Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Магнитогорский государственный технический университет им. Г.И. Носова»

А.Ю. Миков
Н.С. Сибилева

АЛГОРИТМЫ И ТЕОРИЯ СЛОЖНОСТИ

*Утверждено Редакционно-издательским советом университета
в качестве практикума*

Магнитогорск
2017

УДК 004.021

Рецензенты:

кандидат педагогических наук,
заместитель директора,
Филиал ОАНО ВО «Московский психолого-социальный университет»
в г. Магнитогорске
А.Ю. Воробьева

кандидат технических наук,
доцент кафедры бизнес-информатики
и информационных технологий,
ФГБОУ ВО «Магнитогорский государственный
технический университет им. Г.И. Носова»
К.А. Рубан

Миков А.Ю., Сибилева Н.С.
Алгоритмы и теория сложности [Электронный ресурс] : практикум / Анатолий Юрьевич Миков, Наталья Сергеевна Сибилева ; ФГБОУ ВО «Магнитогорский государственный технический университет им. Г.И. Носова». – Электрон. текстовые дан. (0,87 Мб). – Магнитогорск : ФГБОУ ВО «МГТУ им. Г.И. Носова», 2017. – 1 электрон. опт. диск (CD-R). – Систем. требования : IBM PC, любой, более l GHz ; 512 Мб RAM ; 10 Мб HDD ; MS Windows XP и выше ; Adobe Reader 8.0 и выше ; CD/DVD-ROM дисковод ; мышь. – Загл. с титул. экрана.

В практикуме рассмотрена машина Тьюринга, представлена классификация задач по степени сложности. Рассмотрены некоторые задачи из класса NP и алгоритмы их решения. Приведены контрольные вопросы и задания для закрепления материала.

Издание предназначено для студентов, обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника» при изучении дисциплин «Алгоритмы и теория сложности», «Алгоритмы на сетях и графах», «Структуры и модели данных».

УДК 004.021

Содержание

ВВЕДЕНИЕ

При решении задач производительность алгоритма всегда имеет большое значение. Он не будет достаточно эффективным, если работает очень долго или требует слишком много памяти либо других ресурсов компьютера.

Теория вычислительной сложности – тесно связанный с этим раздел информатики, изучающий сложность вычислительных задач.

Теория вычислительной сложности – обширная и сложная тема, на полное раскрытие которой данный практикум не претендует. Тем не менее любой программист, изучающий алгоритмы, должен иметь о ней хотя бы общее представление, что в первую очередь касается классов P и NP. В данном пособии рассмотрены некоторые примеры задач из класса NP и приводятся алгоритмы поиска точно решения для них.

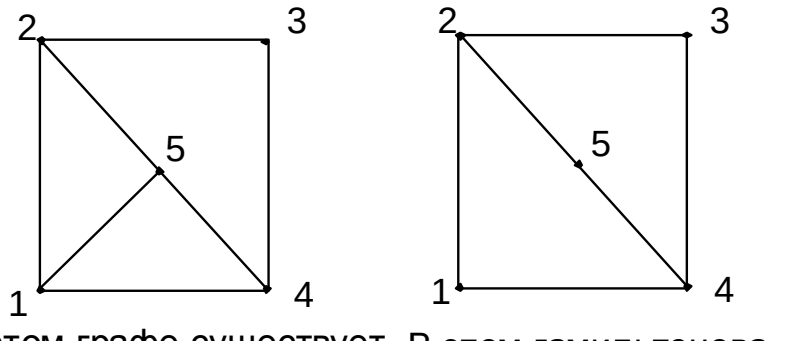
В данном практикуме кратко приводятся теоретические сведения и на конкретных задачах показывается их практическое приложение.

В практикуме акцент сделан на алгоритм с возвратом и его модификации для поиска точного решения некоторых задач класса NP. Алгоритм с возвратом – рекурсивный алгоритм, осуществляющий рекурсивный вызов всегда, когда существуют разные варианты решения задачи, т.е. происходит ветвление. Рекурсивный процесс запоминает текущее состояние алгоритма, поэтому если данный вариант решения оказался неудачным, то процесс возвращает решение задачи в то положение, в котором произошло ветвление (точка ветвления) и пытается достичь успеха на другом пути. Из модификаций данного алгоритма рассматриваются: метод ветвей и границ, принцип включения-невключения.

Для закрепления рассматриваемых в данном практикуме алгоритмов необходимо ответить на контрольные вопросы и реализовать алгоритм на каком-либо языке программирования.

1. КЛАССИФИКАЦИЯ ЗАДАЧ ПО СТЕПЕНИ СЛОЖНОСТИ

Для решения задачи об эйлеровом цикле имеется прекрасный линейный алгоритм – его сложность O(m), т.е. он линейно зависит от числа ребер. Существует похожая задача. Она состоит в поиске цикла, проходящего через каждую вершину графа только один раз. Эту задачу изучал Гамильтон, и в честь него она получила название задачи Гамильтона (рис. 1.1).



В этом графе существует гамильтонов цикл

В этом графе гамильтонова цикла нет

Рис. 1.1. Задача Гамильтона

Многие математики в течение нескольких веков занимались задачей Гамильтона. До сих пор неизвестно ни одного простого необходимого и достаточного условия существования гамильтонова цикла, и до сих пор не построен алгоритм, который проверял бы существование гамильтонова цикла в произвольном графе за **полиномиальное** число шагов (число, ограниченное многочленом фиксированной степени от числа вершин графа).

Задача об эйлерове цикле и задача о гамильтонове цикле являются представителями задач различных степеней сложности.

Основное различие между **математикой** и **информатикой** заключается в том, что в информатике недостаточно иметь доказательство существования объекта, и даже недостаточно найти конструктивное доказательство этого факта, т.е. алгоритм. Мы должны учитывать противоречия пространства и времени, которые навязывает нам мир: необходимо, чтобы решение можно было вычислить, используя объем памяти и время, приемлемое для человека или машины.

