_1^d \cdot Q_d(x_2, \ldots, x_n) \\ &= \sum_{i=0}^{d} x_1^i \cdot Q_i(x_2, \ldots, x_n) \end{aligned}$$

  для некоторых полиномов

$$Q_0(x_2, \ldots x_n),\ Q_1(x_2, \ldots, x_n),\ \ldots,\ Q_d(x_2, \ldots, x_n).$$

  Если $Q(\alpha_1, \alpha_2, \ldots, \alpha_n) \equiv 0 \pmod p$ для некоторого $\alpha = (\alpha_1, \ldots, \alpha_n) \in (\mathbb{Z}_p)^n$, то выполняется одно из следующих двух условий:
  (a) $Q_i(\alpha_2, \ldots, \alpha_n) \equiv 0 \pmod p$ для всех $i = 0, 1, \ldots, d$;
  (b) существует некоторое $j \in \{0, 1, \ldots, d\}$, такое что $Q_i(\alpha_2, \ldots, \alpha_n) \not\equiv 0 \pmod p$, а α_1 является корнем полинома

$$\begin{aligned} \overline{Q}(x_1) &= Q_0(\alpha_2, \ldots \alpha_n) + x_1 \cdot Q_1(\alpha_2, \ldots, \alpha_n) + \ldots \\ &\quad + x_1^d \cdot Q_d(\alpha_2, \ldots, \alpha_n). \end{aligned}$$

  Теперь посчитаем число корней – отдельно для случаев (a) и (b).

[21] Иными словами, степень каждой переменной полинома Q не превышает d.

(a) Поскольку $Q(x_1, \ldots, x_n) \not\equiv 0$, существует число $k \in \{0, 1, \ldots, d\}$, такое что

$$Q_k(x_2, \ldots, x_n) \not\equiv 0.$$

На основе предположения индукции мы получаем, что число корней полинома Q_k не превышает значения

$$(n-1) \cdot d \cdot p^{n-2}.$$

Следовательно, существует не более $(n-1) \cdot d \cdot p^{n-2}$ элементов $\overline{\alpha} = (\alpha_2, \ldots, \alpha_n) \in (\mathbb{Z}_p)^{n-1}$, таких что для всех $i \in \{0, 1, 2, \ldots, d\}$

$$Q_i(\overline{\alpha}) \equiv 0 \pmod{p}.$$

Поскольку значение α_1 переменной x_1 не связано с условием (a), оно может быть выбрано из множества $\{0, 1, \ldots, p-1\}$ произвольно. Таким образом, существует не более

$$p \cdot (n-1) \cdot d \cdot p^{n-2} = (n-1) \cdot d \cdot p^{n-1}$$

элементов $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n) \in (\mathbb{Z}_p)^n$, обладающих свойством (a).

(b) Согласно неравенству $\overline{Q}(x_1) \not\equiv 0$ мы получаем, что полином $\overline{Q}$ имеет не более d корней[22] – т. е. не более d значений $\alpha_1 \in \mathbb{Z}_p$, таких что $\overline{Q}(\alpha_1) \equiv 0 \pmod{p}$. Таким образом, существует не более

$$d \cdot p^{n-1}$$

значений наборов $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n) \in (\mathbb{Z}_p)^n$, которые удовлетворяют свойству (b).

Объединяя полученное в (a) и (b), мы получаем, что число корней полинома $Q(x_1, \ldots, x_n)$ не превышает значения

$$(n-1) \cdot d \cdot p^{n-1} + d \cdot p^{n-1} = n \cdot d \cdot p^{n-1}.$$

□

Следствие 8.14. *Пусть d, n и простое p – некоторые натуральные числа, а $Q(x_1, \ldots, x_n) \not\equiv 0$ – некоторый полином над полем $\mathbb{Z}_p$, зависящий от n переменных $x_1, \ldots, x_n$, – причём степень полинома Q не превышает d. Тогда число свидетельств того, что $Q \not\equiv 0$, не меньше, чем*

$$\left(1 - \frac{n \cdot d}{p}\right) \cdot p^n.$$

Доказательство. Число элементов множества $(\mathbb{Z}_p)^n$ в точности равно p^n – а согласно теореме 8.13 мы получаем, что не более чем $n \cdot d \cdot p^{n-1}$ из них *не* являются свидетельствами. Следовательно, число свидетельств не меньше, чем

$$p^n - n \cdot d \cdot p^{n-1} = \left(1 - \frac{n \cdot d}{p}\right) \cdot p^n.$$

□

[22] См. теорему 8.12.

Итак, используя для выбора значений равномерное распределение, мы получаем, что вероятность выбора необходимого нам свидетельства[23] из p^n элементов множества $(\mathbb{Z}_p)^n$ больше или равна

$$\left(1 - \frac{n \cdot d}{p}\right).$$

Для $p > 2nd$ такая вероятность больше $1/2$. Поэтому, выбирая несколько независимых случайных элементов из множества $(\mathbb{Z}_p)^n$, можно сколь угодно приблизить к 1 вероятность нахождения хотя бы одного требуемого свидетельства.

Для некоторых приложений алгоритма AQP важно, что простое p может быть выбрано специальным, подходящим способом – зависящим от входных данных. Такая «степень свободы» может быть применена в ситуациях, подобных сведéнию проблемы эквивалентности к сравнению двух полиномов. Отметим ещё раз, что при этом мы не накладывали какие-либо ограничения на поле, над которым рассматриваются эти полиномы.

8.6 Заключение

Мы можем рассматривать любой рандомизированный алгоритм:

- как недетерминированный с некоторым распределением вероятностей для каждого недетерминированного выбора;
- или как распределение вероятностей для множества детерминированных алгоритмов.

Мы наблюдаем аналогичное случайное *управление* работой алгоритма в природе физических и биологических процессов; его главные особенности – простота и эффективность. Аналогично всё обстоит и при проектировании алгоритмов – где простые рандомизированные алгоритмы часто оказываются намного более эффективными, чем их лучшие детерминированные аналоги.

При разработке коммуникационного протокола, предназначенного для сравнения содержимого двух баз данных, мы получили поучительный пример того, на что способна рандомизация: относительно сложности рассмотренного нами рандомизированного протокола сложность лучшего его детерминированного аналога экспоненциальна. Идея разработки такого рандомизированного протокола основана на методе избытка свидетельств. Свидетельство – это некоторая дополнительная информация для входа, позволяющая алгоритму на его основе эффективно получать результат – несмотря на то, что нам неизвестно[24] эффективных детерминированных подходов, решающих эту проблему.

Для успешной реализации метода избытка свидетельств мы должны уметь находить такое множество свидетельств-кандидатов, что подмножество самих свидетельств является его существенной частью. В этом случае мы можем получить свидетельство путём случайного выбора кандидата из такого множества – возможно, многократного выбора. Причина невозможности эффективного детерминированного поиска подобного свидетельства заключается в том, что свидетельства распределены среди кандидатов очень нерегулярно – случайно. Вследствие этой хаотической структуры

[23] Того, что $Q \not\equiv 0$ – т. е. $P_1(x_1, \ldots, x_n) \not\equiv P_2(x_1, \ldots, x_n)$.

[24] Или, более того, известно, что не существует.

свидетельств-кандидатов при применении любой стратегии детерминированного поиска имеется риск большого числа ложных попыток. Для каждой конкретной задачи искусство применения метода избытка свидетельств заключается в поиске подходящего определения самого понятия «свидетельство». Мы уже показали, как можно определить это понятие для алгоритма рандомизированной проверки простоты числа – алгоритма, который применим для всех нечётных n с нечётным значением $\frac{(n-1)}{2}$. Отметим, что рассмотренная нами формулировка свидетельства делимости (основанная на альтернативном определении составного числа) может быть расширена – и использована для работы с любым натуральным числом.

Специальным случаем метода избытка свидетельств является дактилоскопический метод, применяемый для решения проблем эквивалентности. Идея этого метода состоит в описании определения «отпечатков пальцев»[25] для любого используемого в задаче сложного объекта[26] – и, таким образом, сводимости проверки эквивалентности таких сложных объектов к эффективному сравнению отпечатков пальцев. При таком подходе роль свидетельства играет случайно выбираемое отображение из множества сложных объектов во множество их отпечатков пальцев. Используя дактилоскопический метод мы можем, например, разработать эффективный (полиномиально-временной) рандомизированный алгоритм проверки эквивалентности двух полиномов. Всё это представляет большой интерес – в связи со следующими обстоятельствами:

- во-первых, мы не знаем детерминированных полиномиально-временны́х алгоритмов решения данной проблемы эквивалентности;
- и, кроме того, существуют многие другие проблемы, имеющие большую практическую важность и допускающие возможность сводимости к проверке эквивалентности двух полиномов.

Мотвани и Рахаван в оригинальной работе [48] опубликовали исчерпывающий обзор разработки рандомизированных алгоритмов. К сожалению, эта превосходная монография вряд ли может быть рекомендована новичкам – поскольку данная тема очень сложная. Введение в разработку рандомизированных алгоритмов приведено в [30, 32] (в обеих книгах – в главах 5). Много информации, связанной с рандомизированными протоколами, дано в [29, 31]. Сипсер в [65] привёл описание применения рандомизированного алгоритма, проверяющего эквивалентность двух полиномов, – этот алгоритм используется для решения проблемы семантического сравнения двух структур данных, используемых для представления Булевых формул. Подробный обзор идей и методов, связанных с разработкой рандомизированных алгоритмов вообще, дан Карпом в [36].

Первые рандомизированные полиномиально-временны́е алгоритмы проверки чисел на простоту были разработаны Соловеем и Штрассеном в [53], Миллером в [47] и Рабином в [54, 55]. Летом 2002 года Агравалом, Кайалом и Саксеной был достигнут важнейший результат – разработан первый детерминированный полиномиально-временной алгоритм проверки числа на простоту, см. [44]. Этот алгоритм работает за время $O((\log_2 n)^{12})$ для любого входа[27] n – поэтому для реальных приложений он не может быть рассмотрен в качестве серьёзного конкурента рандомизированных алгоритмов[28].

[25] Т. е. короткого – но при этом достаточного (релевантного) частичного описания.
[26] Т. е. объекта, имеющего сложное полное представление.
[27] Напомним, что длина входа для заданного натурального n составляет $\lceil \log_2(n+1) \rceil$.
[28] Выполняющихся за время $O((\log_2 n)^3)$.

Ваша идея – сумасшедшая.
Вопрос лишь в том,
настолько ли она сумасшедшая,
чтобы быть правдой.
Н. Бор

9

Теория связи и криптография

9.1 Цели и задачи главы

В прошлом столетии теоретическая информатика в первую очередь занималась исследованием последовательных вычислительных моделей, которые соответствуют т. н. концепции фон Неймана. Каковы же главные проблемы *будущего* интереса этой науки? Сейчас пользователь общается не только с конкретным компьютером, но и с компьютерными сетями – сложным взаимосвязанным миром, полным асинхронных и непредсказуемых действий. Текущее понимание понятия «вычисление» здесь пока не является достаточно глубоким, его улучшение – одна из главных задач современных исследований в информатике.

В коротком обзоре вряд ли может быть приведено всё многообразие исследовательских вопросов, относящихся к распределённым вычислениям – т. е. ко всевозможным связям между компьютерами, процессами и пользователями. Например, проблемы, связанные с разработкой и анализом коммуникационных алгоритмов (коммуникационных протоколов), а также с разработкой общедоступных надёжных сетей, напрямую зависят от доступных нам технологий разработки программного обеспечния. Эти технологии напрямую связаны с используемым аппаратным обеспечением («железом»), которое стремительно развивалось от классических телефонных до оптических сетей – и каждая новая технология открывают свой мир проблем разработки и оптимизации алгоритмов. И, поскольку невозможно привести короткий – но в то же время достаточно ёмкий – обзор этой темы, мы ограничимся рассмотрением поучительного примера проектирования объединённой сети – для иллюстрации формулировок тех проблем и методов, которые используются в этой области.

Бóльшая часть этой главы посвящена *криптографии* – науке, связанной, в первую очередь, с проблемами безопасности в коммуникационных сетях. Основная цель главы состоит в рассмотрении фундаментальных понятий криптографии – нужных для проектирования шифровальных протоколов (методов), в которых сообщения, переданные по компьютерным сетям открытым образом, обладают следующими возможностями:

- они сохраняют конфиденциальность – т. е. никто, за исключением получателя, не может прочесть их исходный текст;
- они защищены от несанкционированных манипуляций с данными.

В современной практике основные понятия криптографии – цифровая подпись, электронная коммерция, электронные выборы и т.п. – имеют огромную важность. А вся

современная криптография как наука является естественным продолжением понятий, идей и методов, развитых в предыдущих главах этой книги, – особенно теории сложности, проектирования алгоритмов для решения трудных проблем, рандомизации. Более того, криптография является одной из тех областей науки, которые удивляют своими результатами, часто находящимися в противоречии с нашим повседневным жизненным опытом. Эти результаты открывают внушительные возможности, ранее считавшиеся нереалистическими мечтами. Итак, криптография является именно той областью теоретической информатики, которая действительно способна нас удивить, – и, следовательно, в большей стпени, чем любая другая её область, способна привлечь интерес студентов к изучению теоретической информатики вообще.

