Московский государственный технический университет

имени Н.Э. Баумана

Факультет «Информатики и систем управления»

Кафедра «Информационная безопасность»

Э.Н.Гордеев

ВВЕДЕНИЕ В ТЕОРИЮ СЛОЖНОСТИ АЛГОРИТМОВ.

Электронное учебное издание

Учебное пособие

по дисциплине «Математическая логика и теория алгоритмов».

Москва

УДК 519.7

Рецензенты: проф., д.ф.-м.н., Кузюрин Н.Н.

проф., д.ф.-м.н.,.

Гордеев Э.Н.

Введение в теорию сложности алгоритмов. Электронное учебное издание. - М.: МГТУ имени Н.Э. Баумана, 2012. 84 с.

Издание содержит конспект лекций по курсу «Математическая логика и теория алгоритмов», предусмотренного учебным планом МГТУ им. Н.Э.Баумана. Представлены формальные модели алгоритмов, рассмотрены различные подходы к понятию *сложность задачи*. Описаны наиболее известные классы сложности задач. Приведены примеры исследования сложности известных задач.

Для студентов факультета «Информатики и систем управления» МГТУ имени Н.Э. Баумана.

Рекомендовано учебно-методической комиссией НУК «Информатики и систем управления» МГТУ им. Н.Э. Баумана

Электронное учебное издание

Гордеев Эдуард Николаевич

ВВЕДЕНИЕ В ТЕОРИЮ СЛОЖНОСТИ АЛГОРИТМОВ.

Оглавление.

Оглавление

Оглавление.

Оглавление.

Оглавление.

1. Введение

1.1. Некоторые необходимые определения и понятия.

При сравнении скорости роста двух неотрицательных функций f(n) и g(n) удобно использовать следующие обозначения.

Говорят, что функция g(n) f(n)-ограничена, если g(n)≤f(n). Если f(n) - полином, то функция g(n) называется полиномиально ограниченной, а если экспонента – экспоненциально ограниченной.

Будем говорить, что f(n)= O(g(n)), если существуют такие положительные константы c и N, что f(n)≤c g(n)для всех n>N.

В этой же ситуации можно использовать и обозначение g(n)=Ω(f(n)). Например, справедливы соотношения $\log_2 n = O(n)$, $n = \Omega(\log_2 n)$, $n = O(2^n)$.

Будем говорить, что f(n)= o(g(n)), если $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Если f(n)= O(g(n) и O(f(n))= g(n) , то это будем обозначать через f(n)=Θg(n).

Для числа x целые числа [x] и]x[- это целые части числа с недостатком и с избытком.

Опр. Алфавит A – это конечный или бесконечный набор символов: $A=\{a_1,a_2,\ldots,a_n,\ldots\}$.

Опр. Слово – это конечный упорядоченный набор символов алфавита.

Выделяется специальное пустое слово Λ (слово, не содержащее символов).

Пусть A^* – множество всех слов в алфавите A.

Опр. Язык L – это подмножество A^*. ($L \subseteq A^*$).

Опр. Конкатенация $\alpha\ \beta$ слов α и β.

$$\begin{cases}\alpha = b_1, b_2, b_3, \ldots, b_k \\ \beta = j_1, j_2, j_3, \ldots, j_S\end{cases} \Rightarrow \alpha\beta = b_1 b_2 b_3 \ldots b_k j_1 j_2 j_3 \ldots j_S .$$

Число сочетаний (без повторений) из n элементов по k элементов обозначим через $C_n^k = \frac{n!}{k!(n-k)!}$.

Число сочетаний с повторениями из n элементов по k элементов обозначим через $V_n^k = \frac{(n+k-1)!}{k!(n-1)!}$.

Если не оговорено обратное, вместо log_2n будем писать *log* n.

Простой граф с множеством вершин V, |V|=n, и множеством ребер E, |E|=m, будем обозначать через G=(V,E). Ориентированность графа будет оговариваться отдельно.

Булевы переменные - это переменные, принимающие значения из множества $B=\{0,1\}$. Множество n-мерных векторов из нулей и единиц называется булевым кубом и обозначается $\boldsymbol{B}^n$. Очевидно, что число вершин этого куба равно 2^n.

Отображение $f: \boldsymbol{B}^n \to B$ называется булевой функцией. Число булевых функций от n переменных равно

$$C_f(n) = 2^{2^n}$$

Примеры известных булевых функций: конъюнкция (&), дизъюнкция ($\cup$), отрицание ($\neg$), импликация ($\to$), сложение по модулю «два» ($\oplus$), эквивалентность ($\approx$) и пр.

Табличный способ задания функции представляет собой таблицу размером 2^n**x**(n+1). В последнем столбце таблице записаны значения функции на каждом из 2^n возможных наборах значений переменных. Сами эти наборы записаны в первых n клетках каждой строки.

При работе с булевыми функциями мы иногда будем заменять значок логического умножения & значком ^, обычной точкой (произведением) или не писать его вовсе.

2. Задачи, алгоритмы

В любой математической дисциплине, которая претендует на наличие собственного взгляда на "математическую" проблематику, исходные понятия не определяются математическим языком. Обычно, речь идет об "интуитивных" объектах, сущность которых может обсуждаться за пределами математики.

В этом курсе мы тоже будем иметь дело с несколькими лишь интуитивно определяемыми понятиями. Важнейшие из них - *задача* и *алгоритм*. Сразу оговорим очевидное. Для описания базовых понятий раздела науки используются те же слова, которые применяются и на бытовом

Оглавление.

уровне. Но в науке они становятся *терминами* и приобретают особый смысл. В лекции иногда трудно избежать ситуации, когда одно и то же слово используется и в качестве термина, и в качестве обиходного. В тексте при подозрении на двусмысленное понимание для терминов будет использоваться курсив, а для обиходных – кавычки, а там, где двусмысленность при разумном понимании невозможна – обычный шрифт.

2.1. Задача

Начнем с обсуждения первого из упомянутых понятий – понятия *задача*. При описании базовых понятий или аксиоматики используется три способа: аналогия, пример и *расчленение* «менее элементарных» сущностей на совокупность «более элементарных. Пойдем и мы по этому пути.

Наш курс тесно связан с двумя математическими дисциплинами: математической логикой и дискретной математикой. Поэтому самая простая идентификация понятия может быть дана по аналогии. Вспомним то, что имелось в виду под *задачей* в этих дисциплинах. В нашем курсе мы будем иметь дело примерно с тем же самым.

Например, в [1] определение задачи вообще не дается, считается, что оно «интуитивно очевидно». В [2] под задачей понимается «некоторый общий вопрос, на который следует дать ответ». При этом «задача содержит несколько параметров или свободных переменных, конкретные значения которых не определены».

В [4] под *задачей* понимается «выбор наилучшей конфигурации или множества параметров для достижения некоторой цели». При этом *задачи* делятся на *непрерывные* и *комбинаторные*. «В непрерывных задачах обычно отыскивается множество действительных чисел или даже некоторая функция; в комбинаторных задачах – некоторый объект из конечного или возможно бесконечного счетного множества». Другими словами, задача оптимизации – это пара **(F,c)** , где **F** – произвольное множество, область допустимых точек, а – **c** функция стоимости, отображающая элементы **F** на множество действительных чисел. Требуется найти такую x* точку из **F**, на которой значение функции **c**(x*) обладает определенным свойством, например, минимально, максимально и пр.

Теперь займемся «*расчленением*» сущностей, приведенных выше. Задачи, рассматриваемые в нашем курсе, это, в подавляющем большинстве, задачи *комбинаторные* или *дискретные*. Уже само слово

Оглавление.

Гордеев Э.Н. «Введение в теорию сложности алгоритмов».

комбинаторный относится к упомянутому выше множеству **F**. Вернее, к способу его задания. Мы видим, что при таком задании с *задачей* обычно связывается два *объекта*: *множество параметров* и *структура связей* между ними. *Параметры* – «язык» для описания условий задачи. *Структура связей* содержит нечто, позволяющее описать **F** , а также сформулировать *вопрос* (см. [2]) или требование к **c**(x*) трактуемое как вопрос, свойство функционала и пр. (См. [4]).

В *теории сложности* термин *задача* сразу же разбивается на два термина: *индивидуальная задача* и *массовая задача*. Это связаны со способом задания параметров. *Массовая задача* предполагает описание множества параметров, а *индивидуальная задача* возникает, при фиксации значений этих параметров.

Понятие *формы* задачи, грубо говоря, возникает в связи с возможностью "незначительно" модифицировать вышеупомянутые форму вопроса или свойство **c**(x*).

Поясним сказанное на примерах. Но эти примеры будут носить не только иллюстративный характер. В их тексте мы дадим и разъясним многие общие (выходящие за рамки примера) понятия, которые в дальнейшем будут использоваться.

Многие из этих примеров будут неоднократно упоминаться в дальнейшем, поэтому для упрощения ссылок на них мы дадим приведенным ниже задачам краткие названия.

Пример 1. (Поиск максимального числа). Заданы N целых чисел: $a_1,\ldots a_N$. Требуется найти максимальное из них.

На примере этой простейшей задачи постараемся подробно проанализировать *условие* задачи.

Здесь параметрами являются числа: N и $a_1,\ldots,a_N$. Правильнее представлять их в виде объектов, имеющих числовые значения. Это означает, что количество *сущностей*, между которыми при формулировке и решении задачи будут устанавливаться связи, равно 2N+2. Одна сущность предназначена для описания мощности множества сравниваемых чисел, другая – представляет собой числовое значение этой мощности (оно равно N), N сущностей предназначено для идентификации сравниваемых объектов. Эта идентификация достигается путем индексации: 1,…,N. Остальные N: $a_1,\ldots a_N$ – это веса (значения) проиндексированных объектов. В дальнейшем для сущностей, принимающих числовые значения, будем использовать понятие *параметр*, а для остальных – *объект*. Они не равноправны: сначала должны быть заданы объекты, а затем уже – параметры. Это означает, что задание параметра возможно после задания объекта. Можно также задать все объекты

Оглавление.

и лишь часть параметров или задать часть объектов, а затем определить параметры, с ними связанные.

В нашем примере *связи* на множестве объектов строятся через их параметризацию с использованием отношения *сравнения чисел*. *Задание* (свойство **c**(x*))в данном случае состоит в нахождении объекта, значение которого максимально. Если всем этим объектам и параметрам присвоены конкретные *значения*, то получается *индивидуальная* задача. В ситуации, когда речь идет о произвольных значениях объектов, мы имеем дело с *массовой* задачей. Заметим, что роли параметра N и параметров $a_1,\ldots a_N$ неодинаковы, т.е. можно говорить о множествах индивидуальных задач, получаемых из массовой задачи путем фиксации параметра N.

Уже такая простейшая задача позволяет проиллюстрировать понятие *формы* задачи. В данном случае можно говорить о задаче в форме *оптимизации*. Подобная форма возникает тогда, когда речь идет об экстремуме некоторого функционала на множестве исходных параметров задачи.

Задача в *вычислительной* форме получается при следующей модификации вопроса: "Требуется найти *величину* максимального из данных чисел". То есть здесь не нужно указывать сам объект, а только его числовое значение. Казалось бы, это условие не очень важное, но для ряда задач трудности решения их в оптимизационной и вычислительной формах различаются весьма существенно.

Другой распространенный тип задач - задачи в *форме распознавания*. В нашем случае для подобных задач добавляется еще один параметр ***B***, а задание выглядит следующим образом. Требуется ответ на вопрос, существует ли среди объектов $a_1,\ldots a_N$ такой, значение которого не меньше, чем ***B***? При этом, вообще говоря, не требуется ни указывать сам объект, ни его значение.

В последнем случае мы видим, что речь идет не только о форме задания, но и изменении множества параметров. То есть задача уже другая, но "близкая" к исходной. Это соответствует интуитивному представлению о том, что «незначительное» изменение в формулировке условия может приводить к «незначительному» изменению самой задачи.

Что значит *незначительно*? Как при этом меняется трудность решения? Этими вопросами мы тоже будем заниматься в нашем курсе.

Пример 2. (Разложение на множители). Дано натуральное число N. Требуется разложить его на простые множители.

Оглавление.

Параметром здесь является само число (значение объекта). При его фиксировании возникает индивидуальная задача. Если говорить о форме задачи, то она явно не оптимизационная, да и вычислительной не является, хотя в широком смысле под *вычислением* в данной задаче можно понимать нахождение всех простых делителей с их кратностями. Близкая к данной задача в *форме распознавания* состоит в определении простоты числа (**задача о простоте числа**).

Пример 4. (Задача о гамильтоновом цикле). Дан простой неориентированный граф G с N вершинами. Требуется узнать, существует ли в нем гамильтонов цикл. (Гамильтоновым называется цикл в графе, который проходит через каждую вершину графа ровно по одному разу.)

Набор исходных параметров - это число N и множество ребер графа, задаваемых как пары разных чисел из множества 1,…,N. Понятие *гамильтонова цикла* задает связь на множестве параметров.

Задача представлена в форме распознавания. Вычислительную и оптимизационную форму здесь искать малоинтересно. Но возникает еще одна форма - *перечислительная*. В этой задаче требуется найти число различных гамильтоновых циклов графа.

В дальнейшем будем, как правило, называть эту задачу просто: "**гамильтонов цикл**".

Пример 5. (Задача о коммивояжере.) Дан граф G с N вершинами, ребрам которого приписаны веса c_{ij}. Требуется найти гамильтонов цикл экстремальной длины. (Длина гамильтонова цикла может задаваться по-разному. В большинстве случаев – это сумма длин входящих в цикл ребер. Если не оговаривается обратное, то именно эта ситуация и имеется в виду. Но за длину можно брать и максимальный (минимальный) вес входящего в него ребра, и более сложный функционал от длин ребер. Наиболее распространенный случай, когда под экстремумом понимается минимум. Если же экстремум трактуется как максимум, то это оговаривается отдельно. Такие задачи называются задачами коммивояжера на максимум).

Можно считать, что граф полный (отсутствующим ребрам приписать специальным образом выбранные «большие» значения). Тогда набор исходных параметров определяется числом N и матрицей весов ребер $C=(c_{ij})$. Индивидуальная задача получается из массовой фиксированием этих параметров.

В дальнейшем будем называть эту задачу просто "**коммивояжер**".

Оглавление.

В формулировку задания входит понятие гамильтонова цикла, которое и задает связь на множестве параметров.

В вышеупомянутом варианте мы имеем задачу в форме оптимизации. Вычислительный вариант задачи состоит, например, в нахождении длины минимального гамильтонова цикла. В случае задачи на минимум в варианте распознавания задается еще число B и требуется ответить на вопрос, существует ли гамильтонов циклы с длиной, не превосходящей B.

Можно получить еще один вариант - *перечислительный*. Он состоит в нахождении количества минимальных гамильтоновых циклов.

Приближенный вариант без оценки точности состоит в нахождении гамильтонова цикла, который отличается от минимального "незначительно". При этом в понятие «незначительно» никакого формального смысла не вкладывается.

Приближенный вариант с оценкой точности может выглядеть следующим образом. Требуется для заданного числа k найти гамильтонов цикл, превосходящий минимальный не более чем в k раз.

Пример 6. (Задача о выполнимости КНФ.) Задана булевская функция F от N переменных в виде конъюнктивной нормальной формы K. Требуется определить, существует ли такой набор значений переменных, на котором функция обращается в единицу.

Здесь мы имеем задачу в форме распознавания. Исходными параметрами являются N и K. Что касается задания связей на множестве исходных параметров, то их можно проиллюстрировать следующим образом. Всех конъюнкций от N переменных 3^N. Обозначим это множество через Q(N). Индивидуальная задача получается из массовой путем фиксирования числа N и подмножества M из Q(N).

В дальнейшем будем называть эту задачу просто "**КНФ-выполнимость**".

Задача о клике ("**клика**") состоит в нахождении максимального (по числу вершин) полного подграфа K заданного графа G. *Задача о кратчайшем остове* ("**остов**") состоит в нахождении минимального (например, по сумме длин входящих ребер) остовного дерева T графа G, ребрам которого приписаны веса. Само название *Задача о кратчайшем пути* ("**кратчайший путь**") между двумя фиксированными вершинами графа уже все определяет. *Задача об изоморфизме графов* ("**изоморфизм графов**") состоит в определении изоморфизма пары заданных графов.

Оглавление.

2.2. Алгоритм

Если с понятием *задачи* все теперь более или менее понятно, то вот с понятием алгоритма дело обстоит сложнее. Работа в этом направлении породила не только теорию сложности или столь распространенные сейчас компьютеры, но и разделение математики, в которой на некоторое время остро стал вопрос о конструктивности. Но это тема отдельного большого разговора.

Итак, есть задача. Встает вопрос о ее решении. Какую бы форму задачи мы не взяли, решение задачи предполагает нахождение некоторого объекта. В случае индивидуальной задачи по исходному набору параметров (объектов и их значений) требуется найти некоторый *объект* (если речь идет об оптимизационной форме), *число* (если речь идет о вычислительной или перечислительной форме), *вариант* (один из двух: "да" или "нет", если речь идет о задаче в форме распознавания).

В жизни мы отнюдь не всегда для решения задач используем "алгоритмы". (Скорее, почти никогда их не используем). Действительно, мы можем найти объект, угадав его или используя наш опыт.

В этих случаях процесс нахождения состоит из двух важных этапов. Сначала в результате угадывания или опыта мы в качестве предполагаемого решения выбираем некоторый объект. Затем мы убеждаемся, что данный объект является требуемым решением задачи. Назовем первый этап этапом *предложения*, а второй - этапом *проверки*.

Если мы имеем задачу в форме распознавания, то нам не важна "логика" действий на первом этапе. Действительно, мы просто строго проверяем объект на наличие требуемого свойства. Например, в задаче о камнях мы считаем сумму весов предложенного подмножества объектов, в задаче о гамильтоновом цикле проверяем предложенное подмножество ребер на "гамильтоновость".

В том случае, когда *правильное* предложение возникает в результате отгадки, интуиции, опыта и пр., принято объединять все эти процессы, вводя понятие *оракула*. То есть *оракул*- это абстрактная сущность, которая мгновенно, без усилий способна предоставить предложение, не нуждающееся в проверке. Конечно, его можно проверить, но оракул на то и оракул, чтобы не ошибаться.

Для задач в форме оптимизации тоже может иметь место этап предложения, но этап проверки эквивалентен исходной задаче. В жизни мы можем его заменить верой или интуицией, но в математике требуется нечто иное. Требуется процедура, не зависящая от производящего ее субъекта,

которая бы абсолютно достоверно позволяла находить (проверять) требуемый в условии задачи объект.

Это и есть интуитивное представление алгоритма. В [2] например, оно звучит так. Алгоритм - это общая, выполняемая шаг за шагом процедура решения задачи. Прилагательное "общая" отвечает за автоматизм, независимость от субъекта, использующего алгоритм. Требование выполнения работы "шаг за шагом" здесь не ограничивает тип алгоритма (например, не отбрасывает параллельные или вероятностные алгоритмы), а лишь указывает на предсказуемость и проверяемость самой процедуры решения.

Во многих попытках дать определение алгоритма используются понятия "искусство" и "автоматизм" (машинное исполнение). При этом первое как раз должно отсутствовать в этом самом "алгоритме", а второе должно его характеризовать.

Формализация понятия "алгоритм" необходима также из-за "несимметричности" ответа на вопрос, существует ли алгоритм для решения задачи Z?

Действительно, ответ "да" может быть получен путем построения некоторой процедуры решения, которая укладывается в "интуитивное" понятие алгоритма. Такие процедуры и строились в процессе всей истории математики, когда формальным построениям самого понятия *алгоритм* внимания не уделялось. Если для некоторой задачи не могли построить метода решения, то это списывалось на недостаток умения.

Имея лишь интуитивное представление об алгоритме, нельзя доказать, что этого самого алгоритма не существует.

В 30-е годы стали появляться первые *формальные схемы* алгоритма. Эти схемы были предназначены исключительно для теоретических исследований. С некоторыми из них мы познакомимся здесь. Речь пойдет, например, о *машинах Тьюринга* (МТ), *нормальных алгорифмах Маркова* (НАМ) и др.

Алгоритм производит некоторые "действия" с объектами и параметрами, начиная с исходных условий задачи (входные условия, *вход*). Во всех известных формальных схемах этот *вход* как-то задается. В самом общем случае можно считать, что это задание выглядит в виде слова в некотором алфавите.

Оглавление.

Множество (конечное или бесконечное) символов назовем алфавитом *A*. Словом *P* в алфавите *A* назовем *любую* конечную последовательность символов алфавита.

Если множество символов одного алфавита является подмножеством символов другого, то первый называется подалфавитом второго. А второй - надалфавитом первого.

Пусть А - подалфавит некоторого алфавита В. Слово, состоящее из символов алфавита А, будем называть словом в алфавите А. Слово, состоящее из символов алфавита В, будем называть словом над алфавитом А.