Идеальный случай, когда для решения задачи известна явная математическая формула. Тогда сложность задачи не зависит от входных данных и является постоянной.

Если же мы не располагаем ни явной формулой, ни рекурсивным выражением приемлемой сложности, нам остается только два способа:

– построение эффективного алгоритма;
– перебор всех или почти всех вариантов.

Сложность задачи – сложность наилучшего алгоритма, известного для ее решения.

Сложность алгоритма – число шагов, необходимых для решения наихудшего из всех возможных случаев, допускающих применение алгоритма. Это число выражается как функция от размерности входных данных задачи – например, числа вершин и рёбер графа.

Сразу возникает вопрос – если сложность задачи зависит от нашего знания «хороших» алгоритмов, до какого предела можно улучшать данный алгоритм? Существуют ли методы перевода задачи из класса «сложных» в класс «простых»?

Три класса задач

Вычислительная сложность алгоритма равна числу элементарных шагов, выполняемых алгоритмом в самом плохом случае, и зависит от размерности входных данных.

Объясним более точно, что подразумевается под «элементарным шагом».

Допустим, что транслятор типичной ЭВМ переводит программу в машинный язык, имеющий арифметические операции, условные переходы, операции ввода-вывода, команды переноса из памяти в буфер и т.д. Выполнение любой такой операции будем считать элементарным шагом. Очевидно, что сложность алгоритма будет зависеть от конкретного транслятора. Однако нас будет интересовать не абсолютная сложность, а сложность с точностью до умножения на произвольную постоянную. Такая сложность показывает, как быстро растет число шагов при неограниченном увеличении размерности входных данных. Такая **сложность** называется **асимптотической** и не зависит от способа трансляции. Асимптотическая сложность (производительность) определяется функцией, которая указывает, насколько ухудшается работа алгоритма с усложнением поставленной задачи. Такую функцию записывают в круглых скобках, предваряя прописной буквой O.

Самыми лучшими являются линейные алгоритмы порядка O(n), где n – размерность входных данных, например алгоритм поиска эйлеровых циклов в графе. Другие «хорошие» алгоритмы имеют сложность, представляющую собой многочлен заданной степени, не зависящей от входных данных: O(nm), $O(n^2)$, O(n^3) и т.д. Задачи, решаемые алгоритмами полиномиальной вычислительной сложности, относятся к **классу *P* полиномиальных алгоритмов**.

Есть задачи, которые по своей природе имеют экспоненциальную сложность порядка a^n, где a – константа или полином от n. Это задачи, в которых требуется построить все подмножества данного множества, все клики (полные подграфы) данного графа и т.д. Число ответов в этих задачах уже само по себе экспоненциально, поэтому только их перечисление потребует экспоненциальное число шагов. Это **класс *E* экспоненциальных алгоритмов**.

Остаются задачи, не попавшие ни в класс P, ни в класс E. В их условиях не содержатся экспоненциальные множества, для них не разработан полиномиальный алгоритм, но и не доказано, что такого алгоритма не существует.

Вот небольшой список таких задач (в приложении содержатся точные математические постановки 32 различных задач):

- ✓ Решение систем уравнений в целых числах.
- ✓ Определение гамильтонова цикла.
- ✓ Составление расписаний и раскрасок.
- ✓ Существование множества значений логических переменных, которые позволяют сделать значение произвольного заданного логического выражения истинным.
- ✓ Оптимизация пути коммивояжера через сеть городов.
- ✓ Отбор файлов при запросе в информационный банк данных для получения информации с наименьшей стоимостью.
- ✓ Размещение обслуживающих центров для максимального числа клиентов при минимальном числе центров.
- ✓ Оптимальная загрузка емкости, оптимальный раскрой.
- ✓ Диагностика (болезни, поломки).

Этот список далеко не полон, так как большая часть современных задач относится к этому классу. Кроме того, названные задачи являются модельными. Каждой из них соответствует несколько реальных формулировок:

- ✓ упорядочение операций;
- ✓ размещение персонала;
- ✓ оптимизация перевозок;
- ✓ управление производством;
- ✓ проектирование в области электроники.

Для большинства этих задач (так называемых **NP–полных задач**) была доказана эквивалентность – если для одной из них удастся найти полиномиальный алгоритм, автоматически будут решены все остальные.

Назовем его классом ***NP*** (**недетерминированных полиномиальных алгоритмов)**, класс *P* входит в этот класс. За вычетом класса *P* остальные задачи этого класса являются **труднорешаемыми** – точные алгоритмы, известные для их решения, имеют экспоненциальную вычислительную сложность. Задачи из класса NP, решаемые за полиномиальное время, называются **легкорешаемыми**. Задачи из класса NP, для решения которых неизвестны полиномиальные алгоритмы, называются **труднорешаемыми**..

При практическом изучении подобных задач нужно помнить следующее:

Для небольших *n* экспоненциальный алгоритм часто более эффективен, чем полиномиальный.

Для реальных задач большой размерности следует разработать приближенный или эвристический полиномиальный алгоритм.

Взаимоотношения между классами P и NP

Верно ли, что P = NP? Данный вопрос представляет собой важнейшую задачу в теории вычислительных систем, и каждый алгорист должен понимать, о чем речь. Следует отметить, что все результаты современной теории алгоритмов получены в предположении, что ***P≠NP.***

В сравнении классов сложности P и NP главным вопросом является, действительно ли *верификация* решения представляет более легкую задачу, чем первоначальный *поиск* решения. Допустим, что при сдаче экзамена вы «случайно» заметили ответ у студента, сидящего рядом с вами. Принесет ли это вам какую-либо пользу? Вы бы не рискнули сдать этот ответ, не проверив его, т.к. вы способный студент и могли бы решить данную задачу самостоятельно, если бы уделили ей достаточно времени. Но при этом важно, можете ли вы проверить правильность подсмотренного ответа на задачу быстрее, чем решить саму задачу самостоятельно.