Глава организована следующим образом. Раздел 9.2 посвящён классическим криптографическим системам. В разделе 9.3 представлена концепция криптосистем с открытым ключом, а в качестве иллюстрации рассмотрена известная система RSA. В разделе 9.4 показано использование подобных систем для разработки коммуникационных протоколов с цифровой подписью. В разделе 9.5 описаны протоколы передачи данных, применяющиеся в интерактивных системах проверки доказательств – рандомизированных и с нулевым разглашением; эти системы нужны, например, для проверки математических доказательств без их чтения. Наконец, в разделе 9.6 представлена разработка общедоступной коммуникационной сети – как примера решения конкретной оптимизационной задачи.

9.2 Классические криптографические системы

Криптология – название науки, которое переводится как «секретное письмо»; в ней мы различаем криптографию и криптоанализ. Криптография посвящена разработке криптосистем,[1] в этой главе мы будем иметь дело только с ней. Типичный сценарий работы криптографической системы, рассматриваемый далее в этой главе, приведён на рис. 9.1.

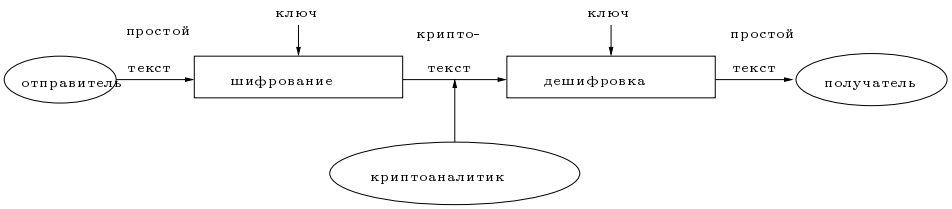


Рис. 9.1.

Персона, называемая **отправителем**, пытается отправить секретное сообщение другой персоне, называемой **получателем**. Секретное сообщение может быть представлено как текст – называемый в криптологии исходным текстом. При этом нет иного

[1] А крипоанализ посвящён математическим методам нарушения конфиденциальности информации без знания ключа, т. е. искусству проведения т. н. криптоатак – незаконного перехвата сообщений и их расшифровки (дешифровки).

способа передачи секретной информации, кроме как через общедоступную сеть; поэтому нельзя исключить то, что и посторонние лица могут «слушать» передаваемые по сети сообщения. Но посторонний человек не должен иметь возможности узнать нашу тайну – поэтому мы посылаем сообщение в закодированной форме. Способ шифрования (и дешифровки) является общей тайной отправителя и получателя сообщения, причём шифрование выполняется с использованием т. н. ключа. Зашифрованный текст называется криптотекстом. Криптотекст передаётся через общедоступную сеть. После получения криптотекста получатель его дешифрует, получая исходный текст.

Формально криптосистема является тройкой $(\mathcal{K}, \mathcal{A}, \mathcal{S})$, где:

- $\mathcal{K}$ – множество всех допустимых вариантов исходного текста;
- $\mathcal{A}$ – множество всех возможных криптотекстов;
- $\mathcal{S}$ – множество ключей.

Часто $\mathcal{K} = \Sigma^m$ для некоторых натурального m и алфавита Σ. Смысл такого допущения состоит в том, что исходный текст должен быть разделён на блоки (слова) длины m, и при этом каждый блок кодируется независимо от других. В таком случае мы также обычно считаем, что $\mathcal{A} = \Gamma^k$ для некоторых натурального k и алфавита Γ.

Каждый ключ $\alpha \in \mathcal{S}$ однозначно определяет взаимно-однозначное отображение E_α, действующее из $\mathcal{K}$ в $\mathcal{A}$. Поэтому кодирование соответствует вычислению $\mathrm{E}_\alpha(x)$ для исходного текста $x \in \mathcal{K}$, а дешифровка соответствует вычислению $\mathrm{E}_\alpha^{-1}(c)$ для криптотекста $c \in \mathcal{A}$. Обычно для обозначения функции E_α^{-1} (обратной к E_α) мы используем запись D_α.

Общие требования к криптосистемам таковы.

- Функции E_α и D_α должны быть эффективно вычислимыми.
- Без знания ключа α задача вычисления исходного текста x на основе заданного криптотекста $\mathrm{E}_\alpha(x)$ должна быть «трудной»[2].

К простейшим криптосистемам можно отнести CAESAR. В ней множество $\mathcal{K}$ исходных текстов, а также множество $\mathcal{A}$ криптотекстов, являются множеством всех слов над латинским алфавитом с 26 символами.[3] Множеством ключей $\mathcal{S}$ является $\{0, 1, 2, \ldots, 25\}$.[4] Для данного ключа $k \in \mathcal{S}$ кодирование состоит в замене каждого символа исходного текста на символ, отстоящий от него на k шагов направо в алфавитном порядке; а если при этом мы достигаем конца алфавита, то циклически продолжаем с его начала. Например, если $k = 3$, то заданный исходный текст

`CRYPTOGRAPHYISFASCINATING`,

мы заменяем на криптотекст

`FUBSWRJUDSKBLVIDVFLQDWLQJ`.

Очевидно, что соответствующая дешифровка заключается в перемещении на k символов в обратном порядке, т. е. налево.

[2] По крайней мере с точки зрения теории сложности.

[3] Более того, мы можем рассматривать $\mathcal{K}$ и $\mathcal{A}$ как латинский алфавит – точнее, как множество однобуквенных слов над этим алфавитом. При этом простейшие блоки исходного текста имеют длину 1.

[4] И иногда тоже считается латинским алфавитом.

Такая криптосистема очень слаба: если мы знаем, что используется именно система CAESAR, то криптоаналитик просто перебирает все ключи. Таким образом, что криптосистема с малым числом ключей малопригодна для практического использования: число возможных ключей должно быть столь велико, чтобы попытка перебрать все ключи была бы практически невозможной.

Система CAESAR может быть улучшена путём рассмотрения ключей из множества $\{0, 1, \ldots, 25\}^m$ – для некоторого достаточно большого натурального m. При этом для некоторого ключа

$$\alpha = \alpha_1, \alpha_2, \ldots, \alpha_m$$

мы разбиваем исходный текст на блоки длины m,[5] после чего заменяем i-й символ каждого блока на символ, получающийся из него со сдвигом на α_i позиций направо (т. е. в алфавитном порядке). Если $\alpha = 3, 1, 6$, то для исходного текста

$$\begin{array}{cccccccccccc} C & R & Y & P & T & O & G & R & A & P & H & Y \\ 3 & 1 & 6 & 3 & 1 & 6 & 3 & 1 & 6 & 3 & 1 & 6 \end{array}$$

мы получаем криптотекст

$$F\ S\ E\ S\ U\ U\ J\ S\ G\ S\ I\ E.$$

Эта криптосистема тоже может быть взломана. Например, для криптоатак на такие системы могут быть полезными знания статистического распределения вероятности появления конкретных символов в текстах на естественных языках. Однако существуют и такие классические криптосистемы – работающие достаточно быстро как при шифровании, так и при дешифровке – которые при современном уровне знаний практически невозможно взломать. Недостаток классических криптосистем состоит в том, что они обычно основаны на общем секретном ключе отправителя и получателя, и при этом знание механизма кодирования непосредственно подразумевает знание механизма дешифровки.[6] Следствием является то, что отправитель и получатель перед началом работы с криптосистемой должны согласиться использовать фиксированный секретный ключ, а для изменения этого ключа иметь специальный канал. И главная тема следующего раздела заключается именно в том, как можно решить эту проблему.

9.3 Системы с открытым ключом и RSA-кодирование

Классические криптосистемы, введённые в разделе 9.2, называются также симметрическими (симметричными) криптосистемам – поскольку знание алгоритма шифрования [дешифровки] автоматически даёт обратный алгоритм – дешифровки [шифрования]. Вследствие этого отправитель и получатель обладают одинаковыми знаниями о криптосистеме, т. е. обладают общим секретным ключом. И, помимо необходимости договариваться об этом ключе без использования какого-либо безопасного канала, у симметрических криптосистем есть и другой недостаток: если в систему коммуникации входит несколько лиц (сторон), то для нарушения безопасности всего секретного коммуникационного обмена достаточно только одного предателя.

[5] Т. е. $\mathcal{K}$ и $\mathcal{A}$ являются множествами слов длины m.

[6] Например, часто ключи для шифрования и дешифровки совпадают.

Революционная идея криптографии с открытым ключом, преодолевающая упомянутые недостатки, основана на следующем соображении, взятом из теории сложности. Мы ищем т. н. **одностороннюю функцию** f – обладающую следующими свойствами:

(a) f *может* быть эффективно вычислена;
(b) f^{-1}, вообще говоря, *не* может быть эффективно вычислена;
(c) f^{-1} может быть эффективно вычислена в том случае, если мы знаем специальный секрет – называемый **ловушкой**.[7]

Если у получателя имеется некоторая односторонняя функция f, то он может опубликовать f (сделать её общедоступной) – после чего каждый отправитель может использовать f для кодирования сообщений для этого получателя. После публикации функцию f (например, в телефонном справочнике) мы на основе её свойства (b) можем гарантировать, что никто не сумеет расшифровать криптотекст $f(x)$. Только получатель, обладающий функцией-ловушкой для f, может вычислить f^{-1} – и таким образом расшифровать принятое сообщение. Очевидно, что такие системы не являются симметричными – и вследствие открытой публикации функции f мы называем их **криптосистемами с открытым ключом**.[8]

Возникает вопрос о том, *существуют ли* вообще такие односторонние функции. Можно даже ожидать, что ответ на этот вопрос отрицателен – поскольку свойства (a), (b) и (c) вместе кажутся противоречивыми. Следующая простая идея показывает, что концепция односторонних функций всё же является реальной.

Таблица 9.1.

	Фамилия	Номер телефона
C	Cook	00128143752946
R	Rivest	00173411020745
Y	Yao	00127345912233
P	Papadimitriou	00372453008122
T	Turing	00192417738429
O	Ogden	00012739226541
R	Rabin	00048327450028
A	Adleman	00173555248001
P	Papadimitriou	00372453008122
H	Hopcroft	00013782442358
Y	Yao	00127345912233

Рассмотрим механизм кодирования, в котором каждый символ (буква) исходного текста независимо от других кодируется последовательностью из 14 цифр следующим образом. Для этой буквы *недетерминированно* выбирается имя из телефонной книги – при этом оно должно начинаться на эту букву – и в качестве шифра выбирается соот-

[7] Сама функция f^{-1} при этом называется функцией-ловушкой. Отметим аналогию между этим понятием и термином «свидетельство», использовавшемся в предыдущей главе.

[8] Другое название, применяемое как в русской, так и в английской литературе – асимметрические криптосистемы. Смысл такого названия следует из приведённого далее описания алгоритмов шифрования и дешифровки для этих систем. *(Прим. перев.)*

ветствующий номер телефона.[9] В таблице 9.1 приведён подобный пример кодирования для слова «CRYPTOGRAPHY». Данная процедура кодирования является недетерминированной, и некоторому исходному тексту может соответствовать несколько вариантов криптотекста – но, несмотря на это, каждый криптотекст однозначно определяет заданный исходный текст.

Теперь рассмотрим применение функции-ловушки получателя – являющейся в данном случае специальным телефонным справочником, отсортированным *относительно номеров телефонов*. Пользуясь им, получатель может эффективно выполнить дешифровку – а без знания этой ловушки, т. е. не обладая таким телефонным справочником, он должен применить полный (*линейный*) поиск в неотсортированном списке, сложность (временны́е затраты) которого на расшифровку одного символа пропорциональна длине телефонного справочника. Поскольку мы должны сделать метод кодирования общедоступным, эту криптосистему можно рассматривать как своеобразную игру на общедоступном ключе криптосистемы; мы употребляем слово «игра» только потому, что любому пользователю достаточно один раз отсортировать телефонный справочник по порядку возрастания номеров – получив тем самым функцию-ловушку для нашей процедуры кодирования.[10]

Итак, приведённый пример вряд ли может рассматриваться в качестве кандидата для реальной криптосистемы – но он показывает возможность применения на практике понятий односторонней функции и соответствующей функции-ловушки. В терминах вычислительной сложности эти понятия формализованы в следующем определении.

Определение 9.1. *Пусть Σ и Γ – некоторые алфавиты. Функция $f : \Sigma^* \to \Gamma^*$ называется* **односторонней функцией**, *если она удовлетворяет следующим свойствам:*

- *Существуют некоторые константы $c, d \in \mathbb{N}_0$, такие что для всех $x \in \Sigma^*$ выполнено следующее:*
$$\frac{1}{c} \cdot |x| \leq |f(x)| \leq d \cdot |x|.$$
{Это означает, что $|x|$ и $|f(x)|$ находятся в линейном отношении.}
- *Функция f может быть вычислена за полиномиальное время.*
- *Для каждого рандомизированного полиномиально-временно́го алгоритма A и каждого $k \in \mathbb{N}$ существует константа $n_{A,k}$, такая что для всех $n \geq n_{A,k}$ и случайно выбранного $w \in \Sigma^n$ вероятность*[11] *того, что $A(f(w)) = w$, меньше чем n^{-k}.*
{Это свойство гарантирует, что полиномиально много независимых выполнений рандомизированного полиномиально-временно́го алгоритма не могут обеспечить вычисление функции f^{-1} с вероятностью ошибки, априорно ограниченной некоторой константой.}

До настоящего времени никто не смог доказать, что некоторая конкретная функция является односторонней.[12] Это связано с тем, что очень сложно доказать нижнюю

[9] Если же телефонный номер короче 14 цифр, то перед ним добавляется соответствующее количество нулей.