Заметим, что достаточно ограничиться алфавитом из двух символов, например, 0 и 1. Любой символ a_i алфавита $A=\{a_1,\ldots,a_n,\ldots\}$ может быть закодирован следующим образом: a_i =011...10, где 1 повторяется n раз.

В известных *формальных схемах* (*формализмах*) "алгоритм" можно представить как некоторое пошаговое преобразование одних слова в другие, начиная с входа $x_1,\ldots,x_n = X$. В результате последнего шага таких преобразований получаем некоторое *выходное слово* Y. Его можно трактовать как результат работы алгоритма. Задавая буквы входного алфавита числовыми последовательностями, можно считать, что речь идет о вычислении функции $f(x_1,\ldots,x_n)=f(X)=Y$.

Обратим внимание на употребление понятия *эквивалентности* в связи с наличием нескольких возможных представлений (кодировок, способов записи) входных объектов.

Два выражения (кодировки) В и С считаются эквивалентными $B\approx C$ в двух случаях: либо оба выражения не определены (бессмысленны), либо определены и обозначают один и тот же объект.

Будем говорить, что алгоритм $\mathfrak{I}$ является алгоритмом в алфавите А, если он переводит слова $x_1,\ldots,x_n$ в алфавите А в слова $f(x_1,\ldots,x_n)$ в алфавите А.

Будем говорить, что алгоритм $\mathfrak{I}$ является алгоритмом над алфавитом А, если он переводит слова $x_1,\ldots,x_n$ над алфавитом А в слова $f(x_1,\ldots,x_n)$ над алфавитом А.

При этом алгоритм$\mathfrak{I}$ может работать не с любыми словами алфавита. Множество тех слов, к которым он применим, назовем областью действия алгоритма $\mathfrak{I}$, а результат применения алгоритма $\mathfrak{I}$ к слову Р обозначим через $\mathfrak{I}$ (Р). При этом сам алгоритм не обязан распознавать слова из своей области действия.

Оглавление.

Рассмотрим два алгоритма $\mathfrak{I}$ и $\mathfrak{R}$ над алфавитом A.

Если $\mathfrak{I}(P)\approx\mathfrak{R}(P)$ для любого слова P в алфавите A, то эти алгоритмы называются *вполне эквивалентными* относительно A.

Если $\mathfrak{I}(P)\approx\mathfrak{R}(P)$ для любого слова P в алфавите A такого, что хотя бы одно из выражений $\mathfrak{I}(P)$ или $\mathfrak{R}(P)$ определено и является словом в алфавите A, то эти алгоритмы называются *эквивалентными* относительно A.

Рассматриваемые ниже формальные подходы к понятию алгоритма заслужили внимание благодаря гипотезам типа *тезиса Черча*.

Этот тезис говорит об эквивалентности широкого неформально и смутного понятия "интуитивный алгоритм" узкому и весьма замысловатому на первый взгляд формализму типа *алгорифма Маркова* (МТ) или *машины Тьюринга*. Утверждается, что для любой задачи, для которой существует «интуитивный» алгоритм решения, можно, например, построить МТ, которая будет решать эту задачу.

Рассмотрим теперь данные формализмы подробно. Заметим, что алгоритм может предназначаться как для решения *массовой*, так и для решения *индивидуальной* задачи. (Например, алгоритм сложения двух чисел и алгоритм сложения 5 и 2). Конечно, мы изначально предполагаем, что алгоритм должен решать массовую задачу.

Сделаем одно замечание. Во всех перечисленных ниже формальных подходах алгоритм имеет дело с преобразованием одних слов в другие. То есть, условие задачи Z можно рассматривать как слово в некотором алфавите A. Обозначим все множество слов этого алфавита через A*. Подмножество слов алфавита назовем *языком*. Обратим внимание, что в содержательном смысле не все слова алфавита являются условиями индивидуальных задач из Z.

Особенно удобно ставить некоторый язык из A* в соответствие задачам в форме распознавания. В этих задачах в качестве решения возможно два варианта: 1 ("да") или 0 ("нет"). Таким образом, задаче Z в форме распознавания можно сопоставить язык L_Z, содержащий в качестве слов все слова из A*, которые являются условиями индивидуальных задач с ответом "да".

Оглавление.

3. Нормальные алгорифмы Маркова (НАМ).

Они были предложены в качестве формального подхода к понятию алгоритма выдающимся русским математиком А.А.Марковым в 1950-х годах.

Рассматривается некоторый алфавит А и слова в этом алфавите.

Конкатенацией двух слов Q и R назовем слово QR, то есть в конкатенации сначала идут последовательно все символы слова Q, а затем все символы слова R. (Не зная слов Q и R, мы воспринимаем слово QR как единое целое).

Любую подпоследовательность идущих друг за другом символов слова назовем подсловом.

Слово, не имеющее букв, называется пустым словом и обозначается через Λ.

Слова, состоящие из одинаковых последовательностей символов, считаются эквивалентными. Заметим, что слова PQ, ΛPQ, PΛQ, PQΛ эквивалентны в рамках формализма НАМ.

НАМ состоит из последовательности шагов работы. При определении шагов работы НАМ используются понятия *преобразования, обычного* и *завершающего*.

Обычным преобразованием слова P назовем переход от слова P к слову S. Завершающее преобразование состоит из обычного преобразования и команды остановки, которая обозначается точкой – (•). То есть после выполнения конечного преобразования работа заканчивается.

Преобразование π_i обозначается или в виде $P_i \to S_i$ (обычное), или в виде $P_i \to \bullet S_i$ (завершающее).

Формально НАМ - это четверка {A,B,Δ, П}, где A - входной алфавит ,B - надалфавит A, Δ - пустое слово, П - конечный упорядоченный набор преобразований $П=\{\pi_1,\ldots,\pi_n\}$. В каждом таком преобразовании присутствуют слова над алфавитом A (в алфавите B). Входное и выходное слова - это слова в алфавите A.

Если для слова P не существует ни одного преобразования такого, что P_i является его подсловом, то эту ситуацию обозначим следующим образом: $P \Leftarrow$. Если в результате применения НАМ к слову P образовалось слово R, то обозначим это через $P \Rightarrow R$.

НАМ преобразует входное слово P в выходное Q. Работа НАМ осуществляется шаг за шагом. На первом шаге в качестве входного слова используется слово P. На каждом следующем шаге в качестве входного слова используется результат предыдущего шага. На каждом шаге во входном слове поочередно для i=1,…,n ищутся крайние левые подслова, эквивалентные P_i . Если такое подслово найдено, то выполняется преобразование π_i. Если это преобразование завершающее, то работа НАМ заканчивается. В противном случае переход к следующему шагу. И так для всех i=1,…,n. Если ни для какого из этих i упомянутое выше подслово не найдено, то работа НАМ заканчивается.

Оглавление.

Задача для НАМ выглядит в виде требования построить НАМ, который преобразует данное входное слово (условие любой индивидуальной задачи данной массовой задачи) в выходное слово (решение индивидуальной задачи данной массовой задачи). Естественно, что может быть много различных НАМ, решающих каждую конкретную задачу.

Входной алфавит - это алфавит, в котором написано входное слово. Составление НАМ - это нахождение В и $П=\{\pi_1,\ldots,\pi_n\}$. Предполагается, что пустое слово не входит ни в один из рабочих алфавитов НАМ.

Перейдем теперь к рассмотрению конкретных примеров. Договоримся о некоторых обозначениях. Целые числа задаются в алфавите {1, *}. Целое число n кодируется последовательностью из n+1 единицы. Несколько аргументов функции разделяются символом *. Например, при вычислении значения функции F(x,y,z) от трех переменных x=2, y=5, z=1 входное слово НАМ имеет вид 111*111111*11.

Для сокращения записи схемы алгоритма буквами η, ξ, ζ будем обозначать любые буквы входного алфавита А, а буквами α, β, γ обозначаются конкретные буквы, не входящие в А. Например, имеем алфавит из 10 букв, а в схеме алгоритма идут подряд 10 преобразований стирания каждой одной буквы, которые могут выполняться в произвольном порядке. Все эти преобразования можно обозначать как $\eta\to\Delta$.

4. Машины Тьюринга

Согласно тезису Черча любые две формальные схемы, описывающие понятие алгоритма должны быть эквивалентны друг другу.

Здесь мы рассмотрим еще одну, пожалую самую популярную из таких схем - машину Тьюринга. Подобно предыдущему формализму она предназначается для преобразования некоторого одного слова Р в заданном алфавите в другое слово Q.

Заметим, что существует несколько разновидностей МТ. Более того, даже для одного и того же типа МТ определения могут варьироваться. Это может быть связано с мелкими техническими удобствами, а также с попытками не загромождать формализма "очевидными" тонкостями.

4.1. Однолентоочная МТ

Начнем с самого простого объекта - однолентоочной детерминированной МТ. Она представляет собой пятерку $\{S, A, F, q_0, \delta\}$, где $S=\{q_1,\ldots q_k\}$ - конечное множество, элементы которого называются состояниями, $A=\{a_1,\ldots,a_n\}$ - алфавит, $F\subseteq S$. Элементы F называются конечными состояниями,
Оглавление.

выделено начальное состояние q_0 и схема переходов машины Тьюринга δ. Каждый такой переход еще называется командой, а вся схема (список команд) представляет собой частично определенное отображение из $S\times A\to S\times A\times\{L, R, St\}$. Здесь {L, R, St } - множество из трех элементов, смысл которых ниже поясняется.

Опишем теперь механизм работы МТ. На двусторонней бесконечной ленте, разбитой на ячейке, начиная с некоторой ячейки, записано входное слово P (по одному символу в каждой ячейке). Имеется еще один объект - "головка" МТ, который характеризуется наличием определенного приписанного ей состояния и умением читать и писать.

Работа МТ состоит из последовательности тактов.

В начальный момент времени головка обозревает ячейку ленты, в которой записан первый символ входного слова, и находится в начальном состоянии.

Вообще, положение головки, ее состояние и запись на ленте полностью описывают *конфигурацию* МТ в данный момент времени. Поэтому конфигурацией МТ и называется слово $v_1\dots v_k q_j v_{k+1}\dots v_m$. Эта запись свидетельствует о том, что в данный момент времени головка МТ находится в состоянии q_j и обозревает ячейку ленты, в которой записан символ v_{k+1}. Все же слово, записанное к этому моменту на ленте, имеет вид $v_1\dots v_k v_{k+1}\dots v_m$.

Каждой конфигурации $K=v_1\dots v_k q_j v_{k+1}\dots v_m$ однозначно сопоставляется *пара конфигурации* $q_j v_{k+1}$ и *слово конфигурации* $v_1\dots v_k v_{k+1}\dots v_m$.

Такт работы представляет собой переход из одной конфигурации к другой. Этот переход осуществляется путем выполнения некоторой команды $q_i a_j \to q_r a_t\{Z\}$. (Пару $q_i a_j$ назовем левой частью команды $q_i a_j\to q_r a_t\{Z\}$, а тройку $q_r a_t\{Z\}$ - правой частью этой команды.) Если пара конфигурации не является левой частью ни одной из команд, то МТ останавливается и процесс ее работы заканчивается.

Оглавление.

Если же такая команда $q_i a_j \rightarrow q_r a_t \{Z\}$ найдена, то это означает, что перед выполнением данного такта МТ находилась в состоянии q_i, обозревала ячейку с символом a_j, а после выполнения данного такта она перешла в состояние q_r, в ранее обозреваемой ячейке появился символ a_t .Z имеет одно из трех значений L, R, St. В первом случае головка сдвинулась на данном такте на одну ячейку влево, во втором - вправо, в третьем - осталась на месте. Если q_r является конечным состоянием, то процесс работы МТ заканчивается и МТ останавливается. В противном случае для вновь полученной конфигурации ищется команда, правая часть которой совпадает с парой конфигурации, и процесс продолжается.

Так как каждая команда определяется однозначно, то, в отличие от НАМ, порядок команд в списке не имеет значения.

Переход от конфигурации K_i к конфигурации K_j в результате выполнения некоторого такта работы МТ обозначим через $K_i \vdash K_j$. Если существует такая последовательность команд, что в результате их выполнения МТ перешла из конфигурации K_i к конфигурации K_j, то эту ситуацию обозначим через $K_i \vdash\!\!\Rightarrow K_j$.

Заметим, что МТ может зацикливаться. Это довольно тонкий момент в данном формализме, который восходит к тому факту в определении алгоритма, согласно которому алгоритм не обязан сам распознавать слова из своей области действия. Конечно, если мы требуем, чтобы перед началом работы была проведена проверка на соответствие входного слова области действия, то никакого зацикливания не может быть, так как уже на стадии проверки будут отсеяны те слова, на которых алгоритм не может корректно работать (слова, которым он *не применим*). Если же такой предварительной стадии нет, то она может быть заменена предположением о том, что у нас есть некоторый механизм идентификации зацикливаний. То есть, мы можем различать три ситуации: МТ остановилась и завершила свою работу; МТ находится в процессе работы, совершив некоторой количество переходов, и готовится к очередному такту работы; МТ зациклилась, т.е. будет работать и никогда не остановится.

Если МТ на входном слове P за конечное число тактов останавливается, то говорят, что она *применима* к этому слову. Последовательность конфигураций от начальной до конечной называется *вычислением* МТ на слове P.

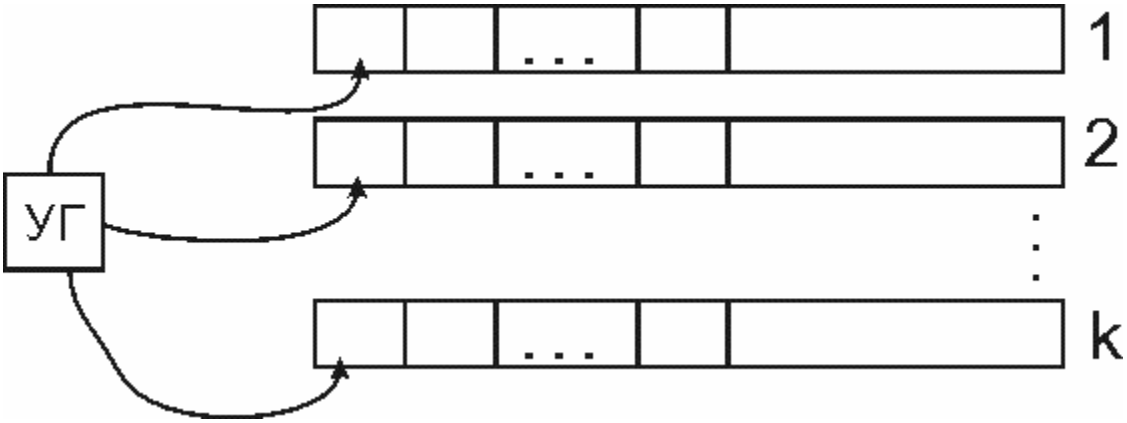
Оглавление.

В отличие от НАМ в данном формализме присутствуют такие понятия как: читающая и пишущая головка, обозреваемая ячейка, сдвиги вправо и влево и пр. Все это с точки зрения формальной схемы интуитивного понятия "алгоритм" также нуждается в формализации. Она более или менее очевидна, особенно с точки зрения людей, знакомых с азами компьютерной техники. (Заметим, что Тьюринг предложил свой формализм в 1935 году, когда о компьютерах еще и речи не было). Предположим, что все эти детали определены, и мы получили некоторый алгоритм $\mathfrak{R}_{T,C}$. Он работает в алфавите C, C является надалфавитом алфавита A машины Тьюринга T. A для любых слов P и Q в алфавите C соотношение $\mathfrak{R}_{T,C}(P)=Q$ выполняется тогда и только тогда, когда существует такое вычисление на машине Тьюринга T такое, что оно начинается с конфигурации q_0R и заканчивается конфигурацией $R_1q_iR_2$, где $R_1R_2=Q$.

Вообще же произвольный алгоритм $\mathfrak{I}$ в алфавите D называется *вычислимым по Тьюрингу* тогда и только тогда, когда существует машины Тьюринга T с алфавитом A и алфавит C, являющийся надалфавитом $A \cup D$ такие, что алгоритмы $\mathfrak{R}_{T,C}$ и $\mathfrak{I}$ вполне эквивалентны относительно D.

4.2. Многоленточная МТ

Простым обобщением МТ является k-ленточная МТ. Вместо одной ленты здесь имеется k лент, каждая из которых обслуживается своей головкой. Имеется управляющее устройство, которое и характеризуется определенным состоянием (в случае одноленточной МТ головка и это устройство объединялись в одном объекте). Конфигурация в этом случае состоит из записей на лентах с указанием мест головок и состояния управляющего устройства. Вместо пары конфигурации мы имеем (k+1)-ку из символа текущего состояния и символов в обозреваемых ячейках.



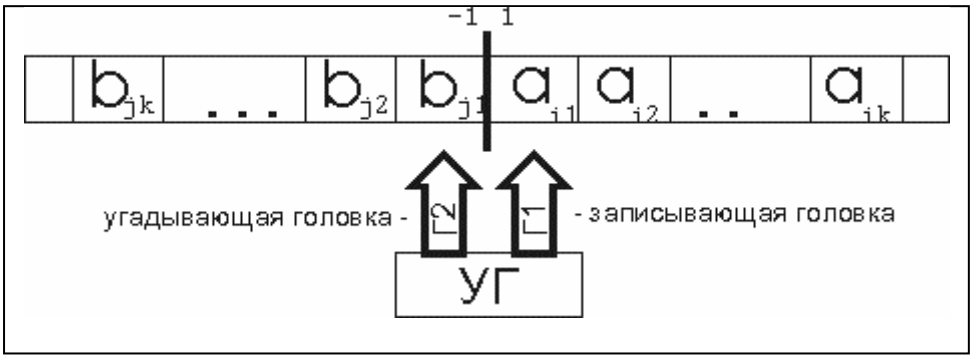
Оглавление.

Команды задаются уже отображением $S\times A^k \to S\times\{A\times\{L, R, St\}\}^k$.

В течение одного такта работы можно изменить состояние, напечатать в обозреваемых ячейках новые символы, если это требуется программой и сдвинуть независимо друг от друга какие-нибудь или все головки в тех направлениях, которые предусмотрены командой.

4.3. Недетерминированная МТ

Это совершенно иное обобщение понятия МТ. Рассмотрим случай одноленточной недетерминированной машины Тьюринга (НМТ).



Имеется две управляющие головки (УГ) и одна лента. Одна головка Г1 - обычная, такая же, как в МТ, а вторая Г2, которая часто называется угадывающей, может только записывать. Запись на ленте представляет собой слово (*условие задачи*), записанное слева направо, начиная с ячейки с номером 1. В начальный момент времени обычная головка наблюдает ячейку с номером 1, а пишущая головка - ячейку с номером (-1). Вначале работает только пишущая (угадывающая) головка. На каждом такте работы она может написать символ в наблюдаемой ячейке и сдвинуться влево, либо остановиться. В последнем случае начинает работать обычная головка с состояния q_0. С этого момента НМТ работает точно так же, как МТ с той лишь разницей, что наблюдает не самый крайний слева символ входного слова.

Оглавление.

Управляющее устройство пишущей головки принимает решение совершенно произвольно, поэтому может никогда не остановиться.

Ниже в разного рода определениях важна только *возможность* написания некоторого слова угадывающей головкой. Так как она пишет произвольно, то может написать любую букву в ячейке с номером -1, любую – в ячейке -2 и т.д. Если мы будем рассматривать **все** возможные результаты работы угадывающей головки, то это будут **все** возможные слова в данном алфавите.

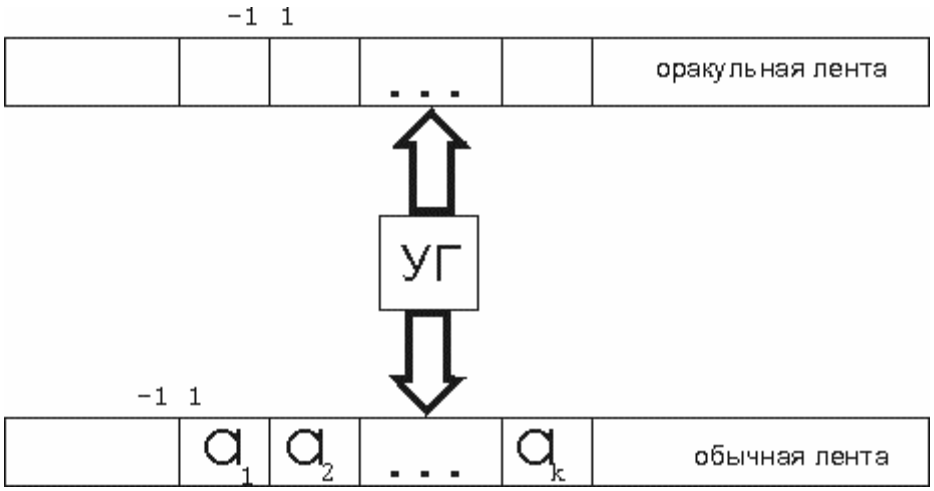
Далее мы будем различать входное слово I и слово U, записанное угадывающей головкой.