Рассмотрим примеры.

✓ Можно ли проверить, что граф содержит маршрут коммивояжера с наибольшим весом *k*, если дан порядок вершин маршрута? Да. Просто сложим вместе веса ребер маршрута и покажем, что общий вес равен, самое большее, *k*. Это ведь легче, чем найти сам путь, *не так ли?*

✓ Можно ли проверить, что данный набор значений истинности представляет решение для данной задачи выполнимости? Да. Просто проверим каждую дизъюнкцию и убедимся, что она содержит, по крайней мере, один истинный литерал из данного набора значений истинности. Это ведь легче, чем найти выполняющий набор значений истинности, *не так ли*?

✓ Можно ли проверить, что граф *G* содержит вершинное покрытие, состоящее из, самое большее, *k* вершин, если известно подмножество *S*, состоящее из, самое большее, *k* вершин, составляющих данное покрытие? Да. Просто обходим каждое ребро *(u, v)* графа *G* и проверяем, что или *u* или *v* является элементом *S*. Это ведь легче, чем найти само вершинное покрытие, *не так ли*?

На первый взгляд, все просто. Данные решения можно проверить за линейное время для всех трех задач, однако для решения любой из них неизвестно никакого алгоритма, кроме полного перебора. Но проблема в том, что у нас нет строгого *доказательства* нижней границы, препятствующей существованию эффективного алгоритма для решения этих задач. Возможно, в действительности существуют полиномиальные алгоритмы (скажем, с временем исполнения $O(n^7)$), но мы просто недостаточно хорошо их искали.

Важнейший вопрос состоит в том, действительно ли класс NP содержит задачи, которые не могут быть членами класса P. Если такой задачи не существует, то классы должны быть одинаковыми и P = NP. Но если существует хотя бы одна такая задача, то классы раз-

ные и P ≠ NP. Большинство алгористов и теоретиков сложности вычислений придерживаются мнения, что классы разные, т.е. что P ≠ NP, но для него требуется гораздо более строгое доказательство, чем заявление: «Я не могу найти достаточно быстрый алгоритм».

Существует громадное дерево сведений задач NP-полноты, которое всецело основано на сложности задачи выполнимости. Часть задач из этого дерева показана на рис. 1.2.

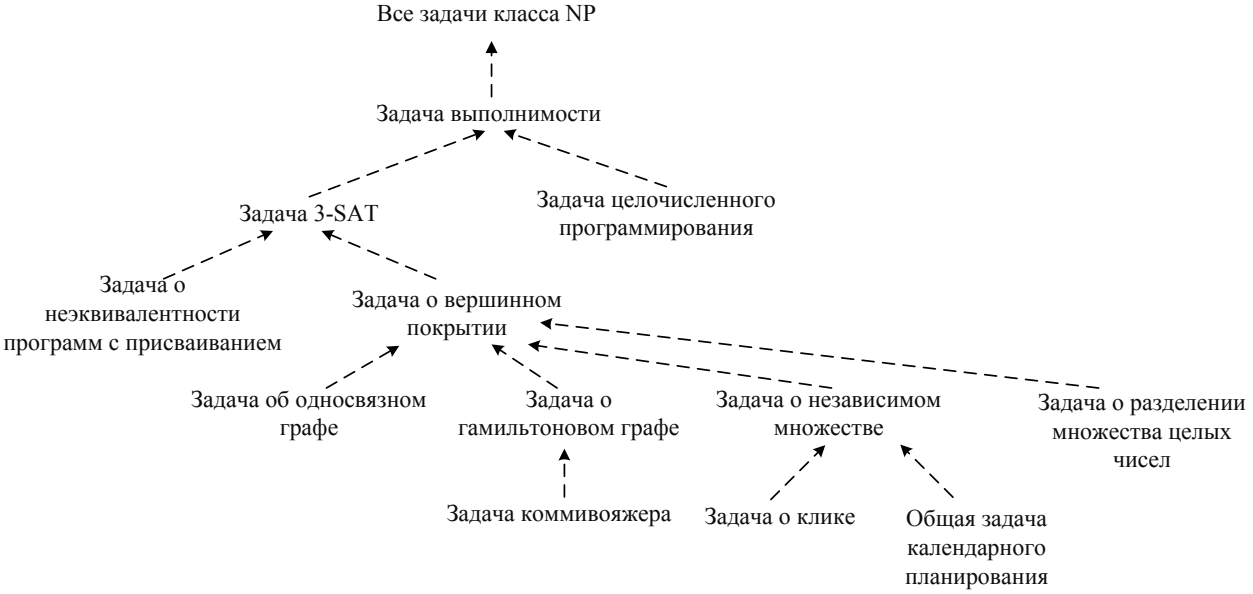


Рис. 1.2. Дерево сведений задач NP-полноты

Существует замечательное сведение, называемое теоремой Кука, которое сводит все задачи класса NP к задаче выполнимости. Таким образом, если доказать, что задача выполнимости, или любая NP-полная задача, является членом класса P, то за ней последуют все другие задачи в классе NP, из чего будет следовать, что P = NP. Так как, в сущности, каждая упомянутая в данном практикуме задача является членом класса NP, то результаты такого развития событий были бы весьма значительными и удивительными.

Теорема Кука доказывает, что задача выполнимости является такой же сложной, как и любая другая задача из класса NP. Кроме этого, она также доказывает, что каждая NP-полная задача такая же сложная, как и любая другая. Здесь уместно вспомнить про эффект домино. Но то обстоятельство, что мы не можем найти эффективного алгоритма ни для одной из этих задач, дает веское основание полагать, что все они действительно сложные и что, вероятно, P ≠ NP.