[10] Отметим, что данная игра является полезной, «развивающей» – поскольку на её примере можно достаточно хорошо продемонстрировать возможности реальных криптосистем. *(Прим. перев.)*

[11] Вероятность рассматривается для случайного выбора варианта работы алгоритмом A, а также для случайного выбора слова w необходимой длины. При этом в обоих случаях все варианты выбора считаются равновероятными.

[12] Не доказано даже *существование* хотя бы одной односторонней функции.

границу вычислительных ресурсов, необходимых для решения этой проблемы – т. е. что конкретная функция f^{-1} удовлетворяет свойству (c).[13]

Однако нам известны некоторые вероятные кандидаты в односторонние функции – например, следующие.

- *Произведение* двух чисел, т. е. вычисление $f(x, y) = x \cdot y$, может быть выполнено эффективно. Обратной к f функцией является *разложение на множители* заданного числа n. В настоящее время неизвестно рандомизированных полиномиально-временны́х алгоритмов для разложения на множители заданного числа – и эта задача рассматривается как трудная.[14]
- Как уже было показано в главе 8, мы можем эффективно вычислить функцию $f(x) = a^x \bmod n$, используя метод последовательного возведения в квадрат. Обратная функция соответствует решению уравнения $a^x \equiv b \bmod n$ для заданных a, b и n. Последняя задача называется *проблемой дискретного логарифма* – и также рассматривается как трудная (не разрешимая рандомизированными алгоритмами за полиномиальное время).

Теперь опишем известную криптосистему с открытым ключом RSA.[15] Её надёжность основана на предположении, что проблема разложения на множители заданного числа является трудной.

Получатель определяет процедуры шифрования и дешифровки путём следующих вычислений. Он генерирует два больших[16] случайных простых числа p и q, после чего вычисляет произведения

$$n = p \cdot q \quad \text{и} \quad \varphi(n) = (p-1) \cdot (q-1).$$

Здесь φ является так называемой Эйлеровой функцией. Для каждого натурального n значение $\varphi(n)$ является количеством целых чисел a, принадлежащих множеству $1, 2, \ldots, n-1$, таких что $\text{НОД}\,(a, n) = 1$.

После этого получатель случайно выбирает достаточно большое значение $d > 1$, такое что

$$\text{НОД}\,(d, \varphi(n)) = 1. \tag{9.1}$$

Равенство (9.1) означает, что у d имеется единственный мультипликативный образ – пусть это e. Итак, получатель вычисляет[17] это значение e – обладающее свойством

$$e \cdot d \mod \varphi(n) = 1. \tag{9.2}$$

Числа n и e определяют открытый ключ, а числа p, q, $\varphi(n)$ и d являются секретными ключами получателя. Этот факт и является причиной того, что p, q и d должны быть сгенерированы случайным образом.

[13] Заметим, что мы не способны доказать даже более слабое требование – что детерминированная временна́я сложность функции f^{-1} может быть вычислена за полиномиальное время.

[14] Однако неизвестно, является ли разложение на множители NP-трудной задачей – но для наших целей это не имеет значения.

[15] Название RSA происходит от первых букв фамилий её разработчиков – River, Shamir и Adleman.

[16] Состоящих из нескольких сотен десятичных цифр.

[17] Это вычисление может быть эффективно выполнено с помощью алгоритма Евклида.

Теперь в качестве исходного текста будем рассматривать только числа, меньшие, чем n. Если же исходный текст длиннее (либо представлен в каком-либо ином виде), то нашей первой целью является его преобразоваие в последовательность цифр длины $\lceil \log_{10} n \rceil - 1$, после чего нужно закодировать эти цифры по-отдельности. Для заданного числа $w \in \{0, 1, \ldots, n-1\}$ кодирование производится с помощью функции

$$\mathrm{E}_{e,n}(w) = w^e \mod n.$$

А функция дешифровки для заданного криптотекста c такова:

$$\mathrm{D}_{d,n}(c) = c^d \mod n.$$

Как уже было показано в главе 8, функции $\mathrm{E}_{e,n}$ и $\mathrm{D}_{d,n}$ эффективно вычислимы методом последовательного возведения в квадрат. С помощью эффективных рандомизированных алгоритмов проверки числа на простоту (см. раздел 8.4) мы можем эффективно генерировать достаточно большие случайные простые числа. Мы выбираем число d случайным образом, после чего проверяем выполнение условия НОД $(d, \varphi(n)) = 1$ – т. е. определяем, действительно ли d и $\varphi(n)$ являются взаимно простыми. Если не являются, то мы выбираем другое значение d и проверяем ещё раз. Поскольку существует достаточно много чисел, взаимно простых с $\varphi(n)$, значение d может быть найдено эффективно. Как уже было упомянуто, число e можно эффективно определить с помощью алгоритма Евклида. Следовательно, мы можем эффективно запрограммировать всю криптосистему с открытым ключом RSA.

Мы не знаем каких-либо эффективных (рандомизированных) алгоритмов, способных вычислить значения p, q, $\varphi(n)$ и d для заданного открытого ключа (e, n). Знания хотя бы одного из этих четырёх чисел достаточно для взлома криптосистемы RSA. Также неизвестно эффективных алгоритмов определения исходного текста x на основе криптотекста $\mathrm{E}_{e,n}(x)$ и открытого ключа (e, n).

Теперь покажем, что криптосистема RSA действительно работает – т. е. что для всех $w < n$

$$\mathrm{D}_{d,n}(\mathrm{E}_{e,n}(w)) = w.$$

Для доказательства того, что $\mathrm{D}_{d,n}$ является инверсной функцией к $\mathrm{E}_{e,n}$ (для всех аргументов, принадлежащих множеству $\{0, 1, \ldots, n-1\}$), нам необходима теорема Эйлера – обобщение малой теоремы Ферма.

Теорема 9.2 (Эйлера). *Пусть w и n – два натуральных числа, причём НОД$(w, n) = 1$. Пусть*

$$\varphi(n) = \left| \{a \in \{1, 2, \ldots, n\} \mid \text{НОД}(a, n) = 1\} \right|$$

– т. н. Эйлеров номер n. Тогда

$$w^{\varphi(n)} \mod n = 1.$$

Доказательство. Теорема Эйлера может быть рассмотрена как последовательное применение следующих результатов теории групп:

1) порядок любого элемента группы является делителем порядка группы;
2) циклическая мультипликативная группа $(Z/(n))^*$ имеет порядок $\varphi(n)$.

На основе определения порядка элемента группы мы получаем, что для каждого $w \in (Z/(n))^*$ и порядка группы k выполняется равенство

$$w^k \mod n = 1.$$

Поскольку условие (a) для некоторого натурального b влечёт равенство

$$\varphi(n) = k \cdot b,$$

мы получаем, что

$$\begin{aligned} w^{\varphi(n)} \mod n &= w^{k \cdot b} \mod n \\ &= (w^k \mod n)^b \mod n \\ &= (1)^b \mod n = 1. \end{aligned}$$

Упражнение 9.3. Приведите альтернативное доказательство теоремы Эйлера – путём обобщения малой теоремы Ферма (раздел 8.4), переформулированной следующим образом.

Пусть $x_1, x_2, \ldots, x_{\varphi(n)} \in \{1, 2, \ldots, n-1\}$ – все числа x_i, обладающие свойством НОД $(x_i, n) = 1$. Тогда для каждого $a \in \{1, 2, \ldots, n-1\}$ последовательность

$$(ax_1 \bmod n,\ ax_2 \bmod n,\ \ldots,\ ax_{\varphi(n)} \bmod n)$$

является перестановкой чисел $x_1, x_2, \ldots, x_{\varphi(n)}$.

Теперь мы готовы доказать корректность криптосистемы RSA.

Теорема 9.4. *Пусть p, q, n, e и d обладают свойствами, описанными выше для системы* RSA. *Тогда для всех $w < n$ выполнено следующее:*

$$\mathrm{D}_{d,n}(\mathrm{E}_{e,n}(w)) = w^{ed} \mod n = w.$$

Доказательство. Согласно условиям выбора d и e, а также равенствам (9.1) и (9.2), мы получаем, что

$$e \cdot d = j \cdot \varphi(n) + 1 \tag{9.3}$$

для некоторого $j \in \mathbb{N}$. Поэтому мы для всех $w < n$ должны доказать, что

$$w^{j \cdot \varphi(n)+1} \mod n = w. \tag{9.4}$$

Будем различать три возможности – в зависимости от значений p, q и w.

- *Ни одно из (простых) чисел p, q не является делителем w.*
  Если p и q не являются делителями w, а $w < p \cdot q$, то

  $$\text{НОД}\,(p \cdot q, w) = 1.$$

  Следовательно, выполнены условия теоремы Эйлера для $n = p \cdot q$ и w. Отсюда

  $$w^{\varphi(n)} \mod n = 1$$

  и поэтому

  $$w^{j\varphi(n)} \mod n = 1. \tag{9.5}$$

  Умножив обе части равенства (9.5) на w, мы получаем требуемое равенство (9.4).

- *Ровно одно из (простых) чисел p и q является делителем w.*
  Без ограничения общности можно предполагать, что p является w делителем w, а q – не является. Согласно малой теореме Ферма[18],

$$w^{q-1} \mod q = 1,$$

  поэтому

$$w^{(q-1)\cdot(p-1)} \mod q = 1, \quad \text{т. е.} \quad w^{\varphi(n)} \mod q = 1.$$

  Отсюда мы получаем, что

$$w^{j\varphi(n)} \mod q = 1. \tag{9.6}$$

  Поскольку p является делителем w, равенство (9.6) выполняется также по модулю $n = p \cdot q$, т. е.

$$w^{j\varphi(n)} \mod n = 1.$$

  Умножая это равенство на w, мы получаем требование теоремы.
- *Оба (простых) числа, p и q, являются делителями w.*
  Этот случай невозможен – т. к. p и q являются простыми, а $p \cdot q > w$.

□

Криптосистемы с открытым ключом имеют много преимуществ по сравнению с симметрическими криптосистемами. Помимо вышеизложенного, они являются основой для создания различных коммуникационных протоколов (например, цифровых подписей), которые не могут быть созданы на основе симметрических криптосистем. Но, с другой стороны, классические симметрические криптосистемы имеют и свои преимущества над криптосистемами с открытым ключом. Вследствие своей возможной реализации «на уровне железа» симметрические криптосистемы обычно работают в сотни раз быстрее систем с открытым ключом. Поэтому сейчас является общепринятым использовать криптосистемы с открытым ключом только для изменения ключа симметрических криптосистем. Остальная же часть коммуникационных систем[19] реализуется с использованием симметрических криптосистем.

9.4 Цифровая подпись

Для иллюстрации некоторых очевидных приложений криптосистем с открытым ключом приведём два простых протокола для цифровых (электронных) подписей. С юридической точки зрения обычная (рукописная) подпись является одной из форм гарантии подлинности. Очевидно, что документы электронной коммуникации нельзя обеспечить рукописными подписями. Однако часто необходимо иметь возможность работы с цифровыми подписями, более того, с такими, подделать которые было бы тяжелее, чем рукописные.

Рассмотрим следующую ситуацию. Клиент K хочет подписать электронный документ для своего банка B. Например, K должен передать банку B право установления

[18] Заметим, что она является специальным случаем теоремы Эйлера – поскольку $\varphi(q) = q - 1$ для простого q.

[19] Причём наиболее «важная» часть.

подлинности перевода денег со своего счёта. На коммуникационные протоколы для таких цифровых подписей должны быть наложены следующие естественные требования.

(a) Банк B должен иметь возможность подтверждения правильности цифровой подписи клиента K – т. е. подтверждать подлинность K как владельца цифровой подписи. Это означает, что обе стороны, K и B, должны быть защищены от возможности криптоатак третьей стороны – фальсификатора F, который претендует на право называться K в коммуникациях с B.

(b) Клиент K должен быть защищён от сообщений, подделанных стороной B – в которых утверждается, что они должным образом подписаны самим K. Кроме того, это требование означает, что и у самого банка B нет возможности воспроизвести (подделать) подпись клиента K.

Упражнение 9.5. Разработайте коммуникационный протокол для цифровых подписей, который основан на симметрических криптосистемах и удовлетворяет требованию (a).

Одновременное выполнение обоих свойств (требований (a) и (b)) кажется более сложным, чем выполнение только одного требования (a),– поскольку на первый взгляд кажется, что свойства (a) и (b) противоречат друг другу. С одной стороны, свойство (a) требует, чтобы у стороны B были некоторые нетривиальные знания о подписи K – для проверки её подлинности. Но, с другой стороны, свойство (b) требует, чтобы B не знал слишком многого об этой подписи (особенно о самой процедуре её генерации) – для того, чтобы у него не было возможности её подделать.

Следующее простое решение этой проблемы возможно на основе криптосистем с открытым ключом. У клиента K имеется криптосистема с открытым ключом – с функциями шифровки E_K и дешифровки D_K. Банк B знает открытую функцию E_K. В этом случае K может подписать документ за следующие 2 этапа:

1) K вычисляет $\mathrm{D}_K(w)$ для документа w и посылает пару $(w, \mathrm{D}_K(w))$ банку B;
2) B проверяет, действительно ли $w = \mathrm{E}_K(\mathrm{D}_K(w))$, с помощью открытого ключа (открытой функции) E_K.

Проверим, что этот коммуникационный протокол удовлетворяет нашим ограничениям.