Разница между МТ и НМТ может быть проиллюстрирована с помощью сравнения модели обычных вычислений и моделью параллельных вычислений.

В качестве примеров можно рассмотреть задачи о выполнимости, о гамильтоновом цикле и о простоте числа.

4.4. Оракульная МТ

Как и в предыдущем случае здесь две головки (обычная и оракульная), но и две ленты (обычная и оракульная). Во множестве состояний выделены два особых: состояние вопроса к оракулу q_c и резюмирующее состояние q_r. Обе головки управляются одним управляющим устройством. В начальный момент времени на обычной ленте записано входное слово I, начиная с ячейки с номером 1. Все остальные ячейки обеих лент пусты.



Оглавление.

Работа оракульной машины Тьюринга (ОМТ) почти аналогична работе двухленточной МТ. Разница в следующем. Если управляющее устройство оказывается в состоянии q_c, то поведение на следующем шаге зависит от фиксированной оракульной функции $g:A^* \rightarrow A^*$.

Этот шаг не изменяет положение основной головки и запись на основной ленте. Его результатом является переход в состояние q_r и изменение за один шаг всего содержимого оракульной ленты по следующему правилу.

Пусть y - слово, записанное на оракульной ленте, начиная с первой ячейки и до ячейки, над которой находится головка. Если головка находится левее ячейки с номером один, то полагается, что y - некоторая заранее оговоренная постоянная. За данный такт на оракульной головке записывается слово g(y), начиная с первой ячейки. Остальная часть ленты стирается, а головка обозревает ее первую ячейку.

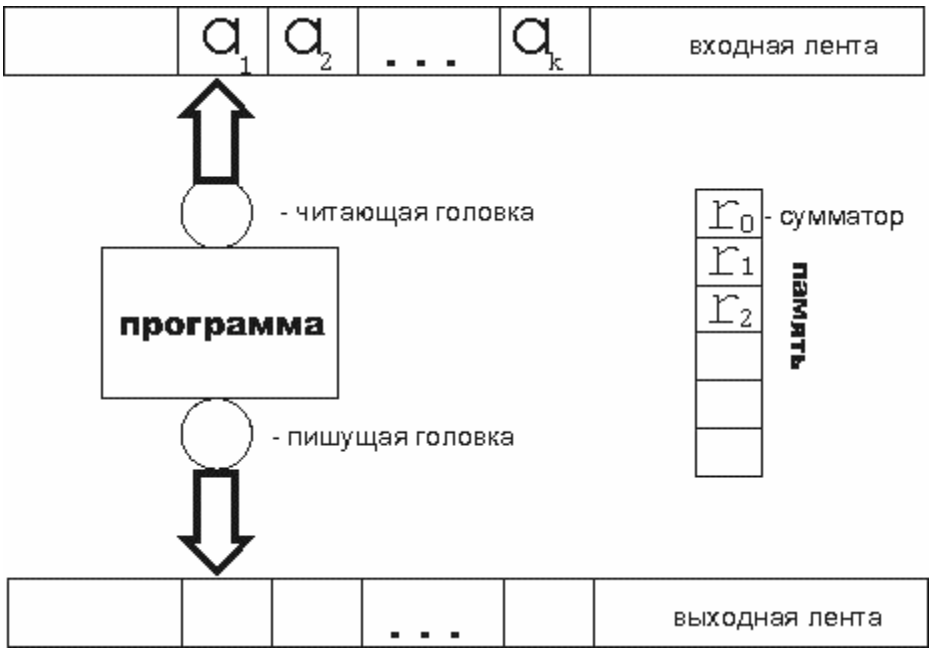
То есть мы имеем как бы комбинацию обычной машины Тьюринга M и некоторого оракула g. Вообще говоря, для одной и той же задачи могут быть использованы разные оракулы. Поэтому можно искусственно вычленить ту часть программы ОМТ, которая касается работы обычной головки. Она ведь не касается действий оракульной головки. Именно эту часть программы называют *программой* ОМТ. Обозначим ее через M. В случае же рассмотрения конкретного оракула g всю программу целиком будем обозначать через M_g, и называть *релятивизированной программой* ОМТ.

5. Равнодоступная адресная машина (РАМ) и некоторые другие подходы.

Машина Тьюринга интуитивно более похожа на современный компьютер, чем НАМ. Но в ней нельзя за один шаг добраться до фиксированной ячейки. В то время как наше интуитивное представление об автоматических вычислениях, созданное в основном знакомством с компьютерами, предполагает такую возможность.

Равнодоступная адресная машина (РАМ) представляет собой усложнение МТ (см. [1]). Вместо одной ленты и головки имеются две: входная, с которой можно только читать, и выходная, предназначенная для записи.

Оглавление.



Вычисления производятся в сумматоре. РАМ имеет также неограниченный объем памяти в виде последовательности перенумерованных ячеек.

Работой лент, сумматора, головок управляет программа, состоящая из перенумерованной последовательности команд. Наличие счетчика команд позволяет расставлять метки в программе.

Программа для РАМ не записывается в память, поэтому предполагается, что программа не изменяет сама себя.

Все вычисления происходят в первом регистре, называемом *сумматором*.

Команда состоит из *кода операции* и *адреса*. Это известно всем, кто программировал на языке ассемблера. Операнд команды может задаваться тремя способами.

1. В виде целого числа (литерала) i, что соответствует команде присвоения в программе. Это записывается как =i .
2. Либо в адресе команды содержится адрес регистра, в котором лежит значение операнда. Здесь i – содержимое регистра,
3. Либо в адресе команды содержится адрес регистра (указатель), в котором содержится адрес регистра, содержащего значение операнда. Здесь *I означает косвенную адресацию. В случае отрицательного адреса программа останавливается.

Оглавление.

Счетчик команд вначале установлен на первую команду в программе P, выходная лента пустая, содержимое c(i)=0 для любого регистра i. После выполнения k-й команды P счетчик команд переходит на (k+1)-ю команду, если это не была команда условного перехода. (См. пример ниже).

Чтобы описать действия команды, задаются значения *v(a)* операнда *a*.

1. *v(=i)=i.*
2. *v(i)=c(i).*
3. *v(*i)=c(c(i)).*

Меняя набор типов команд, можно говорить не о РАМ, а о разных формальных схемах алгоритма, принадлежащих к определенному типу. Но все они содержат некоторый набор базовых операций, поэтому можно считать, что программирование на них имеет одинаковую сложность.

Пусть набор типов команд фиксирован. Тогда алгоритм решения некоторой задачи в данном формализме включает в себя входной и выходной алфавит, а также текст программы.

В [1] исследуется РАМ, имеющая 12 типов команд. Приведем ее в качестве примера. Это команды начала работы и останова, команды чтения и письма, четыре арифметические операции, команда хранения, команда перехода на метку и две команды условного перехода в зависимости от содержимого сумматора.

Команда	Действие
LOAD a	c(0)←v(a)
STORE(i), STORE(*i)	c(i) ← c(0)
ADD(a)	c(0) ←c(0)+v(a)
SUB(a)	c(0) ←c(0)-v(a)
MULT(a)	c(0) ←c(0)v(a)
DIV(a)	c(0) ←[c(0)/v(a)]
READ(i), READ(*i)	c(i) ← очередной входной символ, c(c(i)) ← очередной входной символ. Головка входной ленты сдвигается на

Оглавление.

	клетку вправо.
WRITE(a)	v(a) печатается в клетке выходной ленты, которую на данный момент обозревает головка выходной ленты. Головка выходной ленты сдвигается на клетку вправо.
JUMP(b)	Счетчик команд устанавливается на команду с меткой b.
JGTZ(b)	Если c(0)>0, то счетчик команд устанавливается на команду с меткой b, в противном случае – на следующую команду.
JZERO(b)	Если c(0)=0, то счетчик команд устанавливается на команду с меткой b, в противном случае – на следующую команду.
HALT	Работа программы прекращается.

Создание РАМ для приведенных выше примеров в разделах, описывающих НАМ и МТ, выглядит очень просто. В качестве менее очевидных примеров приведем следующие.

Пример 1. Написать программу для РАМ, которая для натуральных чисел вычисляет функцию n^n.

Пример 2. Дан алфавит {1,2}. Написать программу для РАМ, которая распознает слова в этом алфавите, содержащие одинаковое число вхождений 1 и 2.

Заметим, что программа РАМ не хранится в памяти, поэтому она себя изменять не может.

Равноадресная доступная машина с хранимой программой (РАСП) отличается от РАМ тем, что ее программа хранится в памяти. Каждая команда занимает два регистра памяти, в первом хранится код команды, во втором - адрес.

Во многих источниках при описании алгоритмов используются дальнейшие усложнения этих формальных схем в виде "упрощенных" языков программирования.

Мы рассмотрели достаточное количество примеров. Перейдем к их сравнительному анализу.

Оглавление.

6. Сравнение различных формальных схем.

6.1. Кодировки входных данных.

Вначале поговорим о длине входа (условия) задачи. Дело в том, что кодировать условия задачи можно разными способами. Поясним это на примерах.

Пример 1. Задача: *Требуется найти количество единиц в двоичной записи целого числа.*

Если число n задается в виде n+1 единицы *1...1*, то длина входа n+3. Если число задается в десятичной форме, это уже lgn. К этим двум способам добавим двоичную запись числа n. Ее длина log_2n.

Но рассмотрим еще один подход к решению задачи. Пусть у нас есть 2^n перенумерованных ячеек, в каждой из которых содержится число, равное количеству единиц в двоичной записи номера ячейки. Тогда длина входа задачи для числа n равна 2^n log_2n.

Пример 2. *Задача о простоте числа.*

Если число n задается в виде n+1 единицы *1...1*, то длина входа n+3. Если число задается в десятичной форме, это уже lgn. Если n задано в мультипликативной форме $n=p_1^{a1}\ldots p_k^{ak}$ (числа p_i– простые, а сама *мультипликативная* форма задает разложение числа на простые множители), то длина входа равна $\Sigma(lgp_i+lga_i)$.

Пример 3. *Задачи на графах*.

Выше было много примеров задач на графах. Граф с n вершинами и m ребрами можно задать списками ребер и вершин. В этом случае длина входа лежит между 4n+10m и 4n+10m+(n+2m)[lgn]. Если граф задается списками соседей его вершин, то длина входа лежит между 2n+8m и 2n+8m+2m[lgn]. Порядок же матрицы инциденций графа равен n^2-n+1.

Так как понятие *входа* (условия) задачи не может быть формализовано хотя бы потому, что не формализовано само понятие *задачи,* то здесь мы вновь вынуждены уповать на наши интуитивные представления.

Оглавление.

Это и иллюстрирует первый из приведенных выше примеров. Описанные там два подхода к заданию входа фактически относятся к разным задачам. То есть, понимание условия задачи на интуитивном уровне дает возможность предлагать "разумную" кодировку в том смысле, что разумность заключается в соответствии кодировки условиям задачи.

Первый пример иллюстрирует и важность еще одной меры сложности задачи - объем требуемой при решении памяти. При первом подходе к кодировке входа в данном примере объем памяти в константу раз отличается от длины входа и представляет собой линейную или логарифмическую функцию от n. Время решения задачи, например, ее *сложность в худшем* (см. ниже) на МТ такая же. Во втором случае решение просто сводится к чтению ответа (на МТ это один такт работы), объем же памяти экспоненциально зависит от n.

Пусть мы теперь имеем *"разумные" кодировки*. На этом уровне принимается интуитивная гипотеза о некоторой эквивалентности всех возможных "разумных" кодировок входных данных одной и той же массовой задачи.

Второй пример показывает варианты длины кодировок в зависимости от способа задания чисел. Варианты длины кодировок в третьем примере зависят уже от структуры входных данных. Но в обоих случаях можно говорить о, так называемой, полиномиальной эквивалентности.

В дальнейшем нам полезно будет понятие *полиномиальной эквивалентности* двух функций. Две неотрицательные функции S(n) и R(n) полиномиально эквивалентны, если существуют два полинома Q(x) и P(x) такие, что для всех n справедливы неравенства $S(n) \le P(R(n))$ и $R(n) \le Q(S(n))$.

Пусть S и R две "разумные" схемы кодирования задачи массовой Z. Длины входа в этих схемах для задачи I обозначим через S(I) и R(I). К ним можно применить определение полиномиальной эквивалентности.

В третьем примере все три схемы полиномиально эквивалентны, а во втором полиномиально эквивалентны две последние.

Оглавление.

6.2. О мерах сложности

Сегодня понятие алгоритма ассоциируется в первую очередь с программой на некотором языке программирования. С точки зрения тезиса Черча, это достаточно естественно.

Однако нас будет интересовать не только проблема эквивалентности различных типов алгоритмов, но также возможность их сравнения и те свойства моделей алгоритмов, которые позволяют оценивать сложность.

Здесь есть определенное противоречие. Проще оценивать сложность алгоритма в рамках достаточно замысловатой модели, например, современного языка программирования. В то же время хотелось бы понимать, насколько подобная модель сравнима, например, с машиной Тьюринга. И не только в смысле эквивалентности, но и при пересчете оценок сложности алгоритма в рамках различных моделей вычисления. Это вопрос очень сложный. Поэтому здесь мы его лишь вскользь коснемся и приведем несколько примеров. В дальнейшем мы периодически будем к нему возвращаться.

В случае решения индивидуальной задачи на машине Тьюринга естественно оценивать сложность алгоритма числом тактов работы МТ. Для НАМ такой мерой может быть количество выполненных в процессе работы преобразований.

В случае НМТ сложность определяется следующим образом. Для одного и того же слова I может существовать множество различных отгадок {U}. На каждой из них обычная головка работает $t_T(I,U)$ тактов. В качестве времени решения задачи на входе I рассматривается

$$t_T(I) = \min_{U \in \{U\}} t_T(I,U).$$

Обратим внимание на то, что понятие «отгадка», с одной стороны, имеет оракульный подтекст, а с другой – связано с примитивным полным перебором. Пусть существует конечная запись U', обозначающая объект, соответствующий ответу «да». Например, перечень вершин гамильтонова цикла или обход коммивояжера нужного веса. Так как угадывающая головка случайным образом пишет слова, то **один** из вариантов ее работы – упомянутая запись. Но в приведенной выше формуле минимум берется по всем возможным записям, поэтому и U’ будет учтена. Дальше работает обычная головка, которая читает отгадку и проверяет. Отгадки, конечно, могут иметь разную длину, поэтому суммарное время чтения и проверки для них может быть разное. Формула даст минимум по этим

Оглавление.

Гордеев Э.Н. «Введение в теорию сложности алгоритмов».

временам. Пусть он достигается на слове-отгадке V. Так как поиск этого минимума нам ничего с вычислительной точки зрения не стоит, то, можно считать, слово V сразу пишется угадывающей головкой. В этом заключается «оракульность» отгадки. Но оракульность эта достигается *бесплатностью* перебора по всем возможным записям случайно работающей угадывающей головки. В этом заключается переборный характер механизма получения отгадки.

Для ОМТ сложность решения задачи на входе I определяется совершенно так же, как и для МТ. (Только вычисления-то эти очень разные!)

Кроме временных характеристик рассматриваются и пространственные. В случае МТ сложность задачи может характеризоваться количеством ячеек ленты, которые просматривались головкой в процессе работы.

Говоря о сложности вычисления РАМ, РАСП или упрощенного АЛГОЛа временную сложность индивидуальной задачи оценивают с помощью количества выполненных в процессе решения команд. Но здесь выделяются два подхода.

Если каждая команда требует единицы времени, а каждый регистр занимает единицу памяти, то говорят о *моделях вычисления с равномерным весом*.

Модель вычисления с логарифмическим весом предполагает, что представление целого числа n в регистре требует $\lfloor log|n| \rfloor+1$ битов. При этом емкость регистров не ограничена. Время, затраченное на выполнение команды пропорционально длине ее операндов.

Сложность решения массовой задачи можно определять разными способами, исходя их сложности решения индивидуальных задач.

Ниже через T обозначим некоторую модель вычисления (НАМ, МТ, РАМ и т.п.).

Пусть |Z(n)| - число индивидуальных задач I массовой задачи Z, длина входа которых не превосходит n.

Обозначим время работы алгоритма T (модели вычислений T) на входе I через $t_T(I)$. В случае решения массовой задачи в качестве меры сложности рассматривается функция $t_T(n)$ от длины |I| входа задачи I. Наиболее распространены две меры сложности: "в худшем" и "в среднем". В первом случае

Оглавление.

$$t_T(n) = \max_{I:|I|\le n} t_T(I).$$

А во втором

$$t_T(n) = \frac{\sum_{I:|I|\le n} t_T(I)}{|Z(n)|}.$$

Максимум и сумма берутся по всем индивидуальным задачам I таким, что |I|≤n.

Будем различать *сложность задачи* T_Z и *сложность алгоритма* A(Z) решения этой задачи $T_{A(Z)}$. Если фиксирован алгоритм решения задачи A(Z), то по его описанию любой пользователь имеет возможность подробно (пошагово) увидеть, как он работает. Если в качестве сложности алгоритма выбрана некоторая характеристика его работы, например, одна из приведенных выше мер, то используя описание алгоритма, это меру можно вычислить.

Итак, под сложностью алгоритма понимается значение некоторой функции от длины входа. В то же время сложность задачи, как функция от длины входа – это отдельное понятие. Здесь может быть несколько подходов.

Если сложность задачи напрямую определяется сложностью алгоритмов ее решения, то в большинстве случаев удается говорить лишь об оценках этой сложности. Задачу можно решать несколькими алгоритмами, поэтому полагают, что

$$T_Z \le T_{A(Z)}.$$

Верхней оценкой сложности является наименьшая из известных на данный момент сложность алгоритма ее решения. Если, $T_{A(Z)}$ - полиномиально ограниченная функция, то говорят, что алгоритм A(Z) (задача Z) имеет полиномиальную сложность, а если экспоненциально ограниченная – то экспоненциальную сложность. В следующем разделе мы в качестве примеров такого подхода рассмотрим несколько известных вам задач.

Если бы была известна некоторая нижняя оценка сложности

$$F(n) \le T_Z,$$

совпадающая с верхней, то, то сложность задачи была бы определена, а исследователи бы знали, что существующий на данный момент алгоритм ее решения является наилучшим из возможных и

Оглавление.

Гордеев Э.Н. «Введение в теорию сложности алгоритмов».

дальнейшие попытки поиска более быстрого алгоритма будут безуспешными. Но такая ситуация имеется лишь в очень малом количестве сравнительно простых задач, таких как приведенные ниже: задача нахождения минимального числа (здесь сложность равна O(n)) или задача сортировки (здесь сложность равна O(n*log*n)).

Дело в том, что нижняя оценка сложности получается из анализа структуры задачи, а не из методов ее решения. На сегодня типичной является ситуация, когда нижняя оценка – экспонента, нижняя – линейная функция. Экспоненциальные нижние оценки сложности для некоторых задач были впервые получены в 1980-х годах.

Другой подход к анализу сложности приведен в разделе 8 и связан с понятием *схемы из функциональных элементов*.

6.3. Теоремы сравнения

Подведем теперь некоторый итог нашему рассмотрению. Для этого мы сформулируем и обсудим несколько утверждений. Подробные доказательства можно найти в [3] и в [1].

Интуитивно очевидно, что сводить сравнительно бедный (простой) формализм к более богатому (сложному) тяжелее, чем наоборот. С другой стороны, оценивать сложность решения той или иной задачи с помощью программы на современном языке программирования проще, чем сложность ее решения на МТ. В то же время многие теоретические результаты сформулированы для такой модели, как МТ. Утверждения данного раздела и позволяют проверить, насколько такие оценки теоретически обоснованы.

Первые два утверждения касаются НАМ и МТ.

Теорема. Пусть T - машина Тьюринга с алфавитом A и C - надалфавит A. Тогда существует нормальный алгорифм Маркова $\mathfrak{I}$ над C, вполне эквивалентный алгоритму Тьюринга $\mathfrak{R}_{T,C}$.

Справедливо и обратное утверждение.

Теорема. Пусть $\mathfrak{I}$ - нормальный алгорифм Маркова в алфавите A, $\Delta,\delta \notin A$. $C=\{\Delta,\delta\}\cup A$. Тогда существует машина Тьюринга T такая, что алгоритм Тьюринга $\mathfrak{R}_{T,C}$ в алфавите C обладает следующим свойством. Для любого слова P в алфавите A $\mathfrak{R}_{T,C}$ применим в том и только в том случае,

Оглавление.