Говорят, что задача является NP-*сложной*, если, подобно задаче выполнимости, она, по крайней мере, такая же сложная, как любая другая задача класса NP. Говорят, что задача является NP-*полной*, если она NP-*сложная* и также является членом класса NP. Так как класс NP является таким большим классом задач, большинство NP-сложных задач, с которыми вам придется столкнуться, в действительности будут NP-полными, и вы всегда можете предоставить стратегию проверки решения задачи (обычно достаточно простую). Все рассмотренные в этом практикуме NP-сложные задачи также являются NP-полными.

Тем не менее, существуют задачи, которые кажутся NP-сложными, но при этом не являются членами класса NP. Такие задачи могут быть даже более сложными, чем NP-полные задачи! В качестве примера сложной задачи, не являющейся членом класса NP, можно привести игру для двух участников, такую, как шахматы. Представьте себе, что вы сели играть в шахматы с самоуверенным игроком, который играет белыми. Он начинает игру ходом центральной пешки на два поля и объявляет вам мат. Единственным способом доказать его правоту будет создание полного дерева всех ваших возможных ходов и его лучших ответных ходов и демонстрация невозможности вашего выигрыша в текущей позиции. Количество узлов этого полного дерева будет экспоненциально зависеть от его высоты, равной количеству

ходов, которые вы сможете сделать, перед тем как проиграть, применяя максимально эффективную защиту.

Ясно, что это дерево нельзя создать и проанализировать за полиномиальное время, поэтому задача не является членом класса NP.

Контрольные вопросы.

1. Как соотносятся между собой классы *P* и *NP*?
2. Какие задачи являются труднорешаемыми?
3. В чем отличие между сложностью задачи и сложностью алгоритма?
4. У вас есть два алгоритма. Один состоит из $N^3/75 - N^2/4 + N + 10$ шагов, другой – из $N/2 + 8$ шагов. Каково должно быть значение N, чтобы вы выбрали первый/второй алгоритм?

2. ДЕТЕРМИНИРОВАННАЯ МАШИНА ТЬЮРИНГА (ДМТ)

Структура машины Тьюринга

Машина Тьюринга имеет бесконечную ленту, управляющее устройство и считывающую головку. Лента разделена на ячейки, в каждой ячейке записан или пустой символ или символ некоторого конечного алфавита. В каждый момент на ленте может быть только конечное число пустых символов. Управляющее устройство находится в одном из конечных множеств состояний, среди них присваивается начальное q_1 и заключающий q_2. В начале ДМТ находится в начальном положении, попав в конечное положение, останавливается.

Определение. Машиной Тьюринга называется совокупность пяти объектов <A, λ, Q, q_1, I>:

1. Алфавит A = {a_1, …, a_n} – множество символов данной машины Тьюринга; |A| ≥ 1. Иногда алфавит называют множеством внешних состояний машины Тьюринга.

2. Пустой символ $\lambda \notin A$.

3. Множество состояний $Q = \{q_1, \ldots, q_k\}$, |Q| ≥ 1. В некоторых случаях это множество также называют множеством внутренних состояний машины Тьюринга.

4. Начальное состояние $q_1 \in Q$.

5. Программа I – множество команд вида $q_i a \rightarrow q_j b$, где $q_i, q_j \in Q$.

Один такт работы ДМТ можно описать следующим образом:

1. Считать символ из текущей ячейки.
2. Записать вместо него новый.
3. Остаться на месте или сдвинутся в соседнюю ячейку.
4. В зависимости от текущего состояния и прочитанного символа перейти в новое состояние.

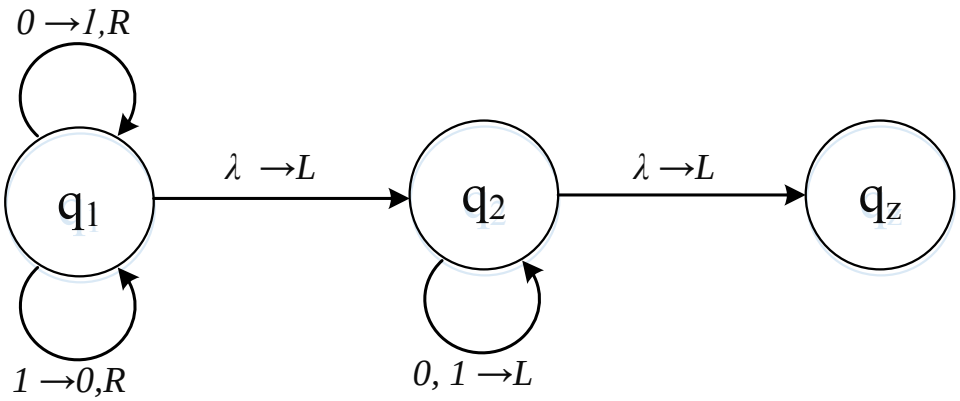
$A = \{a_1; \ldots; a_n\}$ - множество символов

$Q = \{q_1; \ldots; q_n\}$ - множество состояний

$q_i a_j \rightarrow q_i' a_j' dk$, где dk – команда сдвига

Пример 2.1: На ленте записано слово из 0 и 1. $A = \{0,1,\lambda\}$. Составить систему команд для машины Тьюринга, которые заменят 0 на 1, а 1 на 0.

Диаграмма перехода:



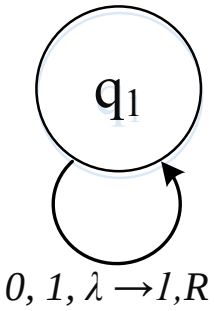
Набор команд:

$$q_1 1 \to q_1 0R$$
$$q_1 0 \to q_1 1R$$
$$q_1 \lambda \to q_2 \lambda L$$
$$q_2 0 \to q_2 0L$$
$$q_2 1 \to q_2 1L$$
$$q_2 \lambda \to q_z R$$

Пример 2.2: Бесконечно двигаясь вправо заполнить всю ленту 1.