(a) Поскольку никто, за исключением K, не может вычислить $\mathrm{D}_K(w)$, банк B уверен, что именно K подписал документ w. Более того, поскольку ключ E_K является открытым, B также имеет возможность убедить любого желающего[20], что именно K подписал w, – это делается предъявлением стороной B пары $(w, \mathrm{D}_K(w))$.

(b) Требование (b) также удовлетворено – поскольку знание E_K и пары $(w, \mathrm{D}_K(w))$ не поможет подписать какой-нибудь другой документ u с помощью $\mathrm{D}_K(u)$.

Очень важно, что такая электронная подпись изменяет общий текст документа w – т. е. электронная подпись является не только текстом, добавляемым в конце документа.

Упражнение 9.6. Представленный выше протокол не пытается работать с w как с секретным текстом. Если третьей стороне доступна коммуникация между K и B, то она может получить информацию, содержащуюся в w. Поэтому для многих приложений может быть очень полезным следующее дополнительное требование (c).

[20] Того, кто знает открытую функцию E_K и уверен, что именно она является открытым ключом клиента K.

(c) Третья сторона не может узнать содержание подписанного документа – даже если она в состоянии прочесть все коммуникационные сообщения.

Разработайте коммуникационный протокол, который удовлетворяет всем трём требованиям – (a), (b) и (c).

Теперь рассмотрим более сложную проблему, называемую **проблемой установления подлинности**. В ней целью является не столько подписание документа, сколько убеждение кого-то в его подлинности. Эта проблема накладывает следующие требования на коммуникационный протокол:

(a′) совпадает с (a);
(b′) клиент K должен быть защищён от таких действий B, в которых B пытается убедить некоторую третью сторону в том, что *он* (т. е. B) является K.[21]

Представленный выше коммуникационный протокол не удовлетворяет всем условиям установления подлинности – поскольку B может изучить подпись $(w, \mathrm{D}_K(w))$ в этом протоколе и впоследствии использовать её для убеждения в том, что некоторая третья сторона является K. Очевидно, что существуют ситуации, где это нежелательно. Более того, некая третья сторона, слушающая сообщения между K и B, может также выучить подпись $(w, \mathrm{D}_K(w))$ стороны K. Итак, противник, обладая знаниями E_K, может проверять корректность подписи $(w, \mathrm{D}_K(w))$ – и использовать полученные знания для того, чтобы выдавать себя за K.

Проблема установления подлинности может быть решена путём использования криптосистемой дополнительного открытого ключа. Таким же образом, как и ранее, сторона K владеет криптосистемой с открытым ключом $(\mathrm{D}_K, \mathrm{E}_K)$. Дополнительно к этому сторона B обладает другой криптосистемой с открытым ключом $(\mathrm{D}_B, \mathrm{E}_B)$.[22] Обе функции шифрования, E_K и E_B, являются открытыми – следовательно, они известны обеим сторонам, B и K. Функция дешифровки D_K является секретной функцией стороны K, а функция дешифровки D_B – стороны B.

Итак, сторона K подписывает электронный документ за следующие 3 этапа.

1. Сторона B выбирает случайное значение w, после чего посылает значение $\mathrm{E}_K(w)$ стороне K.
2. Сторона K проверяет равенство $w = \mathrm{D}_K(\mathrm{E}_K(w))$. После этого K получает значение $c = \mathrm{E}_B(\mathrm{D}_K(w))$ и посылает его стороне B.
3. B проверяет равенство

$$w = \mathrm{E}_K(\mathrm{D}_B(c)) = \mathrm{E}_K(\mathrm{D}_B(\mathrm{E}_B(\mathrm{D}_K(w)))).$$

Очевидно, что в случае прохождения всех 3 этапов сторона B убеждена в корректности сообщения стороны K: ведь K – это единственная сторона, которой известен секрет

[21] Таким образом, свойство (b′) похоже на (b) – поскольку оба этих требования ограничивают возможность подделки сообщения стороной B.

[22] В данном случае необходимо, чтобы обе криптосистемы были коммутативными, т. е. чтобы выполнялось следующее:

$$w = \mathrm{D}_K(\mathrm{E}_K(w)) = \mathrm{E}_K(\mathrm{D}_K(w)) \quad \text{и} \quad w = \mathrm{D}_B(\mathrm{E}_B(w)) = \mathrm{E}_B(\mathrm{D}_B(w)).$$

D_K, поэтому только эта сторона может вычислить значени w на основе $E_K(w)$, после чего сформировать сообщение $D_K(w)$ на основе вычисленного w.

Итак, сообщение $E_K(w)$ может быть расшифровано только стороной K, а сообщение $E_B(D_K(w))$ – только стороной B. Поэтому третья сторона (фальсификатор) не может изучить и проверить подпись $(w, E_B(D_K(w)))$, удовлетворяющую условию (a).

При такой коммуникации B изучает подпись $(w, E_B(D_K(w)))$ стороны K. В процессе нескольких выполнений этого протокола установления подлинности подписи сторона B может выучить несколько таких пар. Однако этого недостаточно для того, чтобы сторона B могла убедить некоторую третью сторону, что она (т. е. сама сторона B) является K. Если все стороны в коммуникационной сети используют этот протокол, то каждая третья сторона C посылает стороне K сообщение $E_K(u)$ – вычисленное для некоторого случайного слова u. Если B как активный противник[23] берёт это сообщение из коммуникационной сети, то, вообще говоря, B не может выучить u – поскольку B не знает секрета D_K стороны K. Единственная попытка, которую может сделать B – это вычислить $E_K(w)$ для всех уже известных ему пар $(w, E_B(D_K(w)))$ подписей стороны K, после чего сравнить эти подписи с $E_K(u)$. Если B достигнет в этом успеха, т. е. найдёт $E_K(w) = E_K(u)$ для одного из запомненных u, то B *может* убедить сторону C в том, что является стороной K – путём посылки стороне C соответствующей подписи $E_C(D_K(w))$. Но если u является случайным числом, состоящим из нескольких сотен цифр, – то вероятность успеха действий стороны B меньше величины, обратной числу протонов в известной нам части Вселенной. Более того, эта вероятность может быть последовательно уменьшена – путём последовательных изменений ключей в используемых нами криптосистемах.[24]

Упражнение 9.7. Рассмотрите проблему установления подлинности, где цель фальсификатора – убедить всех в том, что он является K, но не подписать документ. Прведённый выше протокол является для этой цели надёжным с достаточно большой вероятностью.

Внесите в этот протокол небольшие изменения – для того, чтобы вероятность попыток фальсификации стороной B была бы уменьшена до 0. При этом такая вероятность не должна зависеть от длины списка подписей стороны K, имеющихся у стороны B.

9.5 Доказательства с нулевым разглашением

В главе 6 мы узнали, что у любого языка класса NP есть полиномиально-временной верификатор. Это означает, что для любого языка $L \in \mathrm{NP}$ для всех утверждений $x \in L$ имеются доказательства, полиномиально зависящие от длины $|x|$ – причём корректность этих доказательств может быть проверена за полиномиальное время. Таким образом, языки класса NP относительно проверки доказательств являются простыми. В главе 8 мы утверждали, что практическая разрешимость должна быть значительно больше связана с рандомизированным полиномиальным временем – а не с детерминированным. Поэтому возникают следующий вопрос:

> *для каких языков можно «практически» проверять корректность доказательств?*

[23] Который хочет убедить C, что является K.

[24] Имеются и более сложные протоколы, предназначенные для проблемы установления подлинности, – но в этой книге мы не будем приводить их описание.

Для исследования этого вопроса рассмотрим следующую модель коммуникационного протокола.

Пусть имеются две стороны – **доказывающая** и **проверяющая**.[25] Первая из этих сторон является некоторым алгоритмом (некоторой МТ), имеющим неограниченные вычислительные возможности.[26] Вторая сторона – некоторый рандомизированый полиномиально-временной алгоритм (рандомизированная полиномиально-временна́я МТ). Пусть также L – некоторый язык.

В начале вычисления обе стороны получают один и тот же вход x. Доказывающий и проверяющий могут общаться путём обмена сообщениями, полиномиально зависящими от длины $|x|$. Доказывающий с помощью своих неограниченных вычислительных возможностей пытается убедить проверяющего в истинности утверждения $x \in L$. С этой целью доказывающему позволяется лгать (делать ложные заявления). Задача проверяющего заключается в том, чтобы поставить вопрос доказывающему — с целью вычислить с достаточно большой вероятностью, имеется ли у того доказательство утверждения $x \in L$. Число этапов вычисления[27] не более чем полиномиально (относительно $|x|$), и полное вычисление также выполняется проверяющим за полиномиальное время. Проверяющий должен закончить вычисления, приняв решение о том, действительно ли $x \in L$. Этот коммуникационный протокол между доказывающим и проверяющим называется интерактивной рандомизированной системой проверки доказательств.

Определение 9.8. *Пусть $L \subseteq \Sigma^*$ для некоторого алфавита Σ. Будем говорить, что L обладает* **интерактивной системой доказательств**[28]*, если существует проверяющая сторона (некоторый рандомизированый полиномиально-временной алгоритм) V, такая что для всех $x \in \Sigma^*$ выполняются следующие условия.*[29]

- *Если $x \in L$, то существует доказывающая сторона B, такая что V после коммуникации с B решает принять x с вероятностью, большей чем $2/3$.*
  {Немного подробнее. Если $x \in L$, то существует доказывающая сторона, обладающая доказательством этого факта. При этом V может эффективно проверить такое доказательство в процессе комуникации с B. И не имеет значения, насколько длинным является это доказательство.}
- *Если $x \notin L$, то для любой доказывающей стороны B коммуникация между V и B заканчивается тем, что V отклоняет вход x с вероятностью по крайней мере $2/3$.*
  {Если $x \notin L$, то не существует доказательства того, что $x \in L$. Поэтому каждая стратегия убедить V в истинности утверждения $x \in L$ должна быть обнаружена с достаточно высокой вероятностью.}

---

[25] По-английски проверяющая сторона – «verifier», что совпадает с термином, переведённым ранее (раздел 6.5) как «верификатор». Однако соответствующие понятия различны. *(Прим. перев.)*

[26] Т. е. стороной без каких-либо ограничений на сложность вычислений.

[27] В данном случае – число коммуникационных обменов.

[28] Как в русской, так и в английской литературе иногда применяется и более точное название – интерактивная рандомизированная система проверки доказательств. *(Прим. перев.)*

[29] Всюду в данном определении имеется в виду коммуникации между доказывающей и проверяющей сторонами, в которой первая пытается убедить вторую в истинности утверждения $x \in L$ для рассматриваемого слова $x \in \Sigma^*$. *(Прим. перев.)*

Определим класс языков IP *следующим образом:*

$$\mathbf{IP} = \{L \subseteq \Sigma^* \mid L \textit{ обладает интерактивной системой доказательства}\}.$$

Понятно, что точное значение границ вероятности в определении 9.8 не принципиально. После $O(|x|)$ независимых повторений коммуникации между доказывающей и проверяющей сторонами вероятность ошибки может быть уменьшена до $2^{-|x|}$. Итак, в класссе IP можно повысить вероятность принятия правильного решения с $2/3$ до $1 - 2^{-|x|}$.

Лемма 9.9. NP $\subseteq$ IP.

Доказательство. Поскольку NP = VP, существует польномиально-временной верификатор для каждого языка $L \in$ NP. Это означает, что для любого языка L и любого слова $x \in L$ существует доказательство (свидетельство) c того, что $x \in L$, – причём длина доказательства c является полиномиальной относительно $|x|$. Следовательно, доказыающая сторона, обладающая c, может послать всё это слово проверяющей стороне – после чего последняя может за полиномиальное время детерминированно проверить, действительно ли c является доказательством $x \in L$. Если же $x \notin L$ то доказательства того, что $x \in L$, не существует – поэтому не существует и возможности детерминированно проверить этот факт. □

Следующий вопрос – существуют ли языки, не принадлежащие классу NP, но при этом обладающие интерактивной системой доказательств. Рассмотрим следующую проблему. Для двух заданных графов G_1 и G_2 необходимо ответить на вопрос, являются ли они изоморфными.[30] Проблема изоморфизма графов принадлежит классу NP – поскольку мы можем недетерминированно предположить изоморфизм (т. е. недетерминированно предположить конкретный вариант соответствия вершин) – а затем детерминированно проверить наше предположение за полиномиальное время. Однако неизвестно, принадлежит ли классу NP проблема *не*изоморфизма графов. Мы предполагаем, что она не принадлежит классу NP – поскольку не видим, как недетерминированное угадывание может помочь проверить отсутствие изоморфизма двух графов.

Итак, обозначим

$$\text{NONISO} = \{(G_1, G_2) \mid G_1 \text{ и } G_2 \text{ не изоморфны}\}.$$

Опишем интерактивную систему доказательства для проблемы NONISO. Пусть (G_1, G_2) – входы, известные сторонам V и B.

1. Проверяющая сторона V определяет, действительно ли у графов G_1 и G_2 совпадает количество вершин и рёбер. Если не совпадает, то сторона V отклоняет вход с вероятностью 1.
2. Если у графов G_1 и G_2 совпадает число вершин и рёбер, то сторона V выбирает некоторые случайное $i \in \{1, 2\}$ и случайную перестановку $(j_1, j_2, \ldots, j_n)$ чисел $(1, 2, \ldots, n)$,[31] где n – число вершин. После этого V переводит граф G_i в

[30] Два графа являются изоморфными, если мы можем пометить вершины одного из них пометками вершин другого – так, что оба графа при этом становятся идентичными.