когда к этому слову применим $\mathfrak{I}$, а результатом работы $\mathfrak{R}_{T,C}$ является слово $\mathfrak{I}(P)$, окруженное конечными последовательностями из Δ.

Займемся теперь сравнение РАМ и РАСП.

Теорема. Как при равномерном, так и при логарифмическом весе для любой РАМ-программы с временной сложностью T(n) найдется эквивалентная ей РАСП программа с временной сложностью, не превосходящей kT(n), k - некоторая постоянная.

Теорема. Как при равномерном, так и при логарифмическом весе для любой РАСП-программы с временной сложностью T(n) найдется эквивалентная ей РАМ программа с временной сложностью, не превосходящей kT(n), k - некоторая постоянная.

Эти утверждения позволяют использовать в рассуждениях ту из моделей, которая в данном случае более удобна.

Рассмотрим теперь связь РАМ и МТ.

Возьмем задачу в форме распознавания Z и соответствующий ей язык L_Z. Все перечисленные формальные схемы алгоритма позволяют ввести понятие допустимости языка алгоритмом.

Например, МТ допускает язык L_Z тогда, когда она останавливается в специально отведенном для этого конечном состоянии q_y только на словах этого языка, являющихся условиями индивидуальных задач с ответом "да".

Конечно, вместо состояния можно требовать написания какого-то символа на ленте. Так в РАМ, РАСП и упрощенном АЛГОЛе говорят, что программа допускает язык L_Z тогда, когда она останавливается, записав на выходной ленте 1 только на словах этого языка, являющихся условиями индивидуальных задач с ответом "да".

Заметим, что на всех остальных словах программа либо вообще не останавливается (зацикливается), либо останавливается с некоторой другой записью на ленте.

Теорема. Пусть L - язык, допускаемый некоторой РАМ за время T(n) при логарифмическом весе. Если в программе РАМ нем умножений и делений, то существует многоленточная МТ, допускающая этот язык за время $O(T^2(n))$.

Аналогичное утверждение для равномерного веса неверно.

Оглавление.

Действительно, в случае МТ равномерного веса быть не может. В ячейке записан один символ. Его расшифровка внутри программы просто должна развернуть всю запись числа. А как это делается: явным образом с занятием новых ячеек или неявно в виде последовательности выполняемых команд, - это уже не важно.

Для РАМ же логарифмический вес является абстрактным упрощением, которое иногда полезно, если применяется осознанно.

Сказанное поясняет пример умножения n чисел вида n^{2^n}.

РАМ это может сделать за O(n) шагов при равномерном весе. Ведь в регистр помещается число любого размера. В то время как на МТ только вклад тактов, на которых осуществляется считывание входного слова, составит более 2^n. То есть даже полиномиальной эквивалентности между сложностями вычислений в этих моделях.

6.4. Полиномиальные и неполиномиальные оценки сложности

С ростом n время вычисления $t_T(n)$ обычно растет. Для конкретной модели вычисления зачастую проще и нагляднее сравнивать не сами времена, а их верхние оценки.

При грубых оценках можно сравнивать лишь типы функций, при более точных - степени полинома или даже коэффициенты.

В качестве примеров можно привести следующие две известные таблицы.

Следующая таблица позволяет оценить скорость роста некоторых полиномиальных и экспоненциальных функций.

	Значение n					
Функция сложности	10	20	30	40	50	60
n	0.00001 сек	0.00002 сек	0.00003 сек	0.00004 сек	0.00005 сек	0.00006 сек
n^2	0.0001 сек	0.0004 сек	0.0009 сек	0.0016 сек	0.0025 сек	0.0036 сек

Оглавление.

n^3	0.001 сек	0.008 сек	0.027 сек	0.064 сек	0.125 сек	0.216 сек
n^5	0.1 сек	3.2 сек	24.3 сек	1.7 мин	5.2 мин	13 мин
2^n	0.001 сек	1 сек	17.9 мин	12.7 дней	35.7 лет	366 веков
3^n	0.059 сек	58 мин	6.5 лет	3855 веков	$2x10^8$ веков	$1.3x10^{13}$веков

В следующей таблице отражено влияние совершенствования ЭВМ на возможности алгоритмов.

	Размеры наибольшей задачи, решаемой за один час		
Функция сложности	На современных ЭВМ	На ЭВМ, в 100 раз более быстрых	На ЭВМ, в 1000 раз более быстрых
n	a	100a	1000a
n^2	b	10b	31.6b
n^3	c	4.64c	10c
n^5	d	2.5d	3.98d
2^n	e	e+6.64	e+9.97
3^n	f	f+4.19	f+6.29

Эти таблицы, в частности, иллюстрируют различие между полиномиальными и экспоненциальными алгоритмами. Тем самым аргументируется внимание к разделению алгоритмов на классы по трудоемкости.

Заметим еще, что это различие имеет и другую сторону. Вспомним, например задачу о камнях. Всего различных способов разбиения кучи из n камней на две части - 2^n. Пусть просмотр одного варианта требует O(n) тактов работы, например, МТ. Тогда самый простой способ решения задачи - перебор всех возможных вариантов - потребует O(n) 2^n тактов работы.

Оглавление.

Это обычная ситуация. Она может быть проиллюстрирована и на других упомянутых выше примерах. Поэтому экспоненциальные алгоритмы ассоциируются с простым полным перебором вариантов (отсюда и термин *переборный алгоритм*).

В то же время для построения полиномиальных алгоритмов требуются обычно многочисленные ухищрения, позволяющие избежать перебора.

Более подробные примеры рассмотрены в следующем разделе.

7. Сложность алгоритмов некоторых задач.

Данный раздел содержит примеры алгоритмов, иллюстрирующих проведенные выше рассуждения. Также алгоритмы и задачи этого разделы будут использованы в дальнейшем при обсуждении *классов сложности* задач. Этот раздел ни в коем случае не является попыткой обзора алгоритмов и методов их построения. (Такая попытка сделана, например, в [5], а монографии [1] и [4] целиком посвящены описанию алгоритмов и методов их построения). Более того, примеры этого раздела в большинстве своем либо очевидны, либо знакомы вам из курса дискретной математики.

Далее коротко приводится алгоритм и оценка его сложности «в худшем». Описание задач, а также обозначение их параметров приведены выше.

7.1. Задача нахождения максимального числа.

Решение дает очевидный простой алгоритм: последовательно сравниваем числа и в отдельной ячейке храним максимальное из просмотренных. Для решения необходимо просмотреть все числа, т.е. сделать не менее n-1 сравнений. Но за n-1 сравнений мы найдем решение задачи. Поэтому данный алгоритм позволяет найти сложность задачи, т.е сложность алгоритма - O(n) является одновременно и сложностью задачи нахождения максимального числа. Верхняя и нижняя оценка сложности совпадают.

7.2. Задача сортировки.

В дискретной математике сортировке подлежат объекты разной природы. Пусть необходимо отсортировать (например, по возрастанию) числовой массив из n элементов. Без ограничения общности будем считать, что – степень двойки (чтобы не возиться с целыми частями). В качестве элементарной операции используется операция сравнения чисел. Чтобы сравнивать числа нужно их различать как объекты сортировки (эти объекты имеют численные значения). Сравнение – это операция с результатом «да» или «нет», поэтому кодируем объекты сортировки бинарными последовательностями. Какова минимальная длина такой последовательности для множества их M элементов? Очевидно, что это log M. В теории информации этот факт известен как энтропия по Хартли.

Если мы упорядочиваем n объектов (чисел), то в зависимости от их значений можно получить n! различных *упорядочиваний*. Последовательность сравнений для их различения не может быть меньше log(n!). Применяем формулу Стирлинга: $\log(n!)=(n+1/2)\log n-n+0{,}5\log 2\pi(n)+\alpha/(12n)$, $0<\alpha<1$, $n\to\infty$. Отсюда следует, что задачу сортировки нельзя решить со сложностью алгоритма, меньшей O(nlogn).

С 1950-х годов разработано много методов сортировки со сложностью алгоритма, равной O(nlogn).

7.3. Задача о камнях.

Здесь тоже дан массив из n чисел, но его не нужно упорядочивать или искать в нем число. Нужно попробовать разложить его на две «максимально» равные части. Каждая из этих частей является подмножеством всего множества камней и полностью определяет другую (оставшиеся камни). Всего подмножеств, как вы знаете, 2^n. На сегодняшний день не существует алгоритма решения этой задачи, отличного от полного перебора. Перебирая последовательно все подмножества мы сравниваем их вес с числом $(\Sigma a_i)/2$). Если получили совпадение, то ответ «да», если же такого совпадения мы не получаем после того, как все множества просмотрены, то ответ «нет». Сложность такого алгоритма $O(n2^n)$.

Оглавление.

7.4. Простота числа.

Пусть дано натуральное N. Тогда длина входа задачи: n=logN.

Из школы известно два алгоритма: *простой перебор* и *решето Эратосфена*.

В первом случае нужно осуществить $n^{1/2}$ делений. Сложность алгоритма $t(n)=2^{n/2}$.

Можно показать, что сложность алгоритма *решето Эратосфена* составляет $O(2^n \log n)$ операций в модели вычислений RAM, или $O(2^n\, n(\log n))$ битовых операций, при условии вычисления и зачеркивания каждого кратного числа за время O(1). Таким образом, другая из приведенных в качестве примеров задач – **задача о разложении числа на множители** – тоже имеет экспоненциальную сложность, так как задача о простоте – это частный случай.

7.5. Задача о кратчайшем (минимальном) остове (остовном дереве).

Пусть дан связный граф G=(V,E), |V|=n, |E|=m, с матрицей весов ребер W=(w(i,j)). Остовное дерево – это его подграф на том же множестве вершин, который является деревом. Вес дерева – сумма весов ребер этого дерева. Требуется найти остовное дерево минимального веса.

Всего нумерованных деревьев на n вершинах n^{n-2}. Если устроить перебор по всем этим деревьям, то получим алгоритм сложности $O(n^{n-1})$. Но есть и более простые алгоритмы.

Алгоритм Крускала (или алгоритм Краскала). Алгоритм впервые описан Джозефом Крускалом в 1956 году.

Вначале текущее множество рёбер устанавливается пустым. Затем, пока это возможно, проводится следующая операция: из всех рёбер, добавление которых к уже имеющемуся множеству не вызовет появление в нём цикла, выбирается ребро минимального веса и добавляется к уже имеющемуся множеству. Когда таких рёбер больше нет, алгоритм завершён. Подграф данного графа, содержащий все его вершины и найденное множество рёбер, является его остовным деревом минимального веса. (Корректность алгоритма следует из теоремы Радо-Эдмондса и того факта, что данная задача – частный случай оптимизационной задачи на *матроиде*. [4])

Оглавление.

До начала работы алгоритма необходимо отсортировать рёбра по весу, это требует O(m *log*m) времени. После чего компоненты связности удобно хранить в виде системы непересекающихся множеств. Предполагается, что проверка на появление цикла требует константу времени. (В общем случае она требует времени выражаемом через функцию Аккермана - α(n,m), но в нашем случае α(n,m)<5.) Все операции в таком случае займут O(m), , таким образом общее время работы алгоритма Крускала можно принять за O(m *log*m). Это, конечно, не $O(n^{n-1})$.

Алгоритм Прима — алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа. Алгоритм впервые был открыт в 1930 году чешским математиком Войцехом Ярником, позже переоткрыт Робертом Примом в 1957 году.

Построение начинается с дерева, включающего в себя одну (произвольную) вершину. В течение работы алгоритма дерево разрастается, пока не охватит все вершины исходного графа. На каждом шаге алгоритма к текущему дереву присоединяется ребро минимального веса, соединяющих вершину из построенного дерева, и вершину не из дерева.

Асимптотика алгоритма зависит от способа хранения графа и способа хранения вершин, не входящих в дерево.

1. Если приоритетная очередь *Q* реализована как обычный массив *d*, то *Extract*.*Min*(*Q*) выполняется за *O*(*n*), а стоимость операции присвоения d[u]=w[v,u] составляет *O*(1). Трудоемкость алгоритма Прима в этом случае $O(n^2)$.
2. Если *Q* представляет собой *бинарную пирамиду*, то стоимость *Extract*.*Min*(*Q*) снижается до *O*(log *n*), а стоимость присвоения d[u]=w[v,u] возрастает до *O*(log *n*). Трудоемкость алгоритма Прима в этом случае совпадает с трудоемкостью алгоритма Краскала - *O*(m logn).
3. При использовании *фибоначчиевых пирамид* операция *Extract*.*Min*(*Q*) выполняется за *O*(log *n*), а присвоения d[u]=w[v,u] за *O*(1). Трудоемкость алгоритма Прима в этом случае *O*(m +n logn).

Все это опять же не $O(n^{n-1})$.

Оглавление.

Гордеев Э.Н. «Введение в теорию сложности алгоритмов».

7.6. Задача коммивояжера.

Число гамильтоновых циклов в графе G=(V,E), |V|=n, |E|=m, с матрицей весов ребер $C=(c_{ij})$, равно (n-1)!. Сумму весов ребер одного гамильтонова цикла можно найти за O(n). Общая трудоемкость алгоритма $O(n!) = O((n/e)^{n+1/2})$. Так может быть здесь, как и в предыдущем случае, можно ее сократить до простого полинома? Оказывается нельзя. На сегодня не существует полиномиальных алгоритмов решения задачи коммивояжера, лучшие алгоритмы имеют экспоненциальную трудоемкость «в худшем случае».

В таких случаях на практике вместо точного решения пытаются найти приближенное или используют методы «*направленного перебора*» (*методы ветвей и границ*), но об этом мы поговорим позже.

7.7. Задача о кратчайшем пути.

В графе G=(V,E), |V|=n, |E|=m, с матрицей весов ребер $C=(c_{ij})$ для двух фиксированных вершин ищется путь наименьшей длины между ними. (Длина пути – сумма весов входящих в него ребер). Если мы будем искать его методом полного перебора, то снова получим экспоненциальную оценку. И ничего с ней поделать нельзя, если веса ребер – произвольные числа. А если они неотрицательны, то можно избежать переборного алгоритма и построить более эффективный *Алгоритм Дейкстры*.

Сложность алгоритма Дейкстры зависит от способа нахождения вершины *v*, а также способа хранения множества непосещенных вершин и способа обновления меток. Обозначим через *n* количество вершин, а через *m* — количество ребер в графе *G*.

В простейшем случае, когда для поиска вершины с минимальной пометкой просматривается все множество вершин время работы алгоритма есть $O(n^2 + m)$. Для разреженных графов (то есть таких, для которых m много меньше n^2) скорость работы возможной реализации $O(n\log n + m\log n)$.

Оглавление.

7.8. Задача о выполнимости КНФ (КНФ-выполнимость)

Задана булевская функция f от n переменных в виде конъюнктивной нормальной формы K. Требуется определить, существует ли такой набор значений переменных, на котором функция обращается в единицу. Здесь мы имеем ту же самую ситуацию, что и с задачей о камнях и с задачей коммивояжера. На сегодня не существует алгоритмов, отличных от вариантов перебора с экспоненциальной сложностью. Простейший (сложность $O(n2^n)$) состоит в подстановке в КНФ всех двоичных наборов возможных значений переменных. В курсе дискретной математики вы изучали различные методы нахождения ДНФ минимальной длины. Все они тоже имеют экспоненциальную трудоемкость «в худшем случае». По существу, они просто делают полный перебор «*разумным*» и «*направленным*», но всегда можно привести пример задачи, когда эта «разумность» и «направленность» не дает никаких улучшений по сравнению с «простым» перебором.

8. Схемы из функциональных элементов. Схемная сложность.

8.1. Схемы из функциональных элементов

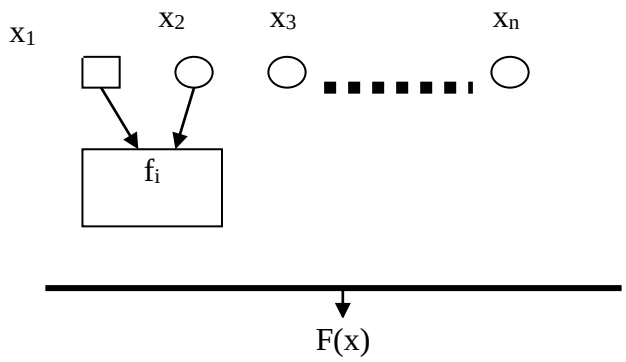
В этом разделе мы приведем некоторые необходимые для дальнейшего сведения о булевых функциях.

Схема из функциональных элементов (булева схема) - это способ вычисления булевых функций: $\boldsymbol{B}^n \to \boldsymbol{B}^m$. Эта *схема* может быть представлена в виде ациклического графа с тремя видами вершин. Вершинам с нулевой полустепенью захода (число таких вершин равно n) сопоставляются n входных переменных. Вершинам с нулевой полустепенью исхода (число таких вершин равно m) сопоставляются m выходных значений функции.

Всем остальным вершинам (функциональным элементам) сопоставляются некоторые булевы функции. В этом случае полустепень захода равна числу переменных функции, а ребра помечены числами, указывающими номера аргументов функции. Схема отождествляется с формулой, если из каждой такой вершины выходит ровно одно ребро.

Оглавление.

Совокупность таких функций (типов функциональных элементов), на которых строится схема, образует базис схемы. Вычисление происходит поэтапно.



Функция в вершине схемы считается вычисленной сразу, как только известны значения ее аргументов. Базис называется полным, если для любой булевой функции существует схема вычисления в данном базисе.

Базисом может быть любая полная система булевых функций (вспомним теорему Поста). Поэтому без ограничения общности можно рассмотреть фиксированный базис.

Пример такого базиса дает следующее утверждение следующее утверждение.

Теорема. Базис из трех функций $\{\cup, \&, \neg\}$ является полным.

Определение. *Схема из функциональных элементов* (СФЭ) — это конечный ориентированный граф без ориентированных циклов, в каждую вершину которого входит не более 2 рёбер. При этом каждой вершине приписывается символ: переменная x_j, если в эту вершину рёбра не входят; отрицание, если в вершину входит одно ребро; конъюнкция или дизъюнкция, если в вершину входит 2 ребра. Некоторым вершинам приписывается *. *Элементами* схемы называются вершины, помеченные логическими операциями.

Сложностью схемы называется число функциональных элементов.

Схемной сложностью $L_B(f)$ вычисления функции f в базисе *B* называется минимальный размер схемы, вычисляющей f в данном базисе.

Оглавление.

Гордеев Э.Н. «Введение в теорию сложности алгоритмов».

$$L_{B,n}(f) = \min L_{B,n}(f)$$

Здесь минимум берется по всем схемам, вычисляющим функцию f(n) в базисе B.

Если базис фиксирован, *B* то индекс опускаем. Отдавая дань истории функция L(n) выражающая максимальную сложность функций от n переменных, называется *функцией Шеннона*.

Какие для этой функции существуют оценки?

Этот вопрос для нас важен в связи с тем, что СФЭ можно использовать для подхода к анализу *сложности задачи*.

Рассмотрим задачу в форму распознавания. Закодируем вход задачи Z булевским вектором из $\boldsymbol{B}^n$, тогда решение задачи – это некоторое отображение φ_z: $\boldsymbol{B}^n \to \boldsymbol{B}^1$. Предположим, что это отображение в явном виде построено, тогда будет определена некоторая булевская функция. Схемная сложность этой функции может задавать *сложность решения* задачи.

Опр. Класс булевых функций, для которых существуют схемы полиномиальной сложность и обозначим через P/poly.

В связи с тем, что этот класс будет рассматриваться ниже, нас интересуют оценки функции Шеннона.

Эта задача была решена советским математиком О.Б.Лупановым. Кратко приведем основные результаты.

Утв. $L(n) \le (n+1)\ 2^n$.

Теорема. Справедливо соотношение $L(n) <\cong 2^n / n$.

9. Теория NP-полноты

В конце 60-х и начале 70-х годов был внесен интересный вклад в теорию сложности, касающийся сравнения различных типов алгоритмов. Главная роль здесь принадлежит американскому математику С.А.Куку.

Оглавление.

Три фундаментальных части предложенной Куком концепции заключаются в следующем.

1. Были введены определения двух классов задач. Один из них был связан с полиномиальными алгоритмами, а другой - с переборными.
2. Были введены понятия *сводимости* задач. Сводимость позволяла характеризовать принадлежность к определенному классу.
3. Для некоторой конкретной задачи было конструктивно доказано, что она в определенном смысле может рассматриваться, как самая сложная. (С помощью сводимости затем множество таких самых сложных задач было дополнено.)

9.1. Классы P и NP.

Рассмотрим дискретную оптимизационную задачу Z в форме распознавания. Ей, как было выше описано, сопоставляется язык L_Z.

Класс *P* - это класс языков, допускаемых МТ за полиномиальное время. (То есть эта МТ, во-первых, допускает язык, а, во-вторых, время ее работы ограничено полиномом.)