$A = \{0,1,\lambda\}$

$Q = \{q_1, q_2\}$



Задание 2.1 (пороговый уровень)

На ленте записано слово. A = {0, 1}. Построить диаграмму переходов для машины Тьюринга согласно вариантам, предложенным в таблице 2.1. Определить множество символов задачи.

Таблица 2.1

Условия для решения задач

Вариант	*Условие задачи*
1	Каждый второй ноль заменить на единицу и вернуться в начало
2	Инвертировать четные единицы на нули, вернуться в начало
3	Заменить каждый третий нуль на единицу, вернуться в начало
4	Инвертировать нечетные нули на единицы и вернуться в начало
5	Каждую вторую единицу заменить на нуль и вернуться в начало
6	Заменить вторую единицу на нуль и вернуться в начало
7	Заменить четные нули на единицы, вернуться в начало
8	Заменить третий нуль на единицу, вернуться в начало
9	Инвертировать нечетные единицы на нули и вернуться в начало
10	Каждую четвертую единицу заменить на нуль и вернуться в начало

Задание 2.1 (средний уровень)

На ленте записано слово. A={0, 1}. Построить диаграмму переходов для машины Тьюринга согласно вариантам, предложенным в таблице 1.2. Определить множество символов задачи.

Таблица 2.2

Условия для решения задач

Вариант	*Условие задачи*
1	Заменить все нули на единицы и две последних единицы на нули, вернуться в начало
2	Все единицы заменить на нули, два последних нуля заменить на единицы, вернуться в начало
3	Все нули заменить на единицы, последнюю единицу на нуль, вернуться в начало
4	Все единицы заменить на нули, последний нуль заменить на единицу, вернуться в начало
5	Все нули заменить на единицы, две последних единицы заменить на нули, вернуться в начало
6	Заменить все единицы на нули, третий нуль заменить на единицу, вернуться в начало
7	Заменить все нули на единицы, вторую единицу с конца заменить на нуль, вернуться в начало
8	Заменить все единицы на нули, второй нуль заменить на единицу, вернуться в начало
9	Заменить все нули на единицы, третью единицу заменить на нуль, вернуться в начало
10	Заменить все единицы на нули, второй нуль с конца заменить на единицу, вернуться в начало

Задание 2.1 (высокий уровень)

Осуществить построение диаграммы переходов для машины Тьюринга согласно вариантам, представленным в таблице 2.3. Определить множество символов задачи.

Таблица 2.3

Условия для решения задач

Вариант	*Условие задачи*
1	На ленте записано число в унарной системе счисления. Разработайте машину Тьюринга для следующей функции: f(x)=2x
2	На ленте записано число в унарной системе исчисления. Разработайте машину Тьюринга для следующей функции: f(x)=x mod 3 +1
3	На ленте дана последовательность из символов «a» и «b». Удалить из последовательности второй символ, если такой есть.
4	На ленте дана последовательность из символов «a», «b», «c». Если первый и последний символы (непустой) последовательности одинаковы, тогда ее менять не нужно, а иначе заменить ее на пустое слово.
5	На ленте дана последовательность из символов «a», «b», «c». Вставить в исходную последовательность символ «b» за первым вхождением символа «c», если такое вхождение есть.
6	На ленте записано число в унарной системе счисления. Разработайте машину Тьюринга для следующей функции: f(x)=x mod 3 +2
7	На ленте дана последовательность из символов «a», «b», «c». Перенести последний символ последовательности в его начало
8	На ленте написаны два числа в двоичной системе счисления, разделенные символом «+». Необходимо вычислить их сумму.
9	На ленте представлена последовательность из десятичных чисел. Требуется получить на ленте запись числа, которое на 1 больше исходной последовательности.
10	На ленте записано число в унарной системе счисления. Разработайте машину Тьюринга для следующей функции: f(x,y) = x + (y mod 3)

Контрольные вопросы:

1. Из каких компонентов состоит машина Тьюринга?
2. Дайте определение универсальной машине Тьюринга.
3. Какими способами можно представить машину Тьюринга?
4. Как много можно спроектировать машин Тьюринга?
5. Сформулируйте тезис Тьюринга?
6. Какие функции называются вычислимыми?
7. Существуют ли невычислимые функции? Почему?
8. В чем заключается проблема остановки?

3. АЛГОРИТМ С ВОЗВРАТОМ

Для задачи существования гамильтонова пути – пути, проходящего через каждую вершину графа только один раз, не построен полиномиальный алгоритм. Попробуем произвести полный перебор всех возможных путей. *N* вершин графа можно расположить в *n*! различных цепочек. Чтобы проверить для каждой цепочки, реализована ли она как гамильтонов путь на графе, необходимо еще *n* шагов, итого сложность полного перебора *n*·*n*! шагов.

Такая величина растет гораздо быстрее экспоненты, поэтому полный перебор является неприменимым методом. Однако число шагов в алгоритмах переборного типа можно значительно уменьшить. Но для начала необходимо рассмотреть общую схему алгоритма с возвратом. В алгоритме, основанном на возвращениях, при каждом выборе запоминается текущее состояние решения задачи. Если выбор не ведет к решению, алгоритм восстанавливает сохраненное состояние и осуществляет следующий выбор.

Общая схема алгоритма с возвратом

Пусть решение, которое мы ищем, имеет вид последовательности <X(1), …, X(*n*)>.

Будем строить его, начиная с пустой последовательности длины 0.

Пусть на каком-то этапе уже построено частичное (неполное) решение длины *i*: <X(1), …, X(*i*)>.

Попытаемся продолжить это решение еще на один шаг. Для этого нужно отыскать допустимое X(*i*+1). X(*i*+1) считается допустимым, если:

– или <X(1), …, X(*i*+1)> уже является решением;
– или относительно <X(1), …, X(*i*+1)> нельзя сразу сказать, что его невозможно расширить до полного решения.