[31] Как i, так и перестановка выбираются с помощью равномерного распределения вероятностей. Другими словами – все возможные варианты выбора, как и всюду в данной главе, считаются равновероятными.

$G_i(j_1, j_2, \ldots, j_n)$ – соответственно этой перестановке. Далее сторона V посылает граф $G_i(j_1, j_2, \ldots, j_n)$ проверяющей стороне B и просит её определить, какому именно графу изоморфен граф $G_i(j_1, j_2, \ldots, j_n)$ – G_1 или G_2.

3. Если G_1 и G_2 *не* изоморфны, то B может определить i (т. е. вычислить, какой из двух графов, G_1 или G_2, изоморфен графу $G_i(j_1, j_2, \ldots, j_n)$) – и послать номер i стороне V.
   Если же G_1 и G_2 изоморфны, то сторона B *не* имеет возможности вычислить такое i. B может только предположить i – таким образом B приходится выбирать случайное $k \in \{1, 2\}$ и посылать это значение стороне V.
4. Если $k \neq i$, то V отклоняет вход (G_1, G_2).
   Если $k = i$, то V повторяет этап 2, посылая некоторые случайные $s \in \{1, 2\}$ и граф-перестановку G_s стороне B – после чего B посылает $l \in \{1, 2\}$ стороне V соответственно этапу 3.
   Теперь если $l \neq s$, то V отклоняет вход (G_1, G_2).
   Если $k = s$, то V принимает вход (G_1, G_2).

Покажем, что приведённый здесь протокол является интерактивной системой доказательства для проблемы NONISO.

Если $(G_1, G_2) \in$ NONISO, то у доказывающей стороны существует возможность определить разницу между графами G_1 и G_2 – и, следовательно, эта сторона всегда выдаёт правильный ответ. Поэтому проверяющая сторона принимает вход (G_1, G_2) с вероятностью 1.

Если же $(G_1, G_2) \notin$ NONISO, то не существует возможности определить разницу между графами G_1 и G_2 относительно гомоморфизма. Вероятность того, что доказывающая сторона корректно угадала два случайно выбранных значения i и s из множества $\{1, 2\}$, не превосходит значения $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$. Таким образом, вероятность ошибки не превосходит $1/4$. При этом мы можем уменьшить её до 2^{-k} – путём постановки k независимых вопросов, заданных проверяющей стороной доказывающей.

Следующий результат показывает огромные возможности рандомизированной проверки.

Теорема 9.10 (Шамира).* IP = PSPACE. □

Самое важное следствие этой теоремы заключается в том, что наиболее короткое доказательство утверждения $x \in L$ для некоторого языка $L \in$ PSPACE может иметь экспоненциальную длину относительно значения $|x|$.[32] Следовательно, доказывающая сторона не может послать проверяющей это доказательство – даже хотя бы некоторую его существенную часть. Таким образом мы можем с большой вероятностью проверять корректность доказательств экспоненциальной длины в пределах рандомизированного полиномиального времени – без чтения этих доказательств.

Закончим наше элементарное введение в криптографию кратким описанием рандомизированных **систем (проверки доказательств) с нулевым разглашением**. Опустим формальное определение этих систем. Основная идея заключается в том, чтобы добавить в доказывающую систему дополнительное требование: проверяющая

[32] Выполнение условия $L \in$ PSPACE «без теоремы Шамира» говорит только о том, что все доказательства утверждений $x \in L$ имеют полиномиальную ширину. Здесь под шириной доказательства понимается длина наиболее длинного предположения, появившегося в последовательности импликаций этого доказательства.

сторона при коммуникациях с доказывающей не может узнать ни одного бита – кроме той информации, которую она может вычислить сама, без коммуникаций. Такое «отсутствие знания» во время коммуникации может быть формализовано следующим образом. Полная коммуникация между B и V моделируется как некоторое вероятностное пространство – где распределение вероятностей определяется случайным выбором, осуществляемым стороной V. Интерактивная система доказательства называется системой с нулевым разглашением, если существует рандомизированная полиномиально-временна́я машина Тьюринга M, которая генерирует всю соответствующую коммуникацию с тем же самым распределением вероятностей. Следствием этого требования является то, что в подобных системах[33] проверяющая сторона не изучает ни одного бита в доказательствах, представленных другой стороной.

У систем проверки с нулевым разглашением имеется много интересных криптографических приложений – опишем одно из них. Пусть нам надо разработать управление доступом для компьютера, позволяющего работать только пользователям с корректным паролем – и в то же время пользователи должны оставаться анонимными. Это означает, что пользователь играет роль доказывающей стороны, которая должна убедить систему управления доступом (проверяющую сторону) в том, что она (доказывающая сторона) действительно обладает правомерным паролем для доступа к компьютеру.

Другим важным приложением систем проверки с нулевым разглашением является использование при коммуникации собственных данных другой стороны – причём без изучения этих данных. Предположим, что проверяющая сторона V стремится вычислить значение $f(x, y)$ – при этом обладая только значением y. Доказывающая сторона B знает значение x, и при этом пытается помочь стороне V – при условии, что V не знает ни одного бита слова x.

Определение класса функций и проблем принадлежности, для которых имеются системы проверки с нулевым разглашением – многообещающая исследовательская проблема. Таким путём может быть решено очень большое количество задач – и этот факт на самом деле является удивительным. Например, мы знаем, что системы проверки с нулевым разглашением существуют для всех языков класса NP. В связи с ограничениями на объём нашей книги мы не приводим доказательства этого результата – но в качестве примера приведём описание подобной системы для проблемы изоморфизма графов.

Вход: Пара графов (G_1, G_2) для B и V. Пусть n – число вершин графов G_1 и G_2.

B: Доказывающая сторона случайно выбирает некоторые $i \in \{1, 2\}$ и перестановку $\pi = (j_1, j_2, \ldots, j_n)$ чисел $(1, 2, \ldots, n)$. После этого B применяет π к G_i и посылает результирующий граф $G_i(\pi)$ проверяющей стороне.

V: Проверяющая сторона случайно выбирает некоторое $j \in \{1, 2\}$ и посылает его доказывающей стороне. Это сообщение j соответствует требованию представления доказательства гомоморфизма графов $G_i(\pi)$ и G_j.

B: Если G_1 и G_2 изоморфны, то доказывающая сторона вычисляет изоморфизм δ, такой что $G_j(\delta) = G_i(\pi)$ – после чего посылает δ проверяющей стороне.
Если G_1 и G_2 не являются изоморфными, и в то же время $i = j$, то доказывающая сторона посылает случайную перестановку $\delta = \pi$. Иначе (т. е. если $i \neq j$ и между графами G_j и $G_i(\pi)$ отсутствует изоморфизм) доказывающая сторона также посылает перестановку π.

[33] Можно также говорить – в протоколах, основанных на нулевом разглашении.

V: Доказывающая сторона принимает (G_1, G_2) если и только если $G_j(\delta) = G_i(\pi)$.

Теперь покажем, что этот протокол является интерактивной системой доказательств. Если G_1 и G_2 являются изоморфными, то доказывающая сторона всегда находит перестановку δ, такую что $G_j(\delta) = G_i(\pi)$. Таким образом, сторона V принимает вход (G_1, G_2) с вероятностью 1. Если же G_1 и G_2 не являются изоморфными, то только при $i = j$ сторона B может послать перестановку δ, такую что $G_j(\delta) = G_i(\pi)$. Вероятность того, что $i = j$ для двух случайных чисел $i, j \in \{1, 2\}$, в точности равна $1/2$. Путём выполнения k независимых повторений этого протокола и принятия входа только в том случае, когда все ответы были корректными, мы получаем итеративную систему проверки с вероятностью ошибки не более чем 2^{-k}.

Поскольку мы опустили формальное определение систем проверки с нулевым разглашением, мы не можем дать и формального доказательства того, что наш протокол соответствует такой системе – однако мы можем интуитивно понять, почему это так. Коммуникация между B и V не обеспечивает нас информацией об изоморфизме между графами G_1 и G_2 – если даже таковой имеется. Первое сообщение $G_i(\pi)$ может быть рассмотрено как случайное событие, которое определяется случайным выбором i и π. Второе сообщение – случайное число j. Последнее сообщение δ также определяется случайной перестановкой π: δ является либо в точности перестановкой π, либо комбинацией π и гомоморфизма между графами G_1 и G_2.[34] Следовательно, для некоторой фиксированной пары графов (G_1, G_2) множество всех вариантов коммуникации $(G_i(\pi), j, \delta)$ между B и V является вероятностным пространством. Мы можем показать, что существует некоторая рандомизированная полиномиально-временна́я машина Тьюринга, которая может генерировать тройки $(G_i(\pi), j, \delta)$ с тем же самым распределением вероятностей.

9.6 Проектирование объединённых сетей

Коммуникации между различными субъектами – такими как люди, компьютеры, процессоры параллельного компьютера и мн. др. – могут быть выполнены с помощью различных вариантов коммуникационных технологий. У каждого из таких вариантов есть свои возможности и свои ограничения – и, следовательно, для работы с телеграфными, телефонными и оптическими сетями и параллельными компьютерами мы должны решать различные задачи. Эти задачи весьма сложны – что проявляется прежде всего при проектировании коммуникационных стратегий в таких сетях; более того, их сложность столь велика, что мы не имеем возможности описать несколько таких задач в одном разделе. Поэтому мы проиллюстрируем проблемы сетевых коммуникаций только на одном примере.

Итак, рассмотрим следующую задачу. Пусть имеется $2n$ **сторон**

$$x_0, x_1, \ldots, x_{n-1}, y_0, y_1, \ldots, y_{n-1},$$

которые представлены как $2n$ узлов[35] сети. Мы должны построить сеть между x-сторонами $x_0, x_1, \ldots, x_{n-1}$ и y-сторонами $y_0, y_1, \ldots, y_{n-1}$ – таким образом, чтобы в любой момент времени любая x-сторона могла получить связь (соединение) для вызова

[34] Который мы также описываем как некоторую перестановку.

[35] Когда мы говорим о коммуникационных сетях, то предпочитаем использовать термин «узел» – вместо термина «вершина» (графа).

любой y-стороны. Сеть рассматривается как граф

$$G = (V, E),\ x_0, x_1, \ldots, x_{n-1}, y_0, y_1, \ldots, y_{n-1} \in V.$$

Соединение узлов x_i и y_j означает поиск *и резервирование* пути между x_i и y_j в графе G – чтобы ни одно из рёбер этого пути не использовалось для некоторой другой коммуникации, т. е. между какой-нибудь другой парой узлов. Иными словами – в любой момент времени любое ребро графа G может быть использовано только для одного вызова.

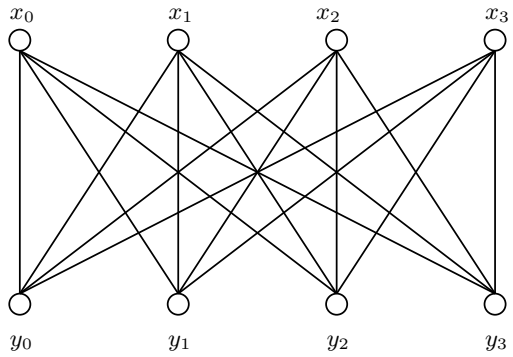


Рис. 9.2.

Простейшее решение этой проблемы заключается в проведении ребра из каждого x-узла (x-стороны) в каждый y-узел. Таким образом мы получаем полный двудольный граф – вроде изображённого на рис. 9.2 для $n = 4$. Но это решение нельзя применить на практике. Во-первых, в случае $2n$ сторон мы получаем n^2 рёбер. Для $n = 10\,000$ это означает, что число связей (физических соединений) равно $100\,000\,000$. Кроме того, в связи с высокой стоимостью изготовления такой сети и большим числом её связей мы для достаточно больших n просто не можем создать подобную сеть. Существуют и иные технологические причины: число связей узла (его степень) должна быть ограничена некоторой константой – т. е. степень сети[36] с увеличением числа сторон не должна возрастать.

Сформулируем требования для нашей коммуникационной проблемы, необходимые для практических приложений.

- Степень каждого узла сети не превосходит 4. При этом степень узлов, соответствующих x- и y-сторонам, не превосходит 2.
- Узлы сети, которые не соответствуют сторонам, называются **узлами-переключателями**. Структура всех узлов-переключателей одна и та же – см. рис. 9.3. У каждого переключателя v имеется в точности 4 инцидентных ребра: $LO(v)$, $LU(v)$, $RO(v)$ и $RU(v)$. При этом он может быть в одном из двух (переключаемых) состояний: $LLRR$ или $LRRL$.

[36] Она определяется как максимум среди степеней узлов, входящих в сеть.

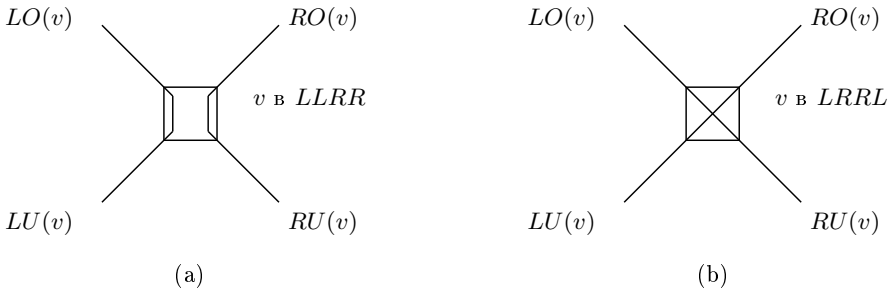


Рис. 9.3.