Задачи (языки) из этого класса называют *полиномиально разрешимыми*.

В случае НМТ для одного и того же слова I, представляющего собой запись условия некоторой индивидуальной задачи, может существовать множество различных отгадок {U}. Зафиксируем слово I и рассмотрим все возможные вычисления НМТ на различных отгадках. На каждой из них обычная головка работает $t_T(I,U)$ тактов.

Если хотя бы для одного такого вычисления НМТ за конечное число шагов остановится в конечном состоянии q_y ,то это вычисление называется принимающим, $t_T(I,U)$ полагается равным числу тактов работы НМТ. В противном случае (НМТ зацикливается или останавливается в состоянии q_n) вычисление называется непринимающим.

В качестве меры трудоемкости решения задачи I в форме распознавания на НМТ рассматривается величина

$$t_T(I) = \min_{U \in \{U\}} t_T(I,U),$$

Оглавление.

где минимум берется по всем принимающим вычислениям.

Если для некоторой массовой задачи Z НМТ допускает ее индивидуальную задачу I тогда и только тогда, когда I имеет ответ "да", то говорят, что НМТ допускает язык L_Z .

Класс *NP* - это класс языков, допускаемых НМТ за полиномиальное время. (То есть эта НМТ, во-первых, допускает язык, а, во-вторых, время ее работы ограничено полиномом.)

Теорема. Если $Z \in NP$, то существует такой полином p, что Z может быть решена на детерминированной МТ за время $O(2^{p(n)})$.

Доказательство. Пусть T - НМТ, решающая Z за время q(n). То есть для каждого I найдется отгадка U(I) такая, что после ее записи НМТ работает уже как обычная МТ, а число тактов работы не превосходит q(n). Ясно, что число символов самой отгадки (ее нужно прочесть в процессе решения) не может превосходить q(n).

Пусть k - число символов в алфавите НМТ. Тогда всего нужно рассмотреть $k^{q(n)}$ отгадок. Построим такую МТ, которая работает также, как обычная головка нашей НМТ, а на входной ленте у нее записаны все $k^{q(n)}$ отгадок. Она поочередно просматривает отгадки. Если хотя бы на одной она останавливается в состоянии, то вычисление завершается. Временная сложность не превосходит $q(n)k^{q(n)}$, что при надлежащем выборе полинома не превосходит $O(2^{p(n)})$.

9.2. Сводимость задач

9.2.1. Смысл сводимости

Мы уже видели, что существует определенный качественный разрыв между алгоритмами полиномиальной и экспоненциальной сложности. Но *сложность алгоритма решения задачи* еще не полностью характеризует *сложность решения задачи*. Действительно, для одной и той же задачи могут быть построены алгоритмы решения разной сложности. С эти связана одна из самых важных проблем теории сложности - получение "хороших" нижних оценок сложности задачи.

Если для некоторой задачи Z построен алгоритм сложности O(f(n)), то можно говорить, что сложность решения задачи не превосходит O(f(n)). Таким образом, верхние оценки сложности

решения задачи могут быть получены конструктивным образом с помощью построения конкретных алгоритмов. Нижние оценки - это утверждения следующего типа: "Задача Z не может быть решена никаким алгоритмом со сложностью, меньшей O(g(n))". Эти оценки нельзя получить, построив конкретный алгоритм. (Может быть, это просто плохой алгоритм, а в будущем кто-то построит хороший.) Эти оценки строятся путем анализа свойств задачи, возможностей кодировок условия и пр. Тривиальная нижняя оценка - длина входа задачи. Но выше мы видели, что проблемы кодировки входа достаточно тонкие. Хотя в большинстве случаев здесь можно все прояснить.

Лишь для небольшого количества задач верхние и нижние оценки сложности близки. Например, сложность сортировки n чисел равна O(n*log*n). То есть, известен конкретный алгоритм, который с такой сложностью сортирует n чисел. Кроме того, доказано, что эту задачу нельзя решить быстрее, чем за O(n*log*n) шагов.

Для многих полиномиальных задач верхняя и нижняя оценки различаются степенью полинома. Это может быть достаточно существенное различие, но в силу сказанного выше оно все же не столь критично как различие экспоненциальной верхней оценки, например, от линейной нижней. А это, к сожалению, достаточно типичная ситуация для многих задач дискретной оптимизации.

Но кроме разделения задач на два класса: с полиномиальной и экспоненциальной верхней оценкой сложности,- хотелось бы, в свою очередь, более детально разобраться со вторым классом.

Вот для этого и служит понятие *сводимости*. Это, грубо говоря, установление некоторой связи между парой задач. Смысл этой связи иллюстрирует следующий пример. Пусть известно две задачи X и Y. Задача X сформулирована 200 лет назад и с тех пор ничего лучшего, чем экспоненциальный алгоритм для нее не построено. Задача Y сформулирована вчера, и мы хотим приложить максимум усилий для ее эффективного решения. Под таким решением мы подразумеваем полиномиальный алгоритм.

Предположим теперь, что существует некоторое понятие сводимости. Если эта сводимость так устанавливается между X и Y, что наличие полиномиального алгоритма для Y влечет за собой существование полиномиального алгоритма для X, то интуитивно ясно, что подобный факт существенно повлияет на наш оптимизм в вопросе эффективного решения задачи Y и, возможно, после здравого рассуждения мы ограничимся эвристиками и переборными алгоритмами для нашей новой задачи.

Оглавление.

Было введено несколько определений сводимости.

9.2.2. Полиномиальная сводимость

Рассмотрим класс задач NP. Здесь можно говорить о сводимости языков. Формальное определения полиномиальной сводимости следующее.

Определение. Говорят, что язык $L_1 \subseteq A^*$ сводится к языку $L_2 \subseteq B^*$, если существует такое отображение $f: A^* \to B^*$, что выполняются два условия.

1. Функция f вычисляется за полиномиальное время. (Например, существует МТ, которая за полиномиальное число тактов вычисляет эту функцию.)
2. Для любого входа $I \in A^*$ соотношение $I \in L_1$ выполняется тогда и только тогда, когда выполняется соотношение $f(I) \in L_2$.

Сводимость языков обозначается $L_1 \propto L_2$.

Вначале приведем два простых примера.

Пример. Задача "гамильтонов цикл" сводится к задаче "коммивояжер".

Схема решения. Матрица весов ребер A задачи *коммивояжер* строится из матрицы инциденций графа G задачи *гамильтонов цикл* по правилам: наличие ребра в G соответствует единице в A, а его отсутствие соответствует двойке. Число B для задачи коммивояжер определяется равным числу вершин в G.

Определим задачу «3-КНФ» как частный случай задачи «КНФ-выпонимость», в котором каждая КНФ содержит не более трех литералов.

Теперь приведем два очевидных свойства соотношения "сводимость".

Лемма. Если $L_1 \propto L_2$ и $L_2 \in P$, то $L_2 \in P$.

Доказательство. Смысл этого утверждения в том, что сведения некоторой новой задачи к известной полиномиально разрешимой позволяет утверждать, что эта новая также полиномиально разрешима.

Мы приводим доказательство этого утверждения, так как оно конструктивно и, по сути дела, описывает алгоритм решения новой задачи.

Оглавление.

Пусть A и B алфавиты языков L_1 и L_2, а функция f: A*→B* осуществляет полиномиальную сводимость L_1 к L_2.

Так как $L_2 \in P$, то существует МТ, которая за полиномиальное число тактов решает эту задачу. Обозначим эту машину через M_2. Через M_f обозначим машину Тьюринга, которая за полиномиальное число тактов вычисляет эту функцию f. Верхние оценки сложности решения обозначим через $p_2(n)$ и $p_f(n)$, где p_2 и p_f - некоторые полиномы.

Построим теперь МТ, которая решает первую задачу (допускает язык L_1). Пусть задан некоторый вход $I \in L_1$ (условие индивидуальной задачи). Сначала с помощью M_f преобразуем его во вход f(I) второй задачи. Затем с помощью M_2 выясняем справедливость соотношения $f(I) \in L_2$. Так как для любого входа $I \in A^*$ соотношение $I \in L_1$ выполняется тогда и только тогда, когда выполняется соотношение $f(I) \in L_2$, то требуемый алгоритм построен. Время работы ограничено $O(p_2(p_f(I)) + p_f(I)))$, что является полиномом, от n.

Лемма доказана.

Мы видим, что в построенном алгоритме по одному разу работают обе машины Тьюринга. В случае программы на некотором современном языке программирования это аналогично ситуации, когда некоторая программа составляется путем последовательного однократного выполнения двух других программ.

Соотношение полиномиальной сводимости транзитивно, то есть справедливо следующее соотношение.

Лемма. Если $L_1 \propto L_2$ и $L_2 \propto L_3$, то $L_1 \propto L_3$.

Доказательство этого утверждения очевидно.

9.2.3. Сводимость по Тьюрингу

Это иной подход к понятию сводимости. Заметим, что в оригинальной работе Кука использовался именно он.

Этот тип сводимости касается более широкого класса задач, чем задачи в форме распознавания.

В задачах оптимизации Z для каждого входа I условия задачи определяют множество конечных объектов $S_Z(I)$, которые являются ее решениями. Так, в задаче "гамильтонов цикл" это некоторый гамильтонов цикл, а в задаче "коммивояжер" на минимум - это гамильтонов цикл минимального веса. Первая из этих задач является задачей в форме распознавания, а

Оглавление.

вторая таковой не является. Но даже при решении первой как задачи в форме распознавания ответом будет только утверждение о наличии или отсутствии гамильтонова цикла, в то время как ее же решение в оптимизационной форме предполагает в качестве ответа предъявление требуемого гамильтонова цикла в случае его наличия.

Считается, что некоторый алгоритм решает задачу Z, если он выдает некоторый $s \in S_Z(I)$, если множество $S_Z(I)$ не пусто. В противном случае он выдает некоторый условленный ответ, например, "нет".

Выше мы видели, что для формальных рассуждений удобно представлять задачу в форме распознавания в виде некоторого языка. По той же причине рассматриваемые здесь задачи удобно представлять в виде известного формального объекта, называемого *словарным отношением*.

Пусть имеется алфавит A. Обозначим через A^+ множество всех непустых слов этого алфавита. Словарным отношением над алфавитом A называется бинарное отношение $R \subseteq A^+ x A^+$.

Со словарным отношением R свяжем язык L над A. Определим некоторое словарное отношение $R=\{(I,s): I \in A^+ \text{ и } I \in L\}$. Здесь s- любой фиксированный символ алфавита A.

Говорят, что функция $f: A^* \rightarrow A^*$ реализует словарное отношение R тогда и только тогда, когда для каждого $I \in A^+$ имеет месть $f(I)=\Lambda$, если не существует $y \in A^+$ такого, что$(I,y) \in R$; в противном случае $f(I)=y$ для некоторого $y \in A^+$, $(I,y) \in R$.

МТ разрешает отношение R, если она вычисляет функцию f, которая реализует R.

Если раньше речь шла только о кодировании условия задачи Z, то теперь схема кодирования включает как кодирование входа I, так и кодирование решения $S_Z(I)$. Тогда задаче Z можно сопоставить словарное отношение $R=\{(x,y)\}$, где x - код входа индивидуальной задачи, а y -код некоторого ее решения. (Как и раньше, чтобы не загромождать изложение мы не будем различать сам вход и его код.)

Сводимость по Тьюрингу базируется на определенном выше понятии оракульной МТ.

Определение. Пусть R и R' - два словарных отношения над A. Будем говорить, что R сводится по Тьюрингу к R', если существует оракульная машина Тьюринга M с входным алфавитом A такая, что для любой функции $g: A^* \rightarrow A^*$, реализующей словарное отношение R', релятивизированная программа M_g будет полиномиальной программой ОМТ, а функция, вычисляемая программой M_g, будет реализовывать отношение R.

Обозначим этот тип сводимости как $R \propto_T R'$. Как и в случае полиномиальной сводимости имеет место транзитивность.

Лемма. Если $L_1 \propto_T L_2$ и $L_2 \propto_T L_3$, то $L_1 \propto_T L_3$.

Данное понятие сводимости относится уже не только к задача в форме распознавания, т.е. оно является более широким и, возможно, позволяет анализировать сложность задач за пределами класса NP задачи.

Словарное отношение R назовем *NP-трудным*, если существует NP-полный язык L (представленный как словарное отношение), который сводится по Тьюрингу к R, т.е. $L \propto_T R$. Более неформально NP-трудная задача определяется как задача, к которой по Тьюрингу сводится некоторая NP-полная задача.

Оглавление.

9.3. Теорема Кука

Само понятие NP-полноты - инструмент исключительно описательный. Из приведенных выше утверждений легко получить следующую теорему.

Теорема. Если $L_1 \propto L_2$ и L_1 является NP-полной задачей, тогда $L \in$ также NP-полная задача.

Это утверждение позволяет гипотетически разбить весь класс NP на части. То есть выделить в нем некоторое множество сравнительно простых задач. Это класс P. И множество задач самых сложных – класс NPC. Это множество NP-полных задач.

Пусть, например, появилась некоторая новая задача Z. Мы пытаемся оценить ее сложность. Если мы придумали полиномиальный алгоритм ее решения, то она относится к классу P. Если наши усилия по построению такого алгоритма долго не увенчиваются успехом, то мы можем попытаться доказать с помощью сформулированной теоремы NP-полноту задачи Z. Если это удается, то можно утверждать, что новая задача в некотором смысле не проще многих известных задач, NP-полнота которых уже доказана.

Но доказательство NP-полноты можно проводить разными способами. Пока нам ясно два: использовать сформулированную только что теорему и технику сведения одной конкретной задачи к другой или непосредственно исходить из определения. Первый способ кажется предпочтительней (да так, наверное, и есть). Но для применения теоремы нужно иметь хотя бы одну NP-полную задачу.

Вот в этом и основной смысл теоремы Кука. Он доказал, пользуясь только определением, NP-полноту одной конкретной задачи. Всюду в дальнейшем пустой символ будем обозначать через Λ.

Теорема Кука. *Задача о выполнимости* (ЗВ) NP-полна.

Мы приведем схему доказательства этой теоремы, взятую из лекций А.Разборова.

По определению NP-полноты мы должны доказать два факта.

1. ЗВ лежит в NP.
2. Любая задача Z из NP полиномиально сводится к ЗВ.

Для доказательства первого факта мы должны построить НМТ, которая за полиномиальное время решает ЗВ. Это очевидно. Действительно, вход задачи - КНФ ϕ от p переменных $x_1,\ldots,x_p$, заданная в

Оглавление.

Гордеев Э.Н. «Введение в теорию сложности алгоритмов».

виде конъюнкции из m конъюнкций. Если она обращается в единицу на некотором наборе значений переменных $x_1=a_1,\ldots,x_p=a_p$, то угадывающая головка данный набор длины p напишет. Подстановка в формулу и проверка требует полиномильного по длине входа времени.

Доказательство второго факта строится по следующему принципу. По входу w задачи Z строится вход задачи ЗВ. Это некоторая КНФ ϕ_w, которая обращается в единицу тогда и только тогда, когда на слове w задача Z имеет ответ "да".

Так как задача Z (язык L_Z) лежит в NP, то имеется НМТ $T_Z=\{S, A, F, q_0, \delta\}$, решающая эту задачу (допускающая язык L_Z) за полиномиальное от длины входа $|w|=n$ время. Пусть p(n) - полином, ограничивающий число тактов работы NMT. Для простоты индексации будем считать, что подобрано подходящее k, такое что $p(n)<n^k$.

Идея состоит в том, чтобы смысл данной фразы записать в виде логической формулы. Наглядно представить себе подобную формулу позволяет подробное описание всех шагов вычисления T_Z на входе w. Их легче всего представить в виде таблицы, строки которой - конфигурации работы T_Z на стадии проверки. Такая таблица называется допускающей таблицей и имеет вид.

Номера конфигураций/ячеек	$-n^k$	...	v+1	v	...	-1	0	1	...	n	n+1	...	n^k
1	Λ	…	Λ	a_v	…	a_1	q_0	w_1	…	w_n	Λ	…	Λ
2													
...													
n^k													

Ее размеры очевидным образом ограничены условием полиномиального времени работы: количество столбцов не превосходит времени работы, а число строк просто равно числу тактов работы НМТ. В первой строке написана отгадка $a_1,\ldots,a_v$ и код слова индивидуальной задачи (слово языка L_Z) $w_1,\ldots,w_n$. В следующей строке приведена конфигурация НМТ после первого такта ее

Оглавление.

работы и т.д. Ясно, что построение такой таблицы потребует полиномиального по длине входа n времени.

Грубо говоря, эта таблица будет теперь входом задачи о выполнимости. То есть, по ней мы построим некоторую КНФ, которая выполнима тогда и только тогда, когда данный вход описывает именно таблицу допустимости.

Логика же здесь простая. Должны быть справедливы одновременно четыре утверждения.

1. В каждой клетке таблицы записан ровно один символ, причем это или пустой символ, или буква алфавита A, или символ (номер) состояния из S. (Пусть это условие можно записать в виде некоторой конъюнкции ϕ_0).
2. В первой строке записана начальная конфигурация, например, в том виде, что на рисунке выше. (Пусть это условие можно записать в виде некоторой конъюнкции ϕ_{start}).
3. В последней строке записана некоторая допускающая конфигурация, т.е. машина остановилась в состоянии q_y. (Пусть это условие можно записать в виде некоторой конъюнкции ϕ_{accept}).
4. Каждая следующая строка в таблице получается вследствие допустимого перехода МТ. Этот переход осуществляется путем выполнения некоторой команды $q_ia_j \rightarrow q_ra_t\{Z\}$,заданной функцией переходов δ.(Пусть это условие можно записать в виде некоторой конъюнкции $\phi_{computs}$).

В результате получаем КНФ

$$\phi_w = \phi_0 \& \phi_{start} \& \phi_{accept} \& \phi_{computs}.$$

Если она обращается в единицу, то выполняются все вышеприведенные четыре условия, которые влекут за собой допустимость некоторого корректного вычисления НМТ на индивидуальной задаче w некоторой массовой задачи Z. Набор значений переменных в задаче о выполнимости, на котором КНФ обращается в единицу, и даст нам содержимое допускающей таблицы, которая и описывает данное корректное вычисление.

Оглавление.

С другой стороны, если мы имеем некоторое корректное вычисление НМТ на индивидуальной задаче w некоторой массовой задачи Z, то оно может быть представлено в виде допускающей таблицы, по которой строится КНФ

$$\phi_w = \phi_0 \& \phi_{start} \& \phi_{accept} \& \phi_{computs}.$$

И просто по построению эта КНФ должна быть выполнима на том входе, который задает содержимое допускающей таблицы.

Остается построить за полиномильное время четыре вышеупомянутые конъюнкции.

В качестве переменных рассмотрим множество булевских переменных

$$Var = \{x_{ij\sigma},\ i=-n^k,\ldots,n^k;\ j=-n^k,\ldots,n^k;\ \sigma \in S \cup A\}.$$

Эти переменные обращаются в единицу, если на пересечении i-й строки и j-го столбца допускающей таблицы стоит символ σ.

Покажем, что первая из четырех конъюнкций ϕ_0 выражается в следующем виде.

$$\phi_0 = \mathop{\&}_{-n^k \le j \le n^k;\, 1 \le i \le n^k} \left[\left(\bigcup_{\sigma \in S \cup A} x_{ij\sigma} \right) \& \left(\mathop{\&}_{\sigma,\rho \in S \cup A;\, \sigma \ne \rho} \left(\bar{x}_{ij\sigma} \cup \bar{x}_{ij\rho} \right) \right) \right]$$

Знак конъюнкции означает, что логическое условие в скобках должно выполняться для каждой клетки таблицы. Первое из условий в круглых скобках соответствует требованию наличия в каждой клетки таблицы хотя бы одного символа (из области значений введенного выше множества переменных *Var*). То требование, что в клетке должно быть не более одного символа, записано в виде логического выражения во второй круглой скобке формулы для ϕ_0.

Так как число состояний и число символов алфавита являются константами, то длина выражения в скобках от n не зависит. Поэтому длина всей конъюнкции ϕ_0 равна $O(n^{2k})$.

Следующая конъюнкция ϕ_{start} выражается формулой

$$\phi_{start} = \mathop{\&}_{-n^k \le j \le -v+1} x_{1j\Lambda} \mathop{\&}_{-v \le j \le -1} x_{1ja_{-j}} \mathop{\&}_{1 \le j \le n} x_{1jw_j} \& x_{11q_0}.$$

Оглавление.

Здесь все понятно. Это просто логическая запись требования к содержимому первой строки таблицы. Для индивидуального входа w и отгадки a оно должно быть именно таким, как показано на рисунке выше. Длина всей конъюнкции ϕ_{start} равна $O(n^k)$.