Если допустимое X(*i*+1) существует, то пытаемся продолжить (или вызываем рекурсию) для частичного решения <X(1), …, X(*i*+1)>.

Если допустимого X(*i*+1) не существует, то делаем возврат: возвращаемся на шаг назад к частичному решению <X(1), …, X(*i*-1)> и для него отыскиваем другое *X*'(*i*), не совпадающее с предыдущим X(*i*).

Более точно, пусть для каждого $i > 0$ существует множество A(*i*), из которого будут выбираться X(*i*) – претенденты на заполнение *i*–й позиции частичного решения.

Очевидно, множества A(*i*) должны содержать все X(*i*), занимающие *i*-ю позицию любого решения. Кроме того, A(*i*) всегда будут содержать какие-то лишние элементы, не содержащие в *i*–й координате ни одного решения.

Теперь общая схема алгоритма с возвратом имеет следующий вид:

```
1  begin
2    k:= 1;
3    while k>0 do
4      if существует еще новый y ∈ A(k), являющийся допустимым then
5        begin
6          X[k]:= y; {y использован}
7          if <X[1],…,X[k]> является решением then write (<X[1],…,X[k]>);
8          k:= k+1
9        end
10     else {возврат на предыдущее частичное решение,
                все элементы A(k) становятся неиспользованными}
11       k:= k-1
12 end.
```

Если все решения имеют длину, меньшую n + 1, и все множества A(k) состоят из конечного числа элементов, то этот алгоритм находит **все решения**.

Этот алгоритм можно очень просто записать с помощью рекурсии.

```
{рекурсивный вариант алгоритма с возвратом}
1 procedure BC(k);
 {генерируем все решения, являющиеся продолжением
  частичного решения <X[1],…,X[k-1]>, массив X–глобальный}
2 begin
3   for y ∈ A(k) do
4     if y допустим then
5       begin
6         X[k]:= y;
7         if <X[1],…,X[k]> – решение then
8           write(<X[1],…,X[k]>);
9         BC(k+1);  {генерируем все решения, являющиеся продолжением
                      частичного решения <X[1],…,X[k-1],y>}
10        y неиспользован;  {возврат на <X[1],…,X[k-1]>}
11      end {цикл 3 выберет для продолжения
             <X[1],…,X[k-1]> следующий y'≠y}
12 end;
```

***Вопрос* 3.1.** Каким образом частичное решение <X(1), …, X(k)> будет продолжено внутри BC(k+1)?

Сгенерировать все решения можно вызовом BC(1) из главной программы. Описание алгоритма возврата мы начали с несколько более сложного нерекурсивного варианта, так как в рекурсивном варианте «возврат» является частью механизма рекурсии.

Задача Гамильтона

Применим общую схему алгоритма с возвратом для генерации всех гамильтоновых циклов в графе G=<V,E>. Каждый такой цикл – последовательность различных вершин графа <X(1), …, X(n+1)> и только X(1) = X(n+1) = k, где k – произвольная фиксированная вершина; соседние вершины X(i) и X(i+1) соединены ребром. A(i) = V (множество всех вершин) для любого $i \le$ n+1.

Условие допустимости вершины y для частичного решения <X[1], …, X[i-1]>:

– $y \in$ ЗАПИСЬ[X[i-1]] (вершина y соединена ребром с X[i-1]);
– y новая, то есть не принадлежит <X[1], …, X[i-1]>.

Нахождение всех гамильтоновых циклов в неориентированном графе

Данные: Граф G=<V,E>, представленный списками ЗАПИСЬ[v], $v \in V$.

Результаты: Печать всех гамильтоновых циклов графа G.

Глобальные переменные: X, DOP.

```
1  procedure GAMI(i);
 {генерация всех гамильтовых циклов, являющихся
       продолжением частичного решения <X[1],…,X[i-1]>}
2  begin
3     for y Є ЗАПИСЬ[X[i-1]] do
```

```
4       if (i=n+1) AND (y=k) then
5         write(X[1],…,X[n],k)
6       else if DOP[y] then
7         begin
8           X[i]:= y; DOP[y]:= false;
9           GAMI(i+1);
10           DOP[y]:= true;
             {все решения, продолжающие <X[1],…,X[i-1],y> уже
              сгенерированы, выбираем другое y',
              а y становится неиспользованным}
10      end
11  end;{GAMI}


1 begin {основная программа}
2   for v Є V do DOP[v]:= true;
3   X[1]:= k ; {k – произвольная вершина}
4   DOP[k]:= false;
5   GAMI(2);
6 end.
```

Протрассируем алгоритм для модельного графа и построим его дерево поиска (рис. 3.1):

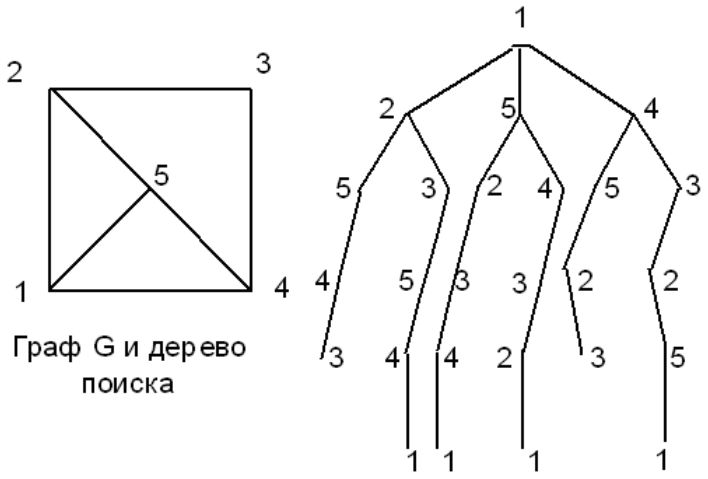


Рис. 3.1. Трассировка алгоритма

Вычислительная сложность этого алгоритма экспоненциальна и для случая, когда ищется ровно одно решение (если задача не имеет решения, то алгоритм перебирает все варианты).