- Если v находится в состоянии $LLRR$ (рис. 9.3 (a)), то мы полагаем, что в данной вершине v зафиксированы соединения между рёбрами $LO(v)$ и $LU(v)$, а также между рёбрами $RO(v)$ и $RU(v)$. Иными словами – рёбра $LO(v)$ и $LU(v)$ должны быть частью какого-нибудь одного коммуникационного пути, а рёбра $RO(v)$ и $RU(v)$ – какого-нибудь другого.
- А если v находится в состоянии $LRRL$ (рис. 9.3 (b)), то в v зафиксированы соединения между $LO(v)$ и $RU(v)$, а также между $RO(v)$ и $LU(v)$. То есть рёбра $LO(v)$ и $RU(v)$ должны быть частью какого-нибудь одного коммуникационного пути, а рёбра $RO(v)$ и $LU(v)$ – какого-нибудь другого.

• Если некоторый узел u является x-стороной, то имеется только два ребра, инцидентных u – это $L(u)$ и $R(u)$. Следовательно, число состояний узла u также равно 2, это L и R – см. рис. 9.4.[37]

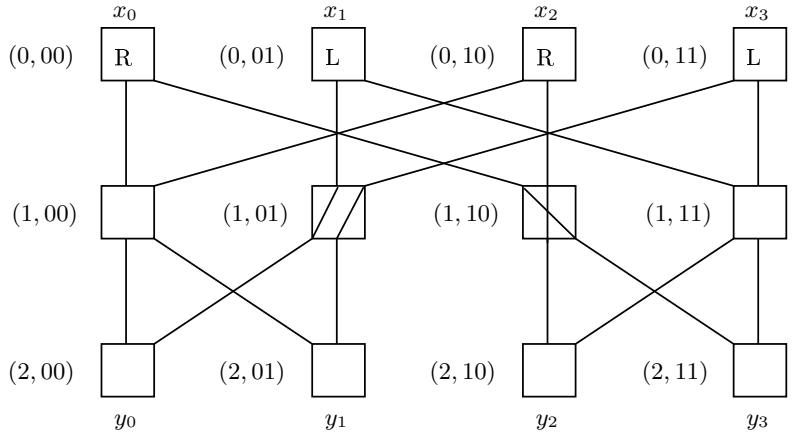


Рис. 9.4.

[37] Отметим, что определять состояния для y-сторон не нужно: конкретные варианты их соединения определяются на основе состояний остальных узлов.

- Решаемая **коммуникационная задача** формулируется как некоторая перестановка

$$(i_0, i_1, \ldots, i_{n-1})$$

чисел $(0, 1, \ldots, n-1)$. Это означает, что для каждого $j \in \{0, 1, \ldots, n-1\}$ нужно установить связь x-стороны x_j с y-стороной y_{i_j}. Сеть должна быть такой, чтобы для любой из $n!$ возможных перестановок $(i_0, i_1, \ldots, i_{n-1})$ существовало некоторое соответствие узлов сети и определённых выше состояний[38] – такое, что одновременно могут быть выполнены все n требований связи, т. е. требования на установление связей между парами сторон

$$(x_0, y_{i_0}), (x_1, y_{i_1}), \ldots, (x_{n-1}, y_{i_{n-1}}).$$

При этом нужно зафиксировать (зарезервировать) n рёберно-независимых путей[39] – между x_j и y_{i_j} для каждого $j = 0, 1, \ldots, n-1$.

Сеть с $2n$ сторонами

$$x_0, x_1, \ldots, x_{n-1},\ y_0, y_1, \ldots, y_{n-1},$$

которая может выполнить *все* коммуникационные задачи, формулируемые перестановками чисел $(0, 1, \ldots, n-1)$, называется ***n*-перестановочной сетью**.[40] Стоимостью перестановочной сети является число узлов-переключателей[41] – и очевидно, что мы должны стараться минимизировать эту стоимость. Другой параметр, который также желательно минимизировать, – это длина наиболее длинного пути между какой-либо парой x- и y-сторон. Было бы также очень хорошо создать регулярную (чёткую и ясную) структуру разрабатываемой сети. Особое значение могла бы иметь её модульность – это означает, что мы должны иметь возможность использовать сеть с $2n$ сторонами в качестве основного компонента для разработки сетей, содержащих большее количество сторон, например, $4n$. Очевидно, что при будущем проектировании расширенной сети модульность уменьшает её стоимость – именно поэтому она очень важна.

Сеть, приведённая на рис. 9.4, является возможным решением для 8 сторон (и, следовательно, является 4-перестановочной сетью) – и при этом имеет малую стоимость. Число узлов-переключателей равно 4, при этом коммуникационный путь между любой парой x- и y-сторон имеет одну и ту же длину 2. Конкретные состояния узлов сети, приведённой на рис. 9.4, определяют решение коммуникацинной задачи для заданной перестановки $(3, 0, 2, 1)$.

Упражнение 9.11. Докажите, что сеть, приведённая на рис. 9.4, действительно является 4-перестановочной сетью.

---

[38] Т. е. состояний L и R для x-сторон, $LLRR$ и $LRRL$ для переключателей.

[39] Некоторое множество путей называется рёберно-независимым (edge-disjoint), если у этих путей нет ни одного общего ребра. *(Прим. перев.)*

[40] В качестве её частного случая можно рассматривать, например, т. н. телефонную сеть. Неформально она может быть получена путём объединения каждой пары вершин x_i и y_i в одну. В телефонной сети любая сторона в любое время имеет возможность устанавливать связь с любой другой стороной.

[41] Заметим, что не имеет значения, рассматриваем ли мы в качестве описательной сложности число переключателей или число рёбер.

Теперь нашей целью является разработка для рассматриваемой проблемы связи асимптотически оптимальной сети. В первую очередь мы покажем, что у каждой сети с $2n$ сторонами имеется по крайней мере $\Omega(n \log n)$ узлов-переключателей.

Теорема 9.12. *У любой n-перестановочной сети ($n \in \mathbb{N}$) имеется по крайней мере $\Omega(n \log n)$ узлов-переключателей.*

Доказательство. Применим обычный пересчёт. Пусть Net_n – некоторая n-перестановочная сеть, и пусть также задана некоторая перестановка, определяющая коммуникационную задачу. Для решения этой задачи мы должны найти соответствующее присваивание[42] узлам сети Net_n некоторых состояний.[43]

Пусть m – число узлов-переключателей в сети Net_n. Отсюда число различных присваиваний состояниям сети Net_n в точности равно

$$2^n \cdot 2^m.$$

Поэтому мы получаем необходимость выполнения следующего неравенства:

$$2^m \cdot 2^n \geq n! \quad - \text{ т. е. } \quad 2^m \geq \frac{n!}{2^n}.$$

Отсюда[44]

$$m \geq \log_2(n!) - n \geq n \cdot \log n - n \cdot (\ln e + 1) \in \Omega(n \log n).$$

□

Упражнение 9.13. Докажите, что каждая n-перестановочная сеть содержит по крайней мере один такой путь между некоторыми x- и y-сторонами, длина которого не менее $\log_2 n$.

Итак, основная задача этого раздела – разработка n-перестановочной сети, имеющей для любого натурального числа n размер в пределах $O(n \log n)$. Мы сначала решим более простую задачу – разработаем т. н. ***r*-мерный *β*-элемент**. Его определение практически совпадает с определением перестановочной сети – однако при этом *не* накладывается требование рёберной независимости выбираемых путей.

Определим частный случай r-мерного β-элемента – т. н. r-мерную бабочку (r-dimensional butterfly) как $But_r = (V_r, E_r)$, где

$$\begin{aligned} V_r &= \{(i,w) \mid i \in \{0,1,\ldots,r\},\ w \in \{0,1\}^r\}, \\ E_r &= \{\{(i,w),(i+1,w)\} \mid i \in \{0,1,\ldots,r-1\}\} \cup \\ &\quad \{\{(i,xay),(i+1,xby)\} \mid i \in \{0,1,\ldots,r-1\},\ x \in \{0,1\}^i, \\ &\qquad\qquad a,b \in \{0,1\},\ a \neq b,\ y \in \{0,1\}^{r-i-1}\}. \end{aligned}$$

Это определение является достаточным – однако для его понимания желательны некоторые комментарии. Например, 1-мерный β-элемент But_1 показан на рис. 9.5.

[42] Будем говорить именно так – это «более программистский» термин, чем «соответствие». Более того, с математической точки зрения соответствие надо в данном случае строго определить. В оригинале – «assignments», мн. ч. *(Прим. перев.)*

[43] Очевидно, что различные перестановки приводят к различным вариантам такого присваивания. Таким образом, общее число всех возможных присваиваний состояниям сети Net_n должно быть не менее $n!$ – это является числом всех перестановок n элементов, и, следовательно, различных коммуникационных задач.

[44] Согласно формуле Стирлинга, $n! \approx \frac{n^n}{e^n} \cdot \sqrt{2\pi n}$.

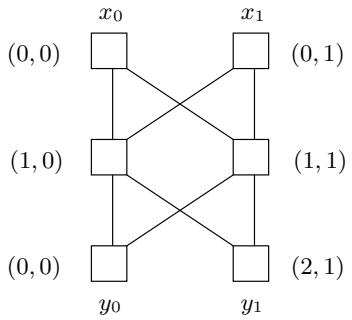


Рис. 9.5.

А для бóльших r представление требуемого β-элемента But_r даёт запись его $(r+1)\cdot 2^r$ узлов в виде матрицы, содержащей $r+1$ строк и 2^r столбцов.[45] Для каждого $j = 0, 1, \ldots, 2^r - 1$ мы считаем, что:

- x-сторона x_j соответствует узлу $(0, w)$, такому что $Number(w) = j$;
- y-сторона y_j – узлу (r, w'), такому что $Number(w') = j$.

Позиция (i, j) этой матрицы содержит вершину (i, w), такую что для w выполняется аналогичное условие: $Number(w) = j$.

Ребро между узлами (i, x) и $(i+1, y)$ присутствует в одном из следующих двух случаев:

- $x = y$;[46]
- единственное различие между x и y заключается в i-м бите их представления.[47]

Пример для $r = 2$ фактически уже был рассмотрен ранее: см. рис. 9.4.

При таком задании графа для каждой пары

$$(x_d, y_c), \quad d, c \in \{0, 1, \ldots, 2^r - 1\},$$

можно найти описываемый далее путь между x_d и y_c. Итак, пусть для некоторых $a_k, b_k \in \{0, 1\}$ $(k = 0, 1, \ldots, r-1)$

$$d = Number(a_0 a_1 \ldots a_{r-1}) \quad \text{и} \quad c = Number(b_0 b_1 \ldots b_{r-1}).$$

Следовательно, узел $(0, a_0 a_1 \ldots a_{r-1})$ соответствует x-стороне x_d, а узел $(r, b_0 b_1 \ldots b_{r-1})$ – y-стороне y_c.

Если $a_0 = b_0$, то путь начинается с «вертикального» ребра

$$\{(0, a_0 a_1 \ldots a_{r-1}), (1, a_0 a_1 \ldots a_{r-1})\}.$$

Если же $a_0 \neq b_0$, то путь начинается с «наклонного» ребра

[45] Важно отметить, что строки и столбцы нумеруются с 0. *(Прим. перев.)*

[46] Это – «вертикальные» рёбра, которые идут в каждом столбце от первой строки до последней.

[47] То есть в 0-м бите для пары узлов в 0-й и 1-й строках, в 1-м бите для пары в 1-й и 2-й строках, и т.д. *(Прим. перев.)*

$$\{(0, a_0a_1 \dots a_{r-1}), (1, b_0a_1 \dots a_{r-1})\}.$$

Итак, мы в обоих случаях за 1 шаг (т. е. после выбора 1 ребра) достигаем узла

$$(1, b_0a_1 \ \dots \ a_{r-1}).$$

А в общем случае для пути из $(0, a_1 \dots a_{r-1})$ в $(r, b_0b_1 \dots b_{r-1})$ мы таким же образом выбираем k рёбер и достигаем узла

$$(k, b_0b_1 \dots b_{k-1}a_ka_{k+1}a_{k+2} \dots a_{r-1})$$

После этого этот путь продолжается $(k{+}1)$-м ребром

$$\{(k, b_0b_1 \dots b_{k-1}a_ka_{k+1} \dots a_{r-1}), (k+1, b_0b_1 \dots b_ka_{k+1} \dots a_{r-1})\}.$$

При этом мы можем сказать, что s-е ребро построенного пути «отвечает за передачу значения» s-го бита a_s биту b_s.