Что касается конъюнкции ϕ_{accept} , то она выражается формулой

$$\phi_{accept} = \bigvee_{-n^k \le j \le n^k; \sigma \in F} x_{n^k j \sigma}.$$

Смысл в том, что в последней строке допускающей таблицы должна стоять конфигурация, содержащая символ конечного состояния. Длина всей конъюнкции ϕ_{accept} равна $O(n^k)$.

Самым сложным является вычисление конъюнкции $\phi_{computs}$. Мы не будем выписывать его во всей строгости, приведем лишь общую идею формулы. Как уже говорилось выше, эта конъюнкция должна быть формальной записью того условия, что каждая следующая строка в таблице получается вследствие допустимого перехода МТ. А сам переход осуществляется путем выполнения некоторой команды $q_i a_j \to q_r a_t \{Z\}$,заданной функцией переходов δ. Обозначим через $\phi_{i(i+1)}$ условие того, что правильно осуществлен переход между соседними конфигурациями. Тогда $\phi_{computs}$ выглядит следующим образом

$$\phi_{computs} = \bigwedge_{1 \le i \le n^k} \phi_{i(i+1)}.$$

Для нахождения $\phi_{i(i+1)}$ заметим следующее. Две соседние строки допускающей таблицы различаются не более чем в трех клетках. То есть все изменения сосредоточены в "окошке" таблицы" следующего вида

(i,j-1)	(i,j)	(i,j+1)
(i+1,j-1)	(i+1,j)	(i+1,j+1)

Первая строка таблицы правильно заполнена в силу условия ϕ_{start}. Заполняем таблицу, начиная со второй строки. Поэтому в паре i-й и (i+1)-й строк проверка того условия того условия, что каждая следующая строка в таблице получается вследствие допустимого перехода МТ, сводится к правильности проверки заполнения (i+1)-й строки. Для этого "проходим" всю пару вышеупомянутыми окошками.

Оглавление.

Если в верхней части окошка нет символа состояния, то в нижней части символы алфавита совпадают с верхними. Как только символ состояния встретился, читаем стоящий рядом символ алфавита. Так как число состояний, число букв алфавита и число правил переходы - постоянные величины, то логическое условие, определяющее правильность перехода между соседними конфигурациями, не зависит от n. Действительно, вспомним теорему о полноте базиса из трех булевых функций: &, $\cup$ и $\neg$. В качестве простого упражнения можно записать упомянутое условие через эти функции.

Но для построения конъюнкции $\phi_{i(i+1)}$ все равно нужно пройти всю строку. Поэтому и длина всей конъюнкции $\phi_{computs}$ равна $O(n^{2k})$.

Таким образом, все четыре требуемые конъюнкции построены за полиномиальное время.

Теорема доказана.

Итак, у нас появилась первая NP-полная задача. Для доказательства NP-полноты следующей задачи можно либо повторить что-то похожее на только что доказанную теорему, либо использовать понятие сводимости.

9.4. Структура класса NP и некоторые выводы

Появление теории NP-полноты дало в руки инженерам и исследователям мощные рекомендации по методологии решения математических задач и инженерных проблем.

Теорема Кука и методика сведения задач привели к появлению в конце 1970-х годов нескольких сотен задач, про которые было доказано, что они являются NP-полными. (См, например, книгу [2]).

Выделим просто на основе здравого смысла в методике решения проблемы (обозначим ее ZZ) несколько этапов, интересующих нас.

1. *Этап первичной формулировки*. Инженер (или математик) сталкивается с некоторой пока неформализованной проблемой, которую нужно решить. Например, составление расписания занятий; составление графика (метода) разлива металла в заданный набор форм; прогноз затрат по данному бизнес-плану; составление наиболее интересного месячного путешествия по Европе и т.п.

Оглавление.

2. *Формулировка задачи*. В данной неформализованной проблеме выделяется набор критичных параметров и целей (вопросов), на основе которых можно сформулировать задачу Z_0. Например, выделив условия и параметры расписания занятий, можно сформулировать конкретную проблему из *теории расписаний*; задав размеры форм и стоимостные характеристики деталей, из ни получаемых, можно прийти к *задаче о рюкзаке*; прогноз затрат по плану может привести к *задаче линейного программирования*; а путешествие по Европе к *задаче коммивояжера*.

3. *Подбор алгоритма*. Зная задачу, исследователь подбирает алгоритм ее решения. Здесь возможны. Например, следующие варианты. ***Вариант 3.1***. В книгах, справочных материалах, Интернете алгоритм $A(Z_0)$ найден. ***Вариант 3.2***. В книгах, справочных материалах, Интернете алгоритм не найден. Исследователь самостоятельно строит некоторый алгоритм $A(Z_0)$. ***Вариант 3.3***. В книгах, справочных материалах, Интернете алгоритм не найден. Исследователь самостоятельно пытается строить некоторый алгоритм, но безуспешно.

4. *Анализ алгоритма и возможная переформулировка задачи.* В распоряжении исследователя есть алгоритм $A(Z_0)$ и задача Z_0, а в случае варианта 3.3 только одна задача. Если параметры задачи и алгоритма укладываются в ресурсы исследователя, то процесс завершается. В противном случае исследователь может с помощью теории NP-полноты определить класс сложности задачи. Если задача полиномиально разрешима и полиномиален, то нужно позаботиться об увеличении вычислительных ресурсов. Это увеличение, как правило, вполне реально и легко оцениваемо. Далее – проблема решена. Если задача находится в классе NPC, то это означает, что десятки лет выдающиеся инженеры и математики безуспешно пытались ее решить. При этом увеличение ресурса ситуацию не спасет, поэтому совершенно естественно (и легко оправдываемо перед руководством) поступить одним из следующих способов. ***Вариант 4.1***. Переформулировать требование к вопросу задачи. Например, искать не точное решение, а приближенное или допустить эвристические методы или методы направленного перебора (ветвей и границ) с элементами эвристик. ***Вариант 4.2.*** Из задачи Z_0 получить задачу Z_1. Возможно, неправильно оценены значимости целей и параметров на этапе первичной формулировки и можно немного изменить их соотношение. Это приведет к изменению формулировки задачи втором этапе. Например, вместо задачи коммивояжера можно получить задачу о кратчайшем остове, а вместо задачи ЦЛП задачу ЛП. ***Вариант 4.3.***

Оглавление.

Из задачи Z_0 невозможно получить задачу Z_1. Анализ значимости целей и параметров на этапе первичной формулировки не позволяет на уровне самого исследователя изменить их соотношение. В этом случае инициативное исследование прекращается, если же задача является производственной, то для дальнейшего принятия решения привлекаются вышестоящие инстанции (и при разговоре с ними теория NP-полноты – решающий аргумент) в случае, если они обладают возможностью неограниченно увеличить ресурсы или кардинально поменять первичную формулировку задачи. А если не удается определить принадлежность задачи. Например, известно, что она лежит в NP, а про ее принадлежность к P или NPC ничего нельзя сказать. Это наиболее тяжелый вопрос, на который проливает некоторый свет следующая теорема и комментарии к ней.

Обозначим NPI=NP\(NPC$\cup$P). Если P$\neq$NP, то класс NPI может быть непустым.

Теорема (Ладнера). *Пусть B – некоторый рекурсивный язык, такой что $B \notin P$ (т.е. слова этого языка не распознаются за полиномиальное время на детерминированной машине Тьюринга). Тогда существует распознаваемый за полиномиальное время язык $D \in P$, такой, что язык $A=D \cap B$ не принадлежит P. При этом $A \propto B$, а $B \propto A$.*

Доказательство этого утверждение можно найти в (Ladner R.E. On the structure of polynomialen time reducibility.- J. Assoc. Comput. Mach. 1975, v. 22, p. 155-171.)

Эта теорема имеет практическое применение, если P$\neq$NP. Пусть B – задача «Гамильтонов цикл», тогда теорема утверждает, что найдется такой класс распознаваемых за полиномиальное время графов, что на этом классе графов задача «Гамильтонов цикл», не имея полиномильного алгоритма решения, в то же время не является NP-полной.

Если P$\neq$NP, то теорема утверждает непустоту NPI класса и дает представление о его структуре. Он состоит из бесконечной совокупности классов эквивалентности языков, каждый из которых «чуть-чуть» сложнее другого. Кроме того в нем есть пары языков, ни один из которых полиномиально не сводится к другому!

Теорема. К сожалению, не дает конструктивных методов построения «естественных» задач из класса NPI. Поэтому с 1971 года имеется достаточно интересная ситуация: существует очень небольшое, постоянно сужающееся множество естественных задач, про которые известно, что они не лежат ни в NPC, ни в P. В то же время не известна их принадлежность и к классу NPI.

Оглавление.

Почему сужающее? Потому, что постепенно задачи из этого множества куда-то определяются. Когда-то там лежала задача ЛП, но Л.Г.Хачиян доказал, что она лежит в P. Задача ***изоморфизм графа***, ***простота числа*** (и ряд других из теории чисел) лежат там до сих пор.

10. Иерархия сложности

10.1. Классы NP и co-NP

Для любой Z задачи из NP можно сформулировать дополнительную задачу (язык) Z'. Если язык L_Z образуют все слова, являющиеся условием индивидуальных задач с ответом "да", то язык $L_{Z'}$ образуют все слова, являющиеся условием индивидуальных задач с ответом "нет". Совокупность таких "дополнительных" задач и составляет класс co-NP. Аналогично для класса P может быть определен класс co-P.

Вообще говоря, дополнение множества слов, являющихся условиями индивидуальных задач с ответом "да", содержит два типа слов: слова, являющиеся кодами условиями индивидуальных задач с ответом "нет", и слова, не являющиеся условиями задач.

Оглавление.

Предполагается, что последние можно отсеять с помощью каких-то вспомогательных "простых" процедур, например, синтаксических проверок и т.п.

Для любой задачи из класса P дополнение можно получить заменой заключительного состояния (q_{no} заменяем на q_{yes} и наоборот). Так как нет никакой угадывающей головки, то на любом корректно заданном условии задачи МТ остановится. Таким образом, мы автоматически получаем из МТ, решающей задачу Z, МТ для решения Z'. Поэтому классы P и co-P совпадают.

Пример. Рассмотрим задачу "Связность графа". Ее входом является неориентированный граф, а вопрос состоит в проверке его связности. Пусть граф G=(V,E) задан списками соседей вершин.

Рассмотрим следующий алгоритм. Берем пустое множество W.

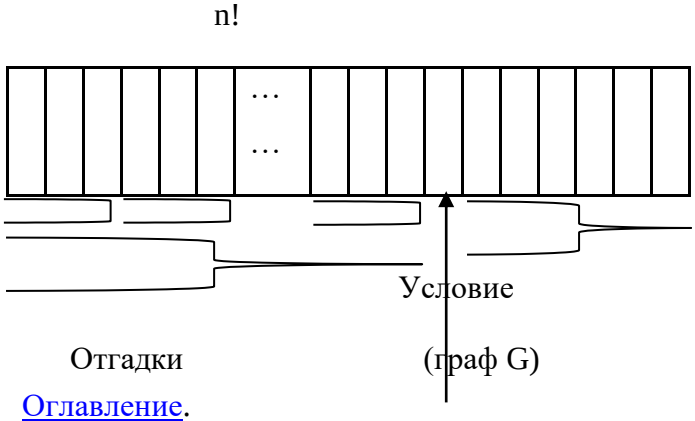
1 шаг. На первом шаге включаем в него некоторую вершину v. Она считается помеченной, но не просмотренной. Переходим ко второму шагу.

2 шаг. Включаем в W всех соседей всех непросмотренных вершин. После этого только вновь включенные вершины считаются помеченными, но непросмотренными. Переходим к шагу 3.

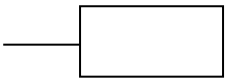
3 шаг. Если W=V, то ответ "да". Если W≠V и непросмотренных вершин нет, то ответ "нет". В остальных случаях переходим к шагу 2.

Ясно, что шагов не более n. И на каждом шаге не более m проверок. Очевидно, что данный алгоритм решает как задачу с вопросом, связен ли G, так и задачу с вопросом, является ли G несвязным.

Для NP и co-NP ситуация иная. Например, обратная к задаче "гамильтонов цикл" задача состоит в ответе на вопрос: "Правда ли, что в данном графе нет гамильтонова цикла?" Как здесь может выглядеть подсказка? На сегодняшний день единственной возможной подсказкой представляется выписывание всех возможных гамильтоновых циклов и проверка их отсутствия в данном графе.



Оглавление.

(все возможные циклы) 

Но такой список циклов имеет экспоненциальную длину.

Поэтому вопрос о соотношении NP и co-NP является открытым.

Теорема 9.2. Если существует NP-полная задача Z такая, что ее дополнение лежит в NP, то

NP=co-NP.

Доказательство строится конструктивно с помощью композиции машин Тьюринга.

Пусть дополнение Z' некоторой NP-полной задачи Z лежит в NP. Покажем, что тогда дополнение W' любой задачи W лежит в NP. По определению полиномиальной сводимости функция f, осуществляющая эту сводимость, например, задачи W к задаче Z одновременно осуществляет полиномиальную сводимость задачи W' к задаче Z'. Функция f (одна МТ) вычисляется за полиномиальное время, проверка принадлежности Z' к NP (другая МТ, уже недетерминированная) тоже вычисляется за полиномиальное время. Поэтому проверка принадлежности W' к NP осуществляется путем последовательной комбинации этих двух машин, т.е. может быть проведена за полиномиальное время на недетерминированной МТ.

10.2. Классы P-SPACE и NP-SPACE.

10.2.1. Классы сложности P-SPACE и NP-SPACE

Рассмотрим многоленточную машину Тьюринга, в которой выделена входная и выходная ленты. С входной ленты можно только считать условие задачи, а на выходную - только записать ответ. Кроме того, есть одна или несколько рабочих лент. В качестве меры сложности решения будем теперь брать не время, а память.

Будем говорить, что Z принадлежит классу P-SPACE, если существует детерминированная МТ, которая решает задачу так, что число задействованных для решения ячеек ленты (а это ячейки выходной и рабочих лент) ограничено полиномом от длины входа задачи.

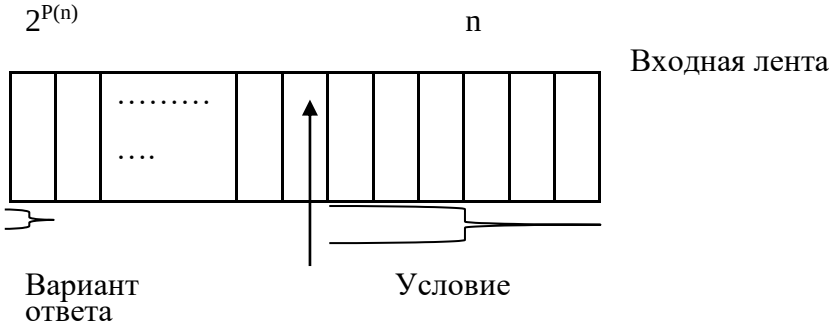
Оглавление.

Будем говорить, что Z принадлежит классу NP-SPACE, если существует недетерминированная МТ, которая решает задачу так, что число задействованных для решения ячеек ленты ограничено полиномом от длины входа задачи.

Если алгоритм (например, МТ) работает полиномиальное время, то он, очевидно, использует только полиномиальную память. Поэтому P⊆PSPACE. Но оказывается, что справедливо и более сильное утверждение.

Теорема 9.3. NP⊆PSPACE.

Берется произвольная задача Z из NP и НМТ с алфавитом A, решающая эту задачу за полиномиальное время p(n). Обозначим эту машину через T_Z. Условие полиномиальности влечет за собой полиномиальность длины слова, записанного угадывающей головкой.



Вспомним определение НМТ и смоделируем на его основе уже детерминированную МТ, работу которой представим следующим образом. Входом является условие индивидуальной задачи Z. В МТ входит генератор всех слов в алфавите A, длина которых не превосходит p(n). Таких слов $|A|^{p(n)}$. После генерации очередного слова работает T_Z.

Оглавление.

Затем слово стирается и на его место записывается очередное слово-отгадка. Такая машина работает уже детерминированно. Время ее работы экспоненциально, но пространство на ленте полиномиально ограничено.

Совершенно аналогично можно доказать и следующее утверждение.

Теорема 9.4. Co-NP⊆PSPACE.

Более того, вспомним задачу "K-е по порядку подмножество". Ее принадлежность к NP невыяснена. Но к классу PSPACE она принадлежит. Построим МТ, которая поочередно просматривает все подмножества множества {1,...,n} и хранит счетчик числа просмотренных подмножеств, вес которых не больше B. В процессе работы каждое следующее проверяемое подмножество записывается на место предварительно стертого предыдущего.

С помощью понятия полиномиальной сводимости можно ввести определение PSPACE-полной задачи. Это такая задача, к которой полиномиально сводятся все задачи из класса PSPACE.

Вообще говоря, между классами P и PSPACE может быть довольно большое различие. Здесь анализ можно проводить в двух направлениях. Рассмотрим их поочередно.

Выше при определении co-NP мы заметили, что можно не рассматривать входные слова, которые не являются условиями задачи, т.е. можно считать, что любое входное слово - это условие задачи с одной из двух ответов "да" или "нет". Это предположение будет использоваться и дальше.

Мы уже проводили аналогию между НМТ и обращением к оракулу. А теперь представим себе, что у нас есть множество входных слов I - условий индивидуальных задач некоторой массовой задачи Z. Кроме того, есть два игрока: белый (w) и черный (b), каждый из которых имеет своего карманного оракула. Они делают по очереди свои ходы $w_1,\ldots$ и $b_1,..$, которые можно представить в виде ответов их оракулов. Число ходов заранее задано. Длина каждого ответа ограничена полиномом от длины входа. Смысл хода белых - получить в результате ответ "да", а черные стремятся к обратному.

Каждую игру из k можно представить в виде последовательности из 2k+1 слова: $I,w_1,b_1,\ldots,w_k,b_k$. (Последний ход черных может отсутствовать!) Каждой такой последовательности однозначно соответствует один из двух ответов "да" или "нет". Множества последовательностей с ответом "да" обозначим через $W(I,w_1,b_1,\ldots,w_k,b_k)$. (То есть, на множестве таких последовательностей задан

Оглавление.

некоторый предикат $W(I,w_1,b_1,\ldots,w_k,b_k)$). Дополнение этого множества обозначим через $B(I,w_1,b_1,\ldots,w_k,b_k)$.

Можно себе представить эту игру как действие, состоящее из двух этапов. На первом этапе готовится входное слово для МТ. Это слово имеет вид

$$I\#w_1\#b_1\#\ldots\#w_k\#b_k.$$

А на втором этапе МТ отрабатывает на этом слове и останавливается в одном из двух состояний: *допускающем* слово и *отвергающем* его.

Можно считать, что оракулы не всемогущи и игроки стараются получить от них информацию, позволяющую уменьшить перебор вариантов при решении задачи. Проверка может быть осуществлена только один раз на основе I и всех ходов (подсказок). Но при этом для любого I задача имеет единственный ответ, то есть либо белые, либо черные при правильных ходах (выигрышной стратегии) гарантируют себе выигрыш. Ниже при доказательстве теоремы это будет конструктивно показано.

Пусть L_w - множество слов I, на которых выигрывают белые, а через L_b- множество слов I, на которых выигрывают черные.

Поясним сказанное на примере аналогии. Эта аналогия не соответствует описываемой схеме, а только **иллюстрирует** ее.

Рассмотрим, например, задачу некоторую задачу на графе. Если мы зададим вопрос: «Правда ли, что в графе есть гамильтонов цикл?» - то получим задачу "*Гамильтонов цикл*". Усложним вопрос: «Правда ли, что в графе есть гамильтонов цикл, но при этом нет клики размером, больше пяти?». Тогда все графы разделятся на два непересекающихся множества: графы с ответом «да» на поставленный вопрос и графы с ответом «нет». При этом, любой заданный граф принадлежит одному из множеств. Если на нем идет игра, то белые победят в случае ответа «да», а черные – в случае ответа «нет». Эту аналогию можно расширить очевидным образом, например, продолжить вопрос: «Правда ли, что в графе есть гамильтонов цикл, при этом нет клики размером, больше пяти, минимальный тест матрицы смежности графа не меньше семи, а сели его ребрам придать единичные веса, то минимальные обход коммивояжера будет не больше двенадцати?» И т.д.

Теперь можно провести следующие аналогии.

Оглавление.