***Вопрос* 3.2.** Докажите, представив соответствующий «плохой» граф, что число шагов в алгоритме нахождения всех гамильтоновых циклов растет экспоненциально с ростом n.

***Ответ* 3.1**. Во всех направлениях, до всех тупиков и полных решений.

***Ответ* 3.2**. Граф типа «погремушка» (см. рис. 3.2). Такой граф не содержит ни одного гамильтонова цикла, в то же время все вершины, кроме V_0, соединены и представляют собой

клику, т.е. все вершины соединены попарно. Всего различных путей будет $\frac{(n-1)!}{2}$, а для проверки каждого из них потребуется n шагов, итого O(n!), что растет быстрее экспоненты.

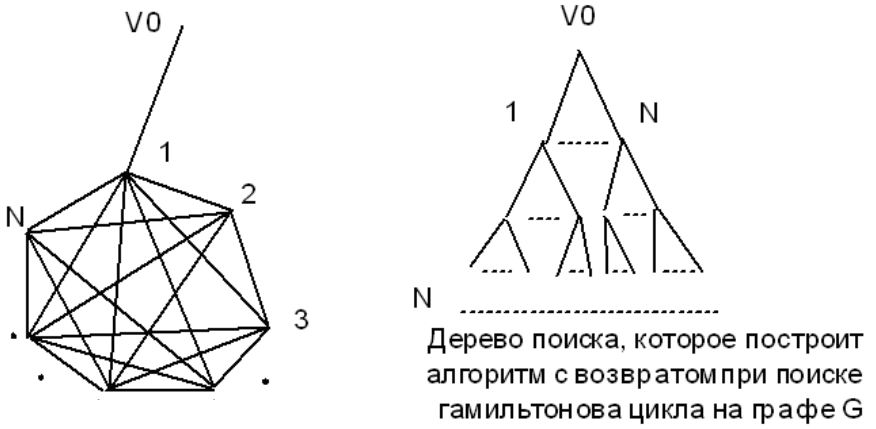


Рис. 3.2. Граф «погремушка»

Задача о весах

Условие задачи: имеется разновес из гирь, причем могут в разновесе может быть несколько гирь с одинаковым весом.

Задание: набрать заданный вес всеми возможными способами с точностью до перестановки гирь.

Варианты наборов исходных данных представлены в таблице 3.1.

Таблица 3.1

Варианты наборов исходных данных

№ варианта	Исходный набор предметов	Вес, который необходимо набрать
1	1 кг, 1 кг, 2 кг, 2 кг, 2 кг, 4 кг, 5 кг, 7 кг	9
2	1 кг, 1 кг, 1 кг, 3 кг, 3 кг, 5 кг, 5 кг, 6 кг, 9 кг, 10 кг	12
3	1 кг, 1 кг, 1 кг, 1 кг, 2 кг, 2 кг, 3 кг, 3 кг, 3 кг, 5 кг, 7 кг, 10 кг	13
4	1 кг, 1 кг, 2 кг, 4 кг, 5 кг, 6 кг, 7 кг, 8 кг	8
5	1 кг, 1 кг, 1 кг, 3 кг, 4 кг, 7 кг, 8 кг, 9 кг	9
6	1 кг, 1 кг, 1 кг, 1 кг, 2 кг, 2 кг, 4 кг, 4 кг, 6 кг, 7 кг, 8 кг	10
7	1 кг, 1 кг, 2 кг, 2 кг, 2 кг, 4 кг, 5 кг, 7 кг, 8 кг	11
8	1 кг, 1 кг, 1 кг, 3 кг, 3 кг, 5 кг, 5 кг, 6 кг, 9 кг, 10 кг	13
9	1 кг, 1 кг, 1 кг, 3 кг, 3 кг, 5 кг, 5 кг, 6 кг, 9 кг, 12 кг	16
10	1 кг, 1 кг, 3 кг, 3 кг, 3 кг, 5 кг, 5 кг, 6 кг, 9 кг, 10 кг	9
11	1 кг, 1 кг, 1 кг, 3 кг, 3 кг, 4 кг, 4 кг, 6 кг	8
12	1 кг, 1 кг, 1 кг, 1 кг, 1 кг, 2 кг, 6 кг, 7 кг	7
13	1 кг, 1 кг, 2 кг, 4 кг, 5 кг, 6 кг, 7 кг, 8 кг	9
14	1 кг, 1 кг, 1 кг, 1 кг, 2 кг, 2 кг, 4 кг, 4 кг, 6 кг, 7 кг, 8 кг	11
15	1 кг, 1 кг, 2 кг, 2 кг, 2 кг, 4 кг, 5 кг, 7 кг	12

Например, необходимо набрать вес – 10 кг при имеющемся разновесе: 8 кг, 7 кг 3 кг, 2 кг, 2 кг, 1 кг, 1 кг, 1 кг.

Разновес удобно хранить в массиве KRATN[1..N], где N – вес самой тяжелой гири и KRATN[g] – количество экземпляров гири веса g.

Решение (набор веса) будем хранить в массиве SOL[1..N], где SOL[g] – количество экземпляров гири веса g в данном наборе.

На рис. 3.1 изображена примерная схема решения задачи.