Итак, мы показали, что в сети But_r можно соединить каждую пару x- и y-сторон путём, имеющим длину r. Но, к сожалению, в сети But_r мы не в состоянии реализовать каждую возможную перестановку, выбирая один из нескольких *рёберно-независимых* путей; показать это проще всего следующим образом. При применении нашей стратегии узел

$$(\lfloor \tfrac{r}{2} \rfloor, b_0b_1 \dots b_{\lfloor r/2 \rfloor}a_{\lfloor r/2 \rfloor+1} \dots a_{r-1})$$

используется во всех путях, начинающихся в одном из узлов множества

$$\{(0, f_0f_1 \dots f_{\lfloor r/2 \rfloor}a_{\lfloor r/2 \rfloor+1} \dots a_{r-1} \mid f_i \in \{0,1\} \text{ для } i = 0, 1, \dots, \lfloor \tfrac{r}{2} \rfloor\}$$

и заканчивающихся в одном из узлов множества

$$\{(r, b_0b_1 \dots b_{\lfloor r/2 \rfloor}e_{\lfloor r/2 \rfloor+1} \dots e_{r-1}) \mid e_j \in \{0,1\} \text{ для } j = \lfloor \tfrac{r}{2} \rfloor + 1, \dots, r-1\}.$$

Всего имеется $2^{\frac{r}{2}-1}$ таких путей – а мы можем использовать каждый из узлов-переключателей не более чем для 2 таких путей.

Таким образом, сети But_r *не* являются перестановочными. Но их элементы удаётся использовать в качестве компонентов при проектировании перестановочных сетей – подробнее см. ниже. Далее в этом разделе будем рассматривать перестановочные сети, а среди них – т. н. **сети Бенеша**. Приведём два простых примера. Сеть $Benes_1$ совпадает с сетью But_1 – см. рис. 9.5. А на рис. 9.6 показана сеть $Benes_3$.

Рассмотрим общий случай. Сначала определим *некоторые* 2^r-перестановочные сети, узлы которых можно расположить в $2r+1$ строках и 2^r столбцах; уже определённые нами сети But_1 и But_3 являются подобными примерами. Первые $r+1$ строк этих сетей вместе с соответствующими рёбрами формируют один r-мерный β-элемент, а последние $r+1$ строк – другой r-мерный β-элемент.[48] Отметим, что приведённой формулировки фактически достаточно для построения *некоторой* 2^r-перестановочной сети – однако очевидно, что для практического применения она неудобна.

[48] Они, вообще говоря, могут не являться сетями But_r. И, конечно, всю сеть $Benes_r$ тоже можно рассматривать как некоторый r-мерный β-элемент.

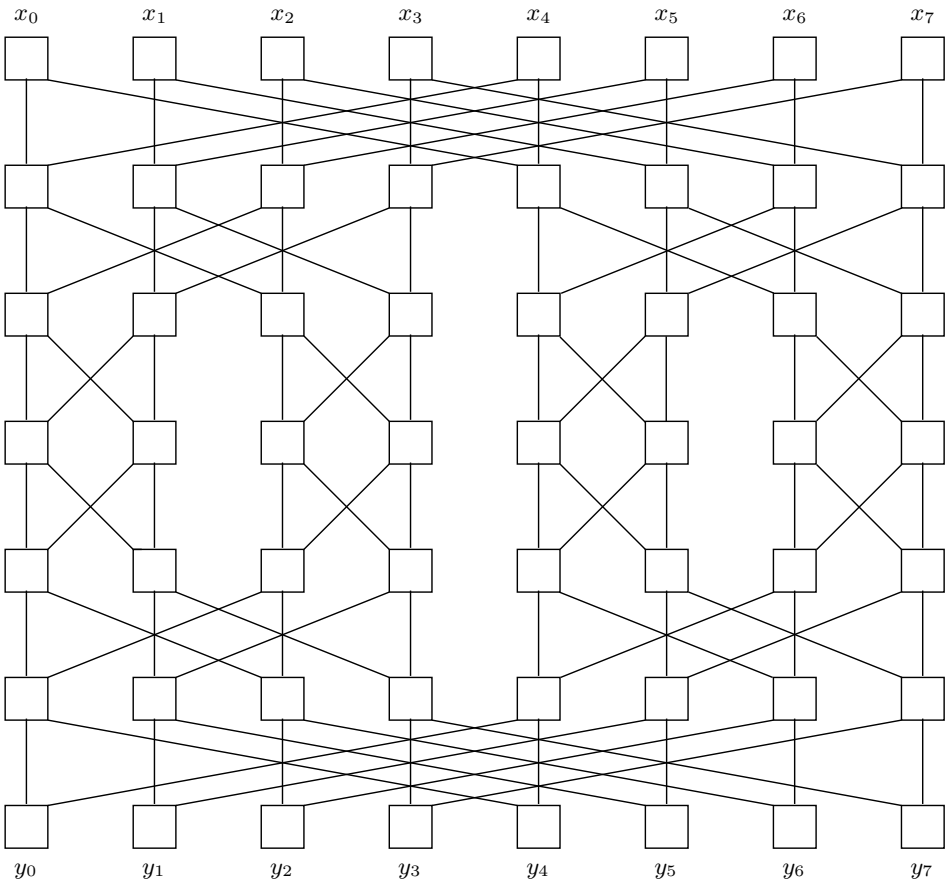


Рис. 9.6.

Продолжим формулировку определения сети $Benes_r$ – для произвольного $r \geq 2$. Нам значительно более важен другой вариант выделения в этой сети подсетей – вариант, отличающийся от описанного в предыдущем абзаце. На рис. 9.7 показано рекурсивное определение сети $Benes_r$ – путём её построения из двух сетей $Benes_{r-1}$ и некоторых дополнительных узлов и дуг. Итак, первые 2^{r-1} столбцов этой сети являются первой сетью $Benes_{r-1}$, обозначенной A, а последние 2^{r-1} столбцов – второй сетью $Benes_{r-1}$, обозначенной B.

Упражнение 9.14. Для каждого натурального r приведите формальное описание сети $Benes_r$ как графа.

Теорема 9.15. *Для каждого натурального r сеть $Benes_r$ является 2^r-перестановочной сетью.*

Доказательство. Покажем, что для каждой перестановки

$$(i_0, \ldots, i_{2^r-1}),$$

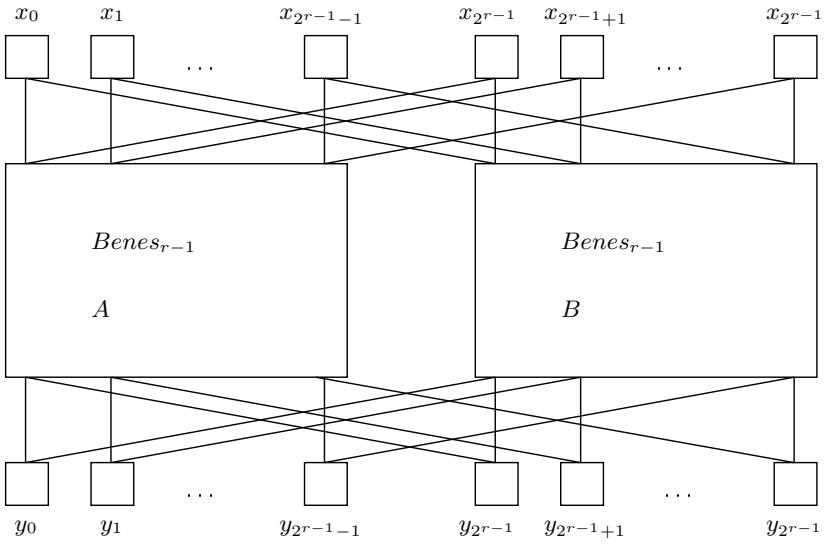


Рис. 9.7.

можно выбрать 2^r путей из x_j в y_{i_j} (т. е. по одному пути для каждого $j = 0, 1, \ldots, 2^r - 1$) – таким образом, что ни один из узлов не используется более чем для одного пути, и при этом каждый узел принадлежит в точности одному такому пути; будем называть каждое из таких множеств попарно-разделяемыми путями.[49] Очевидно, что такое множество путей будет гарантировать возможность присваивания узлам состояний, в точности соответствующего этим 2^r путям – что даст решение коммуникационной задачи $(i_0, \ldots, i_{2^r-1})$.

Итак, докажем существование требуемого множества путей по индукции относительно r. Для $r = 1$ утверждение очевидно: $Benes_1$ является 2-перестановочной сетью, см. рис. 9.5.

Теперь пусть $r \geq 2$. Согласно предположению индукции, в сети $Benes_{r-1}$ для каждой перестановки чисел $(0, 1, \ldots, 2^{r-1} - 1)$ существует 2^{r-1} попарно-разделяемых путей между 2^{r-1} x-сторонами и 2^{r-1} y-сторонами. Теперь рассмотрим $Benes_r$ – как сеть, приведённую на рис. 9.7; она состоит из двух сетей $Benes_{r-1}$ (на рисунке – A и B), а также двух строк (первой и последней), взятых из β-элементов But_r, вместе с соответствующими рёбрами.

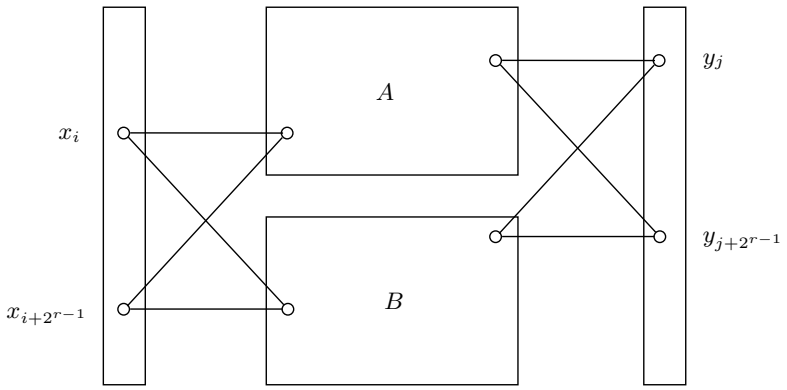
Согласно предположению индукции, нам достаточно найти такую коммуникационную стратегию, что:

- в точности половина путей проходят через компоненту A;
- коммуникационные задачи для A и B могут быть решены с помощью двух перестановок чисел $(0, 1, \ldots, 2^{r-1} - 1)$.

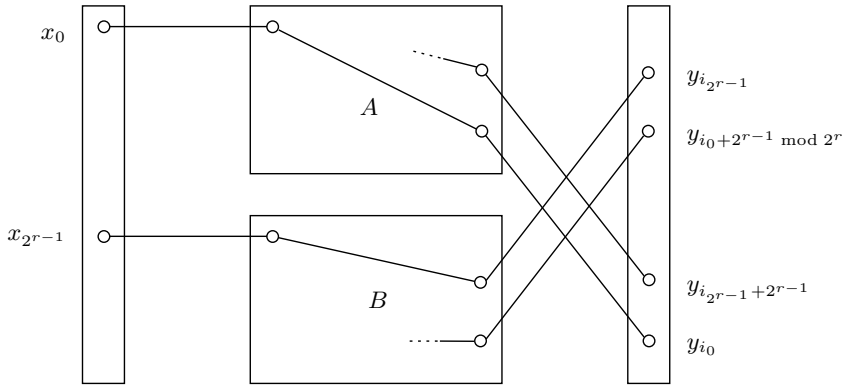
[49] Это название, чаще всего применяемое в литературе на русском языке, по-видимому, неудачно. Встречается также навание «попарно-независимые вершинно-разделяемые пути». В оригинале – «pair-wise node-disjoint paths».

Отметим, что каждое из таких множеств является рёберно-независимым. *(Прим. перев.)*

Для выполнения обоих этих требований достаточно таким образом выбрать дуги для первой строки $Benes_r$, чтобы ни один из узлов v первых строк элементов A и B (т. е. второй строки $Benes_r$) не принадлежал двум различным путям.[50] Аналогично, двум различным путям не должен принадлежать ни один узел u последних строк A и B.[51] Формально мы можем выразить сформулированные условия следующим образом.



(a)



(b)

Рис. 9.8.

(1) Для всех $i \in \{0, 1, \ldots, 2^{r-1}\}$ два пути, начинающиеся в x-сторонах

$$x_i \quad \text{и} \quad x_{i+2^{r-1}},$$

[50] Т. е. можно использовать только одно ребро между v и первой строкой сети $Benes_r$.

[51] Аналогично предыдущему, можно использовать только одно ребро между u и последней строкой сети $Benes_r$.

должны использовать различные сети $Benes_{r-1}$ – т. е. один из этих путей должен проходить через A, а другой – через B.

(2) Для всех $j \in \{0, 1, \ldots, 2^{r-1}\}$ два пути, заканчивающиеся в y-сторонах

$$y_j \quad \text{и} \quad y_{j+2^{r-1}},$$

также должны использовать различные сети $Benes_{r-1}$.

Отметим, что мы можем в обоих сформулированных условиях рассматривать одни и те же два пути. При этом оба условия отражены на рис. 9.8 (a).

Теперь покажем, что один из этих путей может быть определён на основе другого. Пусть

$$(i_0, i_1, \ldots, i_{2^r-1})$$

– произвольная перестановка чисел $(0, 1, \ldots, 2^r - 1)$. Рассмотрим пары

$$(x_0, y_{i_0}) \quad \text{и} \quad (x_{2^{r-1}}, y_{i_{2^{r-1}}}).$$

Для них определим путь из x_0 в y_{i_0} через A, а путь из $x_{2^{r-1}}$ в $y_{i_{2^{r-1}}}$ – через B, см. рис. 9.8 (b). В более простом случае, т. е. если

$$|i_{2^{r-1}} - i_0| = 2^{r-1},$$

обоим условиям (1) и (2) удовлетворяют именно эти пути.