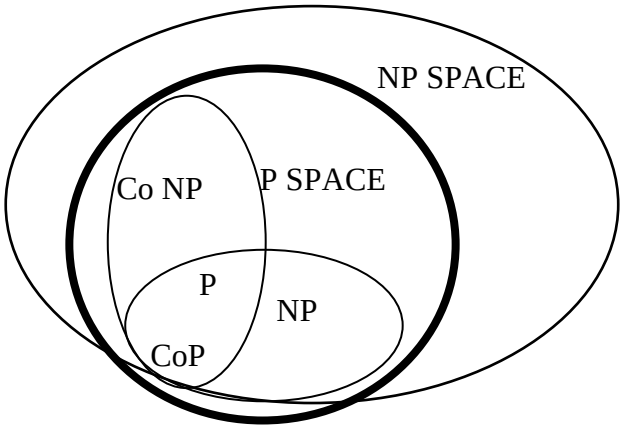
Классу P можно сопоставить множества L_w для игр без ходов (k=0). Тот факт, что P=co-P означает, что классу P можно сопоставить и множества L_b для игр без ходов.

Пусть в игре только белые делают один ход. Тогда получаем аналогию с классом NP.Ход белых - это обращение к оракулу. Если ответом является "да", то оракул предоставит вариант w_1 для проверки. Поэтому L_w - это множество слов I, для которых такой вариант существует. Поэтому NP - это совокупность множеств L_w для всех таких игр из одного хода. В рассматриваемой иерархии класс NP будет соответствовать (совпадать) классу Σ_1.

Пусть в игре только белые делают один ход. Тогда можно получить следующую аналогию с классом co-NP. Ход белых - это обращение к оракулу. Если ответом является "да", то оракул предоставит вариант w_1 для проверки. Но такого варианта оракул может не найти, т.е. что бы он не представил, ответом будет "нет". Поэтому L_b - это множество слов I, для которых варианта ответа «да» не существует. Поэтому co-NP - это совокупность множеств L_b для всех таких игр из одного хода. Назовём этот класс Π_1. Очевидно, что он является дополнением к классу NP=Σ_1 , состоящему из множеств L_w для таких игр.

Если по одному ходу делают белые и черные, то мы уже выходим за рамки простых аналогий. То есть уже формально появляется новый сложностной класс Σ_2, состоящий из множеств L_w для таких игр из двух ходов. Можно представить эту игру так. Существует ход (подсказка оракула) белых такой, что при любом ответном ходе белые выигрывают. Другой сложностной класс Π_2 состоит из множеств L_b для игр из двух ходов. Здесь содержатся все такие условия индивидуальных задач, то при любом ходе белых черные имеют выигрышный ход. Очевидно, что эти классы взаимно дополнительные, т.е. Σ_2=co-Π_2 и наоборот. Аналогично можно ввести классы Σ_k и Π_k для любых k.

Оказывается, что все эти классы лежат в PSPACE:



Оглавление.

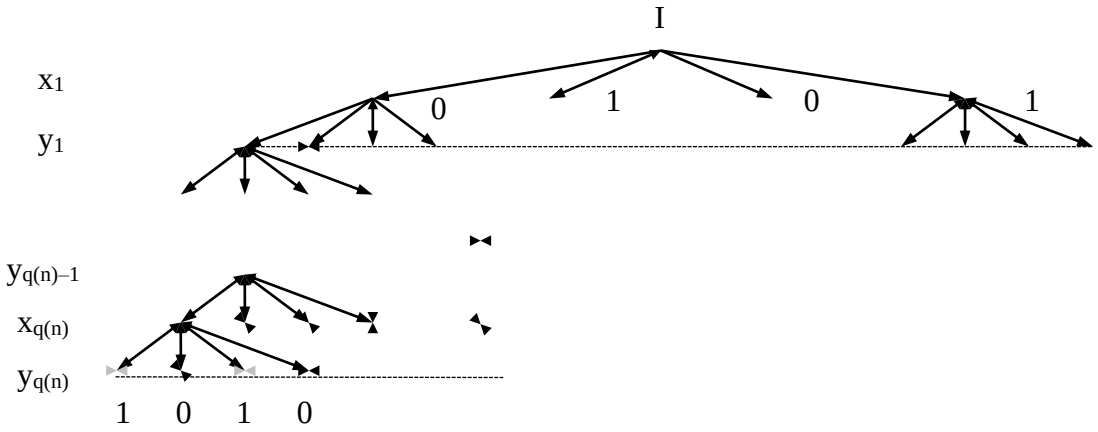
Гордеев Э.Н. «Введение в теорию сложности алгоритмов».

Теорема 9.5. Задача Z лежит в классе PSPACE тогда и только тогда, когда существует игра с полиномиальным от длины входа числом ходов и полиномиально вычислимым результатом такая, что $L_Z=L_w$.

Доказательство. Пусть число ходов ограничено полиномом q(n), а длина каждого записываемого игроками слова ограничена полиномом p(n). Алфавит A конечен |A|=k. Поэтому число всех возможных слов тоже конечно.

Сначала докажем, что игра с полиномиальным от длины входа числом ходов и полиномиально вычислимым результатом лежит в классе PSPACE.

Построим следующее корневое дерево T. Его корнем будет I. Затем последовательно идут ярусы, соответствующие ходам первого и второго игрока. На каждом ярусе расположены вершины, соответствующие всем возможным ходам игрока.



Оглавление.

Каждой вершине можно приписать метку w или b в зависимости от того, кто в данной вершине имеет выигрышную стратегию. По эти меткам можно определить. Кто имеет выигрышную стратегию на всем дереве, т. е. на входе I.

Метки расставляются следующим способом. Каждому листу дерева, а это узлы яруса $y_{q(n)}$, , однозначно сопоставлено значение предиката. Если предикат принимает истинное значение, то пометкой будет w, в противном случае Оглавление пометку b. Затем поочередно рассматриваем узлы яруса $x_{q(n)}$ и вершине приписываем пометку w, если все ее дети имеют пометку w. В противном случае ставим пометку b. По этому правилу помечаем все ярусы, соответствующие ходам первого игрока. Вершины ярусов, соответствующие ходам второго игрока, помечаем следующим образом. Если среди детей есть хотя бы одна, помеченная как b, то пометкой будет b.

По этим пометкам однозначно восстанавливается исход игры на входе I. Таким образом, для решения задачи Z нужно построить МТ, моделирующую вышеприведенный процесс расстановки пометок и вычисления значений предиката на листьях дерева. Число вершин в дереве полиномиально, полиномиально и время одной проверки значения предиката. Поэтому данная МТ требует O(p(n)q(n)) ячеек памяти. В одну сторону теорема доказана.

Пусть теперь есть некоторая задача Z в классе PSPACE. Покажем теперь, что существует игра с полиномиальным от длины входа числом ходов и полиномиально вычислимым результатом такая, что $L_Z=L_w$.

Итак, у нас есть МТ, распознающая на полиномиальной памяти вхождение слова в язык L_Z. Пусть размер памяти p(x). Построим допускающую таблицу такой машины. Как бы не заполняли такую таблицу в конечном алфавите A, |A|=h, с множеством состоянийS, |S|=s, разных таблиц будет не более $(h+s)^{p(x)}p(x)$. Поэтому можно считать, что время работы МТ ограничено экспонентой $2^{q(x)}$, где q(x) — некоторый полином, а допускающая таблица имеет q(x) строк и $2^{q(x)}$ столбцов.

Оглавление.

	1	2	…		$2^{q(x)}$
1					
2					
…					
q(x)					

Игра состоит в следующем. Задан вход I. Белые утверждают, что на этом слове МТ дает ответ "да". Черные хотят проверить. Первым ходом белые записывают состояние МТ после $2^{q(x)}$ тактов работы. Черные своим ходом выбирают один из двух промежутков: от начала до $2^{q(x)-1}$ -го такта или от $2^{q(x)-1}$-го такта до конца. В середине выбранного черными промежутка белые вновь декларируют состояние МТ, черные выбирают одну из половинок этого промежутка. Игра заканчивается за O(q(n)) шагов в тот момент. Когда длина промежутка стала равной единице, т.е. он содержит два последовательных состояния. Если между этими состояниями существует корректный переход в данной МТ, то выиграли белые, в противном случае выигрывают черные.

Если действительно на входе I ответ МТ "да", то белые гарантируют выигрыш, постоянно говоря правду. В противном случае, где-то есть неправильный переход, а черным нужно его указать. Они это обязательно сделают.

Теорема доказана.

Обозначим через TIME(f(n)) класс задач (языков), для которых существует МТ, время работы которой ограничено f(n). В частности, если f(n) экспонента, то соответствующий класс обозначается через EXPTIME.

Заметим, что при доказательстве предыдущей теоремы мы показали, что

$$PSPACE \subseteq EXPTIME.$$

В классе PSPACE существуют PSPACE-полные задачи. Приведем пример.

Теорема 9.6. "Задача выполнимости булевской формулы с кванторами" PSPACE-полна.

Оглавление.

Выше речь шла о задаче выполнимости КНФ, т.е. рассматривалась булевская формула без кванторов. В данном случае речь идет о булевских формулах вида

$$F(x)=Q_1y_1\ldots Q_ny_nH(y_1,\ldots,y_n),$$

где Q_i — это либо квантор всеобщности $\forall$, либо квантор существования $\exists$. По определению, $(\exists xA(x))=(A(0)\cup A(1))$, $(\forall xA(x))=(A(0)\&A(1))$. Задача выполнимости булевской формулы с кванторами (ЗВБФК) состоит в ответе на вопрос о существовании набора значений переменных, на которых формула истинна.

Нам нужно построить сведение любого языка L_Z из PSPACE к задаче ЗВБФК. Берем МТ для задачи Z и строим по ней игру, как это было показано во второй части доказательства предыдущей теоремы. Затем по этой игре строим МТ, проверяющую результат игры. Ходы игроков кодируем булевскими переменными. Тогда, например, наличие выигрышной стратегии у белых задается условием

$$\exists w_{11}\exists w_{12}\ldots\exists w_{1p(|I|)}\forall b_{11}\forall b_{12}\ldots\forall b_{1p(|I|)}f(I,w_{11},\ldots).$$

Здесь $f(I,w_{11},\ldots)$ — результат проверки истинности предиката. Эта проверка может быть представлена как вычисление по булевской схеме. В свою очередь схеме однозначно сопоставляется булевская формула.

Таким образом мы получаем квантифицированную булевскую формулу, которая истинна тогда и только тогда, когда I лежит в L_Z.

Теорема доказана.

Заметим, что при доказательстве теоремы 9.5 класс PSPACE можно заменить на NPSPACE. То есть справедливо следующее утверждение.

Теорема 9.7. Задача Z лежит в классе NPSPACE тогда и только тогда, когда существует игра с полиномиальным от длины входа числом ходов и полиномиально вычислимым результатом такая, что $L_Z=L_w$.

Первая часть доказательства теоремы 9.5.остается просто без изменения. Для доказательства второй части заметим, что допускающая таблица для недетерминированной МТ будет иметь те же размеры, что и аналогичная таблица в теореме 9.5.

Оглавление.

Следствие. PSPACE= NPSPACE.

10.3. Классы P и P/poly.

Рассматриваем, как обычно, задачи в форме распознавания. В начале курса мы говорили о возможности двух принципиально разных подходов к анализу понятия «сложность задачи». Один из них был связан с алгоритмом решения задачи Z, второй – со сложностью СФЭ, реализующих булеву функцию f_Z, значение которой дает ответ на задачу Z в форме распознавания.

Схема доказательства теоремы Кука позволяет легко установить соответствие между классами P и P/poly.

Действительно, для любой задачи из P можно построить допускающую таблицу ее решения на машине Тьюринга так же, как это сделано выше при доказательстве теоремы Кука. По этой таблице мы построим КНФ полиномиальной длины ϕ, а уже для этой КНФ - схему их функциональных элементов, вычисляющую ϕ. Очевидно, что схема будет содержать полиномиальное число вершин. Таким образом, справедливо следующее утверждение.

Теорема 9.1. $P \subseteq P/poly$.

А что будет, если $Z \in P/poly$? Оказывается, что класс P/poly шире класса P. Это связано с тем, что существуют алгоритмически неразрешимые проблемы. В курсе дискретной математики вам о них, наверное, рассказывали, который, по сути, является известным со времен античности *парадоксом брадобрея*. В деревне есть брадобрей, который бреет всех, кто не бреется сам. Попробуйте ответить на вопрос, бреет ли он самого себя. При любом ответе получается противоречие.

Облачим теперь этот парадокс в «математическую форму». *Проблема остановки* (*halting problem*) состоит в том, чтобы ответить на вопрос (мы о нем говорили выше, когда давали определение машины Тьюринга): остановится или зациклится данная машина Тьюринга T на данном входе x? Оказывается, что, как и в случае брадобрея, любой ответ приводит к противоречию. То есть не существует машины Тьюринга, которая решает *проблему остановки*.

Теорема. *Проблема оста*новки алгоритмически неразрешима.

Оглавление.

Доказательство. От противного. Пусть такая машина T* существует. Тогда на входе (T,x) она выдает ответ «да», если машина T останавливается на этом входе, и «нет» в противном случае. (Здесь T – слово в алфавите A, являющееся описанием машины T). Тогда по T* можно построить машину Тьюринга T'(x), которая в случае, если T*(x,x)="да", начинает двигать головку в одну сторону и зацикливается, а в случае T*(x,x)="нет" она останавливается. Что в этом случае будет означать T'(T')? Остановится или нет машина на этом входе? Если «да», то это означает, что T*(T')= «нет», т.е. T' не должна останавливаться на T'. Если «нет», то это означает, что T*(T')= «да», т.е. T' должна останавливаться на T'.

Получили противоречие

Теорема доказана.

Рассмотрим функцию натурального аргумента f(n), принимающую значения 0 или 1. Можно показать, что вычисление такой функции может быть алгоритмически неразрешимой проблемой, т.е. не входит такая задача ни в какой класс сложности, а не только в класс P. Рассмотрим теперь предикат $A_f(x)=f(|x|)$. Для любого фиксированного n предикат равен константе. А константе сопоставляется СФЭ, сложность которой тоже равна константе. Поэтому $A_f(x) \in P/poly$, но его вычисление может быть алгоритмически неразрешимой проблемой.

Конечно, все это связано с тонкостями определений математических объектов и тем, что при вычислении предикатов основная сложность может лежать не в логических операциях, а в вычислении термов, от которых зависит предикат. Это вычисление связано с интерпретацией и допускает наличие в качестве аргумента предиката произвольной, сколь угодно сложновычислимой функции.

Результатом этого рассмотрения является следующая простая теорема.

Пусть задача Z в форме распознавания эквивалентна вычислению булевой функции f.

Теорема. Функция (задача) $f \in P$ тогда и только тогда, когда $f \in P/poly$ и существует машина Тьюринга, которая за полиномиальное от длины входа n время строит СФЭ для f_n.

Для доказательства заметим следующее. Теорема Кука дает конструктивный метод построения СФЭ полиномиального по входу размера за полиномиальное по входу время для функции $f \in P$. Т.е. в этом случае $f \in P/poly$. И наоборот, если $f \in P/poly$ и существует T - машина Тьюринга, которая для каждого

Оглавление.

отдельного n за полиномиальное от длины входа время по n строит СФЭ S_n для f_n. Вычислений значения функции по этой схеме тоже потребует не более, чем полиномиального времени.

Грубо говоря, с точностью до «разницы в определениях» двух подходов: алгоритмического и схемного, можно представлять себе классы P и P/poly достаточно близкими. Про соотношение классов P и NP мы ничего не знаем. А вот на вопрос, как соотносится класс P/poly с классом всех СФЭ, ответ дает теорема Лупанова. Почти все схемы имеют экспоненциальную сложность $2^n/n$, т.е. множество функций (задач в форме распознавания), имеющих СФЭ полиномиального размера , пренебрежимо мало. Этот результат создал иллюзию безысходности в области исследований по алгоритмической сложности в 1960-е годы. На этом фоне результат Кука (1971 год) явился определенным идеологическим прорывом в том смысле, что он обратил внимание исследователей на небезнадежность решения задач, принадлежность которых к классу NPC не удалось доказать после достаточно серьезных усилий квалифицированных специалистов. И хотя таких задач было решено немного (пионером здесь является Л.Г.Хачиян , решивший за полиномиальное время *задачу линейного программирования*), но каждое из таких решений явилось фундаментальным достижением в математике.

10.4. Некоторые результаты

Обозначим через SPACE(f(n)) класс задач (языков), для которых существует МТ, работающая на памяти с объемом, ограниченным f(n).

Аналогично NSPACE(f(n)) класс задач (языков), для которых существует НМТ, работающая на памяти с объемом, ограниченным f(n).

Приведем без доказательства следующее утверждение, уточняющее некоторые из вышеприведенных.

Теорема 9.7. $NSPACE(f(n))\subseteq SPACE(f(n)^2)$.

Оглавление.

11. Подходы к решению NP-полных задач

Мы видели насколько полиномиальная сложность отличается от экспоненциальной. В последнем случае увеличение производительности вычислительной системы мало влияет на максимально возможную размерность решаемой задачи. Если все-таки задачу из NPC решать надо, то на практике используется следующие подходы.

1. Анализируется структура NPC. Оказывается, что в этом классе можно выделить «более легкие» и «более сложные» задачи. И уже в зависимости от того, какой является ваша задача Z, вы можете подбирать методы решения.
2. Для задач в оптимизационной форме требования к точности можно ослабить, т.е. вместо оптимального искать приближенное решение.
3. Наложить на задачу дополнительные ограничения, тогда получается некоторый частный случай. Таким образом попытаться получить нетривиальный полиномиально разрешимый вариант исходной задачи из NPC.
4. Задачу всегда можно решить полным перебором. В некоторых случаях комбинаторная структура задачи позволяет разработать достаточно сложные и тонкие алгоритмы «направленного перебора», которые теоретически будут экспоненциальными, но на практике могут успешно использоваться.

Выше мы уже говорили о том, что оптимизационная форма NP-полной задачи, как правило, является NP-трудной задачей. Именно это мы будем иметь в виду всюду в данном разделе, когда будем говорить об оптимизационных задачах и NP-полноте.

11.1. NP-полнота в сильном смысле. Псевдополиномиальные алгоритмы

Пусть I – некоторая индивидуальная задача массовой задачи Z. Отметим, что задавая условие задачи (слово I в некотором алфавите) мы должны задать «комбинаторные» параметры задачи - число вершин, число ребер, число переменных, число дизъюнкций, ребра графа и т.п. – и «числовые»

параметры: веса ребер графа, вектора объемов и стоимостей в задаче о рюкзаке, веса камней в задаче о камнях и т.п. Обозначим через num(I) – максимальное число, которое используется при задании I.

Разумная кодировка предполагает, что для записи num(I) потребуется слово длины *log*(num(I)). Поэтому в P лежат задачи, для которых существуют алгоритмы решения с трудоемкостью, полиномиальной по |I| и *log*(num(I)).

Опр. Алгоритм решения задачи называется псевдополиномиальным, если он решает задачу за время, ограниченное полиномом от |I| и num(I).

Для большинства задач из NPC нет не только полиномиальных, но и псевдополиномиальных алгоритмов, поэтому наличие для задачи из NPC псевдополиномиального алгоритма, что она в каком-то смысле «проще», чем задача, для которой нет псевдополиномиального алгорима.

Примером задачи, имеющей псевдополиномиальный алгоритм является задача о рюкзаке. Есть ряд похожих на эту задачу – задачи рюкзачного типа, для которых тоже построены псевдополиномиальные алгоритмы. Поэтому на практике вам стоит проверить свою задачу на наличие псевдополиномиального алгоритма, тогда при малых значениях num(I) она будет сравнительно легко решаться.

Рассмотрим теперь задачи, для которых пока нет псевдополиномиального алгоритма. Для некоторых из них доказано, что они являются NP-полными даже тогда, когда num(I)≤p(|I|), где p(|I|)– полином от |I|. Такие задачи называются **NP –полными в сильном смысле** (сильно NP-полными).

Таким образом, класс NPC разбивается на три части: задачи с известными псевдополиномиальными алгоритмами; задачи, для которых не построены псевдополиномиальные алгоритмы, но и не доказано, что они сильно NP-полные; сильно NP-полные задачи. Если вам придется иметь дело с последними, то, по-видимому, ничего не остается как использовать либо эвристические алгоритмы, либо методы направленного перебора (см. ниже).

11.2. Приближенные алгоритмы

Понятие приближенный алгоритм мы будем использовать только для алгоритмов решения задач в оптимизационной форме.

Оглавление.

Опр. Пусть дана задача Z=(**F,c**) и – **c**(x*) значение функции стоимости **c** на оптимальном решении x*. Пусть дан некоторый алгоритм W, результатом работы которого будет выдача некоторого решения x', а $\mathbf{c}_W(x')$– значение функции стоимости на решении x'. Будем говорить, что алгоритм W является **ε-приближенным** (или алгоритмом с оценкой точности ε), если выполняется следующее соотношение:$|(\mathbf{c}_W(x')- \mathbf{c}(x^*))/ \mathbf{c}(x^*) | \leq \varepsilon$.