Если же

$$|i_{2^{r-1}} - i_0| \neq 2^{r-1}$$

то, согласно (2), мы должны провести путь к y-стороне

$$y_{i_0+2^{r-1} \bmod 2^r}$$

через B, а путь к y-стороне

$$y_{i_{2^{r-1}}+2^{r-1} \bmod 2^r}$$

– через A. При этом[52] выполняются необходимые требования на взаимное расположение двух путей, выходящих из узлов

$$x_{q+2^{r-1} \bmod 2^r} \quad \text{и} \quad x_{s+2^{r-1} \bmod 2^r}$$

для

$$i_q = (i_{2^{r-1}} + 2^{r-1}) \bmod 2^r \quad \text{и} \quad i_s = (i_0 + 2^{r-1}) \bmod 2^r.$$

При дальнейшем продолжении этой стратегии каждый выбранный вариант расположения двух следующих дуг одного из путей может формулировать не более чем одно ограничение для другого пути рассматриваемой пары – и это ограничение может быть легко выполнено. Итак, предложенная стратегия даёт прострoение множества попарно-разделяемых путей. □

Упражнение 9.16. Приведите пути в сети $Benes_3$, соответствующие заданным перестановкам $(7, 2, 1, 3, 0, 5, 6, 4)$, $(0, 1, 2, 3, 7, 6, 5, 4)$ и $(0, 7, 1, 6, 2, 5, 3, 4)$.

[52] Т. е. если $|((q + 2^{r-1}) \bmod 2^r) - ((s + 2^{r-1}) \bmod 2^r)| \neq 2^{r-1}$.

Теорема 9.15 даёт решение нашей проблемы разработки n-перестановочной сети: $Benes_r$ является 2^r-перестановочной сетью

$$\text{с } (2r+1)\cdot 2^r \text{ узлами и } r\cdot 2^{r+2} \text{ дугами.}$$

Итак, если число x-сторон равно $n = 2^r$, то число узлов-переключателей ограничено $O(n\cdot\log_2 n)$. Более того, сеть Бенеша имеет регулярную структуру с высокой степенью модульности – см. рис. 9.7. Расстояние между любой x-стороной и любой y-стороной в точности равно $2r$ – и поэтому в зависимости от числа сторон растёт только логарифмически.

Упражнение 9.17.* Пусть $G = (V, E)$ – некоторый граф. **Сбалансированный разрез** графа G определяется как пара (V_1, V_2), такая что:

- $V = V_1 \cup V_2,\ \ V_1 \cap V_2 = \emptyset$;
- $-1 \le |V_1| - |V_2| \le 1$.

Стоимостью разреза (V_1, V_2) является значение

$$cost(V_1, V_2) = |E \cap \{\{v, u\} \mid v \in V_1, u \in V_2\}|$$

– т. е. число дуг между V_1 и V_2. **Разделённым весом** (bisection width) **графа G** назовём минимум стоимостей всех сбалансированных разрезов этого графа. При проектировании объединённых сетей мы обычно стараемся максимизировать значение разделённого веса – из относительно большого его значения следует, что любой разрез сети на две примерно равные части даёт достаточно много коммуникационных каналов между ними.

Докажите, что разделённый вес сетей But_r и $Benes_r$ находится в пределах $\Omega(2^r)$.[53]

9.7 Заключение

Криптография необходима для разработки криптосистем, предназначенных для секретного обмена информацией через открытые коммуникационные каналы. У классических (симметрических) криптосистем как механизм шифрования, как так и механизм дешифровки определяются с помощью ключа – являющегося общим секретом отправителя и получателя. У криптосистем с открытым ключом такого секрета нет – поскольку знание ключа кодирования не поможет найти ключ дешифровки. Идея разработки систем с открытым ключом основана на концепции односторонних функций. Каждая из них эффективно вычислима – в отличие от соответствующей обратной функции; последняя не является эффективно вычислимой без специальных дополнительных знаний – секрета, известного только получателю. Нам пока неизвестно, существуют ли вообще односторонние функции; кандидатами являются умножение (обратной функцией при этом является разложение на множители) и вычисление значения $a^b \pmod n$ (обратная функция – дискретный логарифм).

[53] Разделённый вес этих сетей в пределе примерно равен числу сторон этих сетей. Можно подсчитать этот вес и другим способом – в зависимости от числа *узлов* этих сетей (пусть m); в этом случае разделённый вес находится в пределах $\Omega(\frac{m}{\log m})$.

Интерактивные системы проверки доказательств, имеющиеся у всех языков класса PSPACE, эффективно осуществляют эту проверку рандомизированными алгоритмами. Системы проверки доказательств с нулевым разглашением, имеющиеся у всех языков класса NP, используются для проверки того, обладает ли другая сторона некоторым секретным доказательством – не выясняя значения ни одного бита этого секрета.

Качество сетей передачи данных измеряется числом удовлетворённых коммуникационных запросов (или количеством переданных данных) за некоторый достаточно короткий временной интервал. Разработка сети, оптимальной или почти оптимальной отнсительно некоторых качественных параметров, обычно приводит к нетривиальным оптимизационным задачам. Мы пытаемся решать их методами дискретной математики.

Настоящую революцию произвела концепция криптосистем с открытым ключом – она была предложена в 1976 году Диффи и Хелманом в [17]. Наиболее известная соответствующая криптосистема RSA была разработана Ривестом, Шамиром и Адлеманом в [58]. Концепция интерактивных систем доказательств была описана Голдвассером, Микали и Ракофом в [21]. Шамир в [64] первым доказал равенство IP = PSPACE.

Сети Бенеша были предложены Бенешем в работах [4, 5], посвящённых разработке телефонных сетей. Сети Бенеша являются перестановочными сетями – этот факт был доказан Ваксманом в [70].

В качестве ведения в криптографию мы настоятельно рекомендуем книгу Саломаа [61]. Другой отличный современный учебник написан Делфсом и Кнеблом – [16]. Хорошие обзоры интерактивных систем даны Боветом и Крешензи в [7] и Сипсером в [65]. Книга Лейтона [41] – современный учебник-монография по проектированию сетей и коммуникационным проблемам. Введение в коммуникационные проблемы для объединённых сетей приведено Громковичем, Класингом, Монином и Пейне в [33].

Список литературы

1. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation. Combinatorial Optimization Problems and their Approximability Properties.* Springer-Verlag, 1999.
2. J.L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I.* Springer-Verlag, 1988.
3. J.L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity II.* Springer-Verlag, 1990.
4. V. Beneš. Permutation groups, complexes and rearrangable multistage connecting networks. *Bell System Technical Journal*, 43:1619–1640, 1964.
5. V. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic.* Academic Press, 1965.
6. F. Bock. An algorithm for solving "traveling-salesman" and related network optimization problems: abstract. *Bulletin 14th National Meeting of the Operations Research Society of America*, page 897, 1958.
7. D.P. Bovet and P. Crescenzi. *Introduction to the Theory of Complexity.* Prentice-Hall, 1994.
8. V. Černý. A thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–55, 1985.
9. G.J. Chaitin. On the length of programs for computing finite binary sequences. *Journal of the ACM*, 13:407–412, 1966.
10. G.J. Chaitin. On the simplicity and speed of programs for computing definite sets of natural numbers. *Journal of the ACM*, 16:407–412, 1969.
11. G.J. Chaitin. Information-theoretic limitations of formal systems. *Journal of the ACM*, 13:403–424, 1974.
12. A. Church. An undecidable problem in elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.
13. S. Cook. The complexity of theorem-proving procedures. In *Proceedings of 3rd ACM STOC*, pages 151–157, 1971.
14. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 1990.
15. G.A. Croes. A method for solving traveling salesman problem. *Operations Research*, 6:791–812, 1958.
16. H. Delfs and H. Knebl. *Introduction to Cryptography. Priciples and Applications.* Springer-Verlag, 2002.
17. W. Diffie and M. Hellman. New directions in cryptographie. *IEEE Transitions Inform. Theory*, 26:644–654, 1976.
18. K. Erk and L. Priese. *Theoretische Informatik (Eine umfassende Einführung).* Springer-Verlag, 2000.
19. M. Garey and D. Johnson. *Computers and Intractability.* Freeman, 1979.

20. K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.
21. S. Goldwasser, S. Micali, and C. Rackoff. Knowledge complexity of interactive proofs. In *Proc. 17th ACM Symp. on Theory of Computation*, pages 291–304. ACM, 1985.
22. R. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.
23. D. Harel. *Algorithmics. The Spirit of Computing.* Addison Wesley, 1993.
24. J. Hartmanis, R. Stearns, and P. Lewis. Hierarchies of memory limited computations. In *Proceedings of 6th IEEE Symp. on Switching Circuit Theory and Logical Design*, pages 179–190, 1965.
25. J. Hartmanis and R.E. Stearns. On the computational complexity of algorithms. *Transactions of ASM*, 117:285–306, 1965.
26. D.S. Hochbaum. *Approximation Algorithms for NP-hard Problems.* PWS Publishing Company, Boston, 1997.
27. J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 2001.
28. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, 1979.
29. J. Hromkovič. *Communication Complexity and Parallel Computing.* Springer-Verlag, 1997.
30. J. Hromkovič. *Algorithms for Hard Problems. Introduction to Combinatorial Optimization, Randomization, Approximation and Heuristics.* Springer-Verlag, 2001.
31. J. Hromkovič. Communication protocols: An exemplary study of the power of randomness. In J. Reif J. Rolim P. Pavdalos, S. Rajasekaran, editor, *Handbook of Randomized Computing*, volume 2, pages 533–596. Kluwer Academic Publishers, 2001.
32. J. Hromkovič. *Algorithmics for Hard Problems.* Springer-Verlag, 2003.
33. J. Hromkovič, R. Klasing, B. Monien, and R. Peine. Dissemination of information in interconnection networks. In *Combinational Network Theory*, pages 125–212, 1996.
34. O.H. Ibarra and C.E. Kim. Fast approximation algorithms for the knapsack and sum of subsets problem. *Journal of the ACM*, 21:294–303, 1974.
35. R. Karp. Reducibility among combinatorial problems. In R. Miller, editor, *Complexity of Computer Computation*, pages 85–104. Plenum Press, 1972.
36. R.M. Karp. An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34:165–201, 1991.
37. S. Kirkpatrick, P.D. Gellat, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
38. S.C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112:727–742, 1936.
39. A.N. Kolmogorov. Three approaches for defining the concept of information quantity. *Probl. Information Transmission*, 1:1–7, 1965.
40. A.N. Kolmogorov. Logical basis for information theory and probability theory. *IEEE Transactions on Information Theory*, 14:662–664, 1968.
41. F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publ. Inc., 1992.
42. H.R. Lewis and Ch. Papadimitriou. The efficiency of algorithms. *Scientific American*, 238(1), 1978.
43. M. Li and P.M.B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer-Verlag, 1993.
44. N. Saxona M. Agrawal, N. Kayal. Primes in p. Unpublished manuscript.
45. E.W. Mayr, H.J. Prömel, and A. Steger, editors. *Lectures on Proof Verification and Approximation Algorithms.* Number 1967 in Lecture Notes in Computer Science. Springer-Verlag, 1998.

46. N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculation by fast computing machines. *Journal of Chemical Physics*, 21:1087–1091, 1953.
47. G. Miller. Rieman's hypothesis and test for primality. *Journal of Computer and System Sciences*, (13):300–317, 1976.
48. R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
49. Ch.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
50. Ch.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.
51. E. Post. Finite combinatory process-formulation. *Journal of Symbolic Logic*, 1:103–105, 1936.
52. E. Post. A variant of a recursively unsolvable problem. *Transactions of ASM*, 52:264–268, 1946.
53. V. Strassen R. Solovay. A fast monte carlo test for primality. *SIAM Journal on Computing 6*, pages 84–85, 1977.
54. M.O. Rabin. Probabilistic algorithms. In J.F. Traub, editor, *Algorithms and Complexity: Recent Results and New Directions*, pages 21–39. Academic Press, 1976.
55. M.O. Rabin. Probabilistic algorithms for primality testing. *Journal of Number Theory*, (12):128–138, 1980.
56. R. Reischuk. *Einführung in die Komplexitätstheorie*. B.G. Teubner, 1990.
57. H.G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of ASM*, 89:25–59, 1953.
58. R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. Assoc. Comput. Mach.*, 21:120–12, 1978.
59. H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.
60. A. Salomaa. *Formal Languages*. Academic Press, 1973.
61. A. Salomaa. *Public-Key Cryptographie*. Springer-Verlag, 1996.
62. U. Schöning. *Perlen der Theoretischen Informatik*. BI-Wissenschaftsverlag, Mannheim, Leipzig, Wien, Zürich, 1995.
63. U. Schöning. *Algorithmik*. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2001.
64. A. Shamir. IP= PSPACE. *Journal of the ACM*, 39:869–877, 1992.
65. M. Sipser. *Introduction to the Theory of Computation*. PWS Publ. Comp., 1997.
66. L.J. Stockmeyer and A.K. Chandra. Intrinsically difficult problems. *Scientific American*, 240(5), 1979.
67. B.A. Trakhtenbrot. *Algorithms and Automatic Computing Machines*. D.C. Heath & Co., Boston, 1963.
68. A.M. Turing. On computable numbers with an application to the Entscheidungsproblem. In *Proceedings of London Mathematical Society*, volume 42 of *2*, pages 230–265, 1936.
69. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.
70. A. Waksman. A permutation network. *Journal of ACM*, 15:159–163, 1968.
71. I. Wegener. *Theoretische Informatik – eine algorithmische Einführung*. R.G. Teubner, Stuttgart, Leipzig, 1999.

Предметный указатель