Опр. Пусть дана задача Z=(**F,c**) и – **c**(x*) значение функции стоимости **c** на оптимальном решении x*. Пусть дан некоторый алгоритм W, результатом работы которого будет выдача некоторого решения x', а $\mathbf{c}_W(x')$– значение функции стоимости на решении x'. Будем говорить, что алгоритм W является **эвристическим**, если про соотношение $\mathbf{c}_W(x')$ и **c**(x*) ничего не известно.

Часто эвристические алгоритмы называют приближенными без оценки точности. Примером эвристического алгоритма для задачи коммивояжера называется *жадный алгоритм* или алгоритм *иди в ближайшую*.

Пример. Дана задача коммивояжера на n городах с матрицей попарных расстояний. Будем строить решение x' следующим образом.

Из вершины 1 идем в вершину i_1 такую, что ребро $(1,i_1)$ имеет минимальный вес среди всех ребер вида (1,j), на втором шаге из вершины i_1 идем в вершину $i_2 \neq 1$ такую, что ребро (i_1,i_2) имеет минимальный вес среди всех ребер вида (i_1,j), $j \neq 1$; из вершины i_2 идем в вершину $i_3 \neq 1, i_2$ такую, что ребро (i_2,i_3) имеет минимальный вес среди всех ребер вида (i_2,j), $j \neq 1,i_2$. И так далее. На последнем шаге замыкаем цикл.

Легко привести пример того, что этот алгоритм может не находить оптимального решения. Трудоемкость алгоритма не превосходит $O(n^2)$, но мы ничего не можем сказать про соотношение $\mathbf{c}_W(x')$ и **c**(x*).

А как обстоят дела с приближенными алгоритмами для этой задачи?

Теорема. Если $P \neq NP$, то при любом $\varepsilon > 0$ не существует ε – приближенного алгоритма для задачи коммивояжера.

В некотором смысле ситуация с приближенными алгоритмами для задачи коммивояжера типична для NP-трудных задач: полиномиальные эвристические алгоритмы всегда есть, а полиномиальных приближенных (для общего случая) нет.

Оглавление.

Теорема. Если для задачи о клике существует ε – приближенный алгоритм для некоторого ε>0, то тгда для этой задачи можно будет построить ε – приближенный алгоритм для любого 1> ε>0.

Однако, в отличие от точных алгоритмов, полиномиальные приближенные можно построить для интересных частных случаев NP-трудных задач. Например, для задачи коммивояжера с неравенством треугольника (для любых трех вершин i,j,k длина ребра (i,j) не превосходит суммы длин ребер (i,k)и (k,j)) существуют приближенные алгоритмы. Одним из самых первых был 1/2-приближенный алгоритм Кристофидеса.

Мы видим, что с практической точки зрения, если вам нужно решать –трудную задачу, сложность точного и приближенного решения для общего случая одинакова, поэтому или вам надо искать в своей задаче какие-то особенности, которые сведут ее к частному случаю, либо пользоваться методами направленного перебора.

11.3. Полиномиально-разрешимые частные случаи NP-полных задач

Сразу же после появления теории NP-полноты большой интерес вызвал вопрос поиска полиномиально разрешимых частных случаев таких задач, но интерес к этому направлению достаточно быстро угас. Оказалось, что тривиальные случаи задач можно решать за полином, а , а большинство «сильных» упрощений, все равно, оставляют задачу в NPC.

Выше мы видели, что 3-КНФ – выполнимость еще лежит в NPC и только 2-КНФ – выполнимость лежит в P.

Для задачи коммивояжера связный граф с максимальной степенью вершин, равной двум, является просто гамильтоновым одним циклом. Задача тривиальна. Но уже для кубического графа получаем NP-трудную задачу. Если мы наложим ограничение в виде условия треугольника или потребуем, чтобы веса ребер принимали только одно из двух фиксированных значений, то снова получаем NP-трудную задачу.

Для задачи ЦЛП известен случай полиномиальной разрешимости – n=const. Но это снова очевидная ситуация.

Оглавление.

Задача о клике полиномиально разрешима для следующих графов: степень вершин ограничена константой; планарные графы, реберные графы.(Граф G является реберным графом, если существует граф G’ такой, что вершинам G соответствуют ребра G’ и в G существует ребро (x,y) тогда и только тогда. когда x и y имели в G’ общую концевую вершину.)

Задача гамильтонов цикл лежит в NPC даже если выполняются следующие условия: граф двудольный; максимальная степень вершины равна трем; граф планарен; в графе задан гамильтонов путь. Задача полиномиально разрешима для реберных графов и графов с максимальной степенью вершины, равной двум.

11.4. Методы направленного перебора

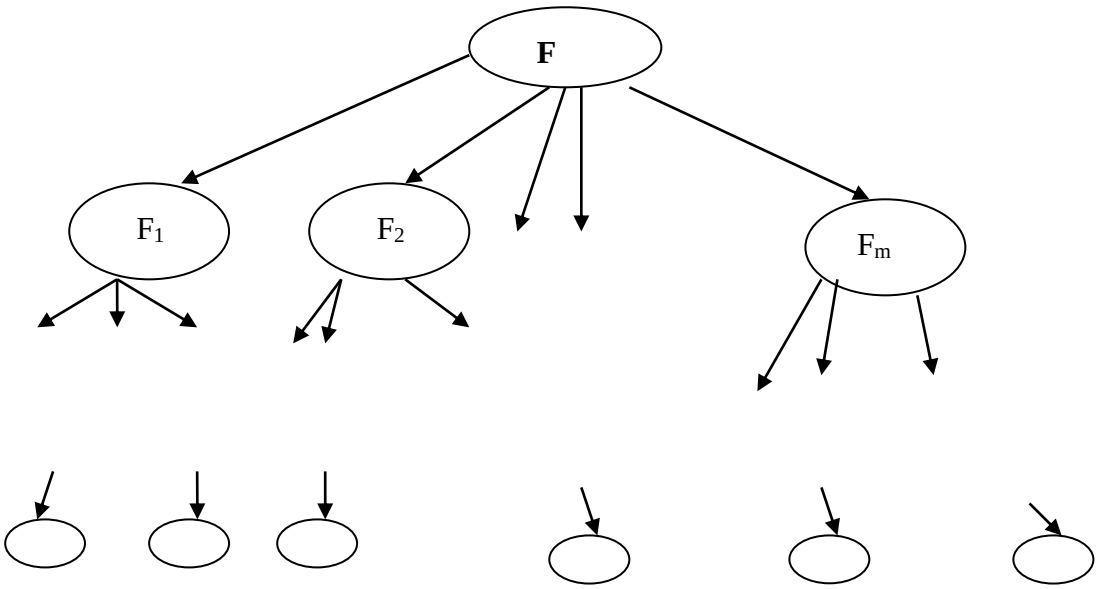
Пусть дана Z - задача оптимизации –пара **(F,c)** , где **F** –множество решений, а – **c** функция стоимости, отображающая элементы **F** на множество действительных чисел. Требуется найти такую x* точку из **F**, на которой значение функции **c**(x*) обладает определенным свойством, например, минимально, максимально и пр. Рассмотрим, например, задачу на минимум.

В задаче коммивояжера – это множество всех гамильтоновых циклов графа, в задаче ЦЛП – множество целочисленных точек многогранника и т.п.

Методы направленного перебора основаны на нескольких простых соображениях.

1. Экстремальное значение на множестве и на подмножестве связаны определенными неравенствами. В случае задачи на минимум $\mathbf{c}_G$ - минимум функции стоимости на подмножестве $G \subseteq F$ не превосходит **c**(x*) - минимума функции стоимости на всем множестве.
2. Пусть **l**(x*)≤ **c**(x*) ≤ **u**(x*), т.е. **l**(x*) и **u**(x*) - нижняя и верхняя оценка для **c**(x*). При этом значения **l**(x*) и **u**(x*) могут быть найдены полиномиальным алгоритмом.
3. Все множество F может быть иерархически разбито на непересекающиеся подмножества. Это разбиение представимо в виде *дерева ветвей*. Пример приведен на рисунке.

Оглавление.



Здесь каждая вершина – некоторое множество допустимых решений. Все множество решений, соответствующее каждому родительскому узлу разбивается на подмножества решений, соответствующих узлам-потомкам. Здесь все подмножества родительского узла не пересекаются, а их объединение в точности дает множество, соответствующее родительскому узлу. Листья дерева – одноэлементные множества.

4. Если для некоторого узла дерева $\mathbf{G} \subseteq \mathbf{F}$ выполняется соотношение

$$\mathbf{l}(x^*) < \mathbf{c}_G,$$

то это означает, что оптимальное решение x* не может находиться среди допустимых решений, соответствующих данному узлу, а все множество **G** можно исключить из дальнейшего рассмотрения. (Отрезать ветвь дерева.)

Отсюда следует другое название методов направленного перебора – методы ветвей и границ.

Оглавление.

12. Коммуникационная сложность

Этот подход к оценке сложности задач в некотором смысле связан с предыдущим и, так же как и параллельные вычисления, обусловлен, в первую очередь, развитием компьютерной техники.

В случае параллельных вычислений основная забота – эффективно использовать вычислительный ресурс. В модели параллельных вычислений процессоры обмениваются информацией. Время обмена информацией и стоимость такого обмена полагается пренебрежительно малой по сравнению со временем и стоимостью вычислений.

Если же поменять акценты на противоположные: время и стоимость вычислений пренебрежительно мала по сравнению временем обмена информацией и стоимостью такого обмена – получаем подход, называемый *коммуникационной сложностью*. Обсчет сложных физических экспериментов научными центрами, обмен информацией между гидрометцентрами в реальном времени для составления прогноза погоды и т.п. – это реальные современные задачи, которые встают в условиях распределенных хранилищ информации, критичности времени обмена и не всегда разумной стоимостью трафика как коммуникационного ресурса.

Так же, как и в случае параллельных вычислений, модель коммуникационного взаимодействия между хранилищами информации в процессе совместного решения конкретной индивидуальной задачи – это предмет постановки задачи. Такая модель обычно связывается с понятием *коммуникационного протокола*. Три основных группы параметров модели связаны со следующими ее составляющими.

1. **Участники**. Это процессоры (ФУ, серверы, ЛВС предприятий и т.п.), осуществляющие вычисления (обработку информации) на местах (в точках распределенной сети). Они могут быть равноправны и неравноправны, потребность в информации (обращении) каждого участника со стороны остальных может быть разной и т.д.
2. **Линии связи**. Стоимость обмена данными может зависеть от конкретной пары обменивающихся участников. Она может быть несимметричной. Пропускная способность канала связи может быть ограниченной и пр.

Оглавление.

3. **Ограничения на возможности (правила) обмена информацией**. Не все участники могут одновременно получать и отправлять информацию (кто-то может делать только одно из этих действий). Друг с другом можно связать только определенные пары участников. И т.д.

Все эти параметры в модели вычислений должны быть оговорены и тогда они станут частью ***протокола*** вычислений, который на их основе опишет правила работы получившейся распределенной вычислительной системы. Те, кто работал с СУБД, сразу смогут увидеть здесь аналогию этого протокола с протоколами обработки распределенной базы данных.

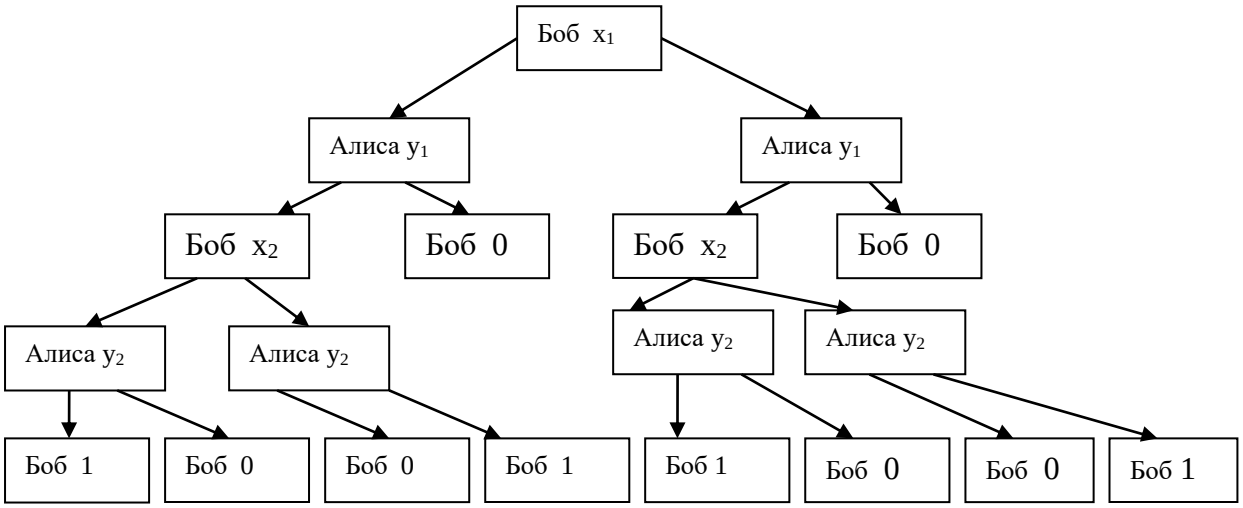
В теории сложности в качестве базового случая рассматривается простейший: два равноправных участника (Алиса и Боб) с симметрично распределенной между ними входной информацией совместно вычисляют булеву функцию $f(x_1,\ldots,x_n)$, обмениваясь в процессе вычисления сообщениями фиксированной длины (они называются *битами*). Процесс обмена представляется корневым ориентированным деревом (корень ассоциируется с инициатором обмена), каждая промежуточная вершина которого соответствует участнику посылающему информацию, висячая вершина – результату вычислений, а ребро значению бита (0 или 1). Такое дерево называется протоколом вычисления функции f, а длиной протокола называется максимум длины пути в этом дереве. Вообще говоря, даже в нашем простейшем случае можно построить разные протоколы (как можно построить разные машины Тьюринга для решения одной и той же задачи), поэтому формально определение коммуникационной сложности для рассматриваемой модели вычислений выглядит следующим образом.

Опр. Коммуникационная сложность вычисления функции – это минимум по длинам всех протоколов, вычисляющих эту функцию.

Получаем такой *min-max* –ый критерий. До сих пор мы имели дело с *max-min*-ми критериями: определение сложности «в худшем», функция Шеннона и пр.

Ниже приведен пример совместной проверки участниками равенства $x_1y_2=x_1y_2$.

Оглавление.



Очевидно, что в нашей простейшей модели коммуникационная сложность любой задачи не превосходит O(n) (здесь n, как обычно, обозначает длину входа задачи). Действительно, тривиальный протокол, состоящей в единичной посылке своих данных одним участником другому приводит к вычислению функции. (Еще раз подчеркнем, что в нашей модели трудоемкость самих вычислений равна нулю).

Приведем несколько примеров задач. Коммуникационная сложность которых может быть меньше O(n).

Пример. Задача «***четность***». Дано два булевых вектора: x (находится у Алисы) и y (находится у Боба). Определить четность длины вектора конкатенации xy.

Задача решается за O(1) путем посылки четности своей части одним участником другому.

Пример. Задача «***сумма бит***». Дано два булевых вектора одинаковой длины n: x (находится у Алисы) и y (находится у Боба). Определить, одинаково ли число единиц в этих векторах.

Задача решается за O(*log*n) очевидным образом путем посылки суммы единиц своей части одним участником другому.

Пример. Задача «***эквивалентность***». Дано два булевых вектора одинаковой длины n: x (находится у Алисы) и y (находится у Боба). Определить эквивалентны ли они.

Оглавление.

Задача решается за O(n) тривиальным алгоритмом путем посылки своего вектора одним участником другому. Оказывается, что ее нельзя решить быстрее. Это, в каком-то смысле тривиальное следствие теоремы Шеннона из теории информации. Вектора битовые. Каждый участник должен получить информацию о каждом бите, но нельзя *бит* закодировать информацией меньшей длины, чем один *бит*.

Но это тривиальные примеры. Учебник по коммуникационной сложности начинается, обычно, со следующего, менее очевидного примера.

Пример. Задача «***медиана***». Медианой упорядоченного числового множества из n элементов называется элемент этого множества, стоящая на]n/2[месте. Алиса и Боб имеют по одному подмножеству множества {1,2,…,n}. Требуется найти медиану объединения подмножеств Алисы и Боба.

Легко построить алгоритм решения со сложностью $O(\log^2 n)$. Для этого заметим, что медиана всего множества лежит между медианами частей Алисы и Боба. Вначале Боб посылает Алисе свою медиану, Алиса в ответ посылает ему медиану той части своего множества, которая лежит между ее медианой и медианой Боба и т.д. Получаем сходящуюся дихотомию, в которой не больше O(logn) шагов и на каждом шаге посылается O(logn) битов.

Однако можно построить и более экономный алгоритм сложности O(logn).

Рассматриваются и более общие (по сравнению с моделью Алиса-Боб) модели. Они, как правило, имеют практическое обоснование и учет их специфики приводит либо к тривиальным результатам, либо требует решения технически сложных задач на стыке комбинаторной математики и теории телекоммуникационных протоколов. Важно, что вы должны знать о существовании таких постановок и при столкновении с ними в своей будущей инженерной деятельности сможете, при желании, найти результаты тридцатилетних исследований в этой области. А, если эти результаты вам не помогут, то можете переквалифицироваться в математиков и попытаться внести свой вклад в эти исследования.

Оглавление.

13. Вопросы для самопроверки по курсу «Математическая логика и теория алгоритмов».

- Понятие «задача». Форма задачи. Индивидуальная и массовая задача. Привести примеры.
- Построить сведение задачи «Гамильтонов цикл» к задаче «Коммивояжер».
- Определение Машины Тьюринга (МТ). Различие детерминированной и недетерминированной МТ.
- Построить сведение задачи «КНФ-выполнимость» к задаче «Клика».
- Определение Оракульной Машины Тьюринга (ОМТ). Различие оракульной и недетерминированной МТ.
- Определение Недерминированной Машины Тьюринга (НМТ). Привести пример.
- Задание «входа» для индивидуальной задачи. Примеры кодировок графов. Понятие полиномиальной эквивалентности для различных кодировок объекта.
- Построить алгорифм Маркова сложения чисел.
- Классы P, NP, NPC. Соотношение между ними.
- Привести примеры *выполнимой* и *общезначимой* формулы в исчислении предикатов.
- Классы PSPASE, NPSPASE. Соотношение между ними.
- Построить СФЭ для функции (00110101).
- Классы PSPASE, EXPTIME. Соотношение между ними.
- Построить минимальную СФЭ для функции (01110100).
- Определение СФЭ. Пример СФЭ.

Оглавление.

- Построить МТ для обращения слов в заданном алфавите.
- Классы P/Poly и P. Соотношение между ними.
- Построить НАМ для обращения слов в заданном алфавите.
- Класс PSPASE и *игра двух лиц*. Соотношение между ними.
- По заданной формуле в исчислении предикатов построить формулу в *приведенной нормальной* форме.
- Отношения. Сводимость по Тьюрингу.
- Примеры полиномиально разрешимых частных случаев NP –полных задач.
- Предикат. Формула в исчислении предикатов. Выполнимые и общезначимые формулы. Приведенная нормальная форма.
- Теорема Кука. Схема доказательства.
- Понятие *алгоритма*. Эквивалентность алгоритмов. Тезис Черча.
- Алгоритм нахождения кратчайшего пути между вершинами графа.
- Примеры подходов к решению NP-полных задач.
- Определение *полиномиальной сводимости* и сводимости по Тьюрингу. Пример полиномиальной сводимости.
- Теорема о PSPACE-полной задаче.
- Классы NP и co-NP. Соотношение между ними. Теорема об NP–полной задаче и классе Co-NP.
- Приближенные алгоритмы с оценкой точности. Теорема о существовании такого алгоритма для «Задачи коммивояжера».
- Игра двух лиц. Определение. Игра двух лиц и классы NP и co-NP.
- Коммуникационная сложность. Коммуникационная сложность задачи о равенстве числа единиц в двух булевских векторах.

Оглавление.

- По заданной формуле в исчислении предикатов построить формулу в *приведенной нормальной* форме.

14. Рекомендованная литература

1. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979.
2. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.
3. Мендельсон Э. Введение в математическую логику. М.: Наука, 1971.
4. Пападимитриу Х., Стайглиц К. Комбинаторная оптимизация. Алгоритмы и сложность. М.: Мир, 1984.
5. Гордеев Э.Н. Задачи выбора и их решение. – Сб. «Компьютер и задачи выбора. М.: Наука, 1989.
6. Кузюрин Н.Н., Фомин С.А. Эффективные алгоритмы и сложность вычислений.М: МФТИ, 2007.
7. Лупанов О.Б. Курс лекций по дискретной математике. - Конспект лекций. МГУ.

Оглавление.