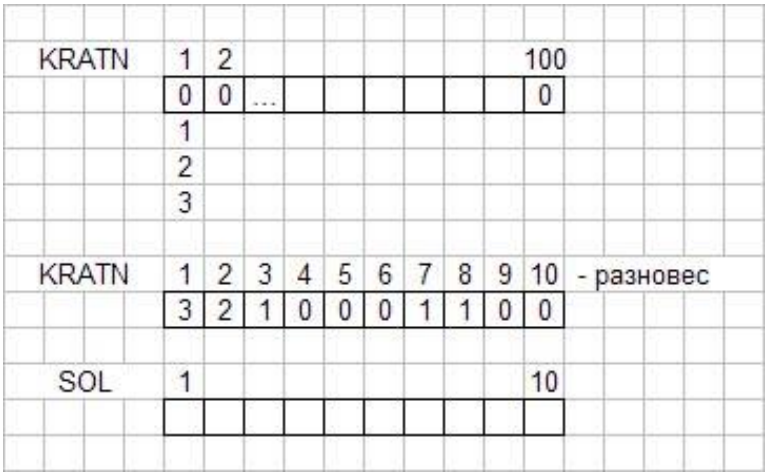


Рис. 3.1. Схема решения задачи

Условие допустимости добавления гири в решение:

Гиря *g* – допустима, если она имеется в разновесе, то есть *KRATN[g]>0* и ее вес не превышает набираемого веса *V (g≤V).*

Пусть такая *g* найдена, удалим ее из разновеса, то есть *KRATN[g]=KRATN[g]-1* и добавим к решению, то есть *SOL[g]=SOL[g]+1* новый текущий набираемый вес *V1=V-g.*

Реализация возврата:

Если допустимой гири в разновесе нет или найдено решение, то последнюю добавленную гирю удаляем из решения и добавляем в разновес:

KRATN[g]=KRATN[g]+1;

SOL[g]=SOL[g]-1;

V=V1+g – текущий набранный вес.

При такой схеме алгоритм будет генерировать повторяющиеся решения.

Контрольные вопросы:

1. Каким образом осуществляется хранение гирь в программе?
2. Перечислите условия допустимости для гирь.
3. Какие действия необходимо произвести при достижении условий допустимости?
4. Объясните, что означает понятие «возврат».
5. При каком условии осуществляется возврат?
6. Каким образом можно избежать повторяющихся решений?

Задача о костях домино*

У игрока имеется набор костей домино (не обязательно полный). Найти последовательности выкладывания этих костей таким образом, чтобы получившаяся в результате цепочка была максимальной длины.

Определим структуры для представления исходных данных и результатов работы программы.

```
Const N = 28; // максимальное количество костей домино
Type domino = record // одна кость домино
  left, right : 0..6; // значения
  used: Boolean; // использовалась ли кость
end;
F: array [1..N] of domino;  // набор костей домино у игрока
NF: integer;                // количество костей у игрока;
```

```
CurPos: array [1..N] of 1..N;    //текущий порядок выставления костей
BestPos: array [1..N] of 1..N;   //лучший порядок выставления костей
BestPosN: integer;               //количество костей в лучшем варианте
```

Правило выставления костей состоит в следующем: на первом ходу на стол выставляется произвольная кость, затем к ней могут быть добавлены совпадающие по значению кости как слева, так и справа. Одна и та же кость может быть использована только один раз. Основная программа вместе с инициализацией значений будет выглядеть следующим образом:

```
begin
writeln("Введите количество костей домино");
readln(NF);
for i:=1 to NF do
  begin
  write("Значения кости : "); readln( F[i].left, F[i].right);
  F[i].used:=false;
end;
BestPosN:=0;
For i:=1 to NF do
  begin
  CurPos[1]:=i; // выставляем i-ую кость в качестве первой
  F[i].used:=true; // помечаем как использованную
  Step(2,F[i].left, F[i].right); // вызов процедуры выставления 2-ой и т.д. кости
  F[i].used:=false;
  end;
```

Стоит обратить внимание на два момента в нашей программе. Первый – каждая кость, после того, как она выставлена, помечается как использованная. Это необходимо для предотвращения зацикливания программы в результате повторного использования кости. Этот флаг должен быть обязательно снят после пробы, чтобы данная кость могла быть использована в цепочке на другом месте, кроме первого.

Второй момент – для выставления второй, третьей и т.д. кости мы будем использовать ту же процедуру, так как эти действия аналогичны. При этом должны быть известны значения на концах цепочки, которая образована на данный момент.

Процедура *Step* должна попробовать в качестве следующей каждую кость из всех неиспользованных. После каждого удачного выставления необходимо перейти к следующему шагу и выставить очередную кость в цепочке.

```
// пытаемся поставить очередную кость в цепочку с границами left, right
Procedure Step (curn, left, right: integer);
Var I: integer;
Res: Boolean;
begin
//текущий вариант лучший по длине и мы его сохраняем
If (curn-1 > BestPosN) then
BestPos:= CurPos;
```

```
// просматриваем все кости как возможные варианты
For I:= 1 to NF do
If not f[i].used then
begin // пробуем поставить i-ую кость
F[i].used:= true;
CurPos[curn]:=i;
if F[i].left = left then //если кость подходит слева
  Step (curn+1, F[i].right, right ); // то пробуем следующую
if F[i].left = right then
Step(curn+1, left, F[i].right);
if F[i].right = left then
Step(curn+1, F[i].left, right);
if F[i].right = right then
Step(curn+1, left, F[i].left);
F[i].used:=false;
end;
end;
```

После вызова этой процедуры в массиве BestPos будет получено лучшее из возможных решений. Это может быть даже цепочка только из одной кости. Остается только напечатать полученное решение.

```
for i:=1 to BestPosN do
writeln(i:4, F[Bestpos[i]].left:4, F[BestPos[i]].right:4);
end.
```

Единственный недостаток нашей программы – мы не определяем, с какой стороны нужно на *i*-ом ходу поставить кость. Думаем, что читатели без труда решат эту проблему, изменив структуры, используемые в программе.

Фрагмент дерева поиска решения для данной задачи представлен на рисунке 3.2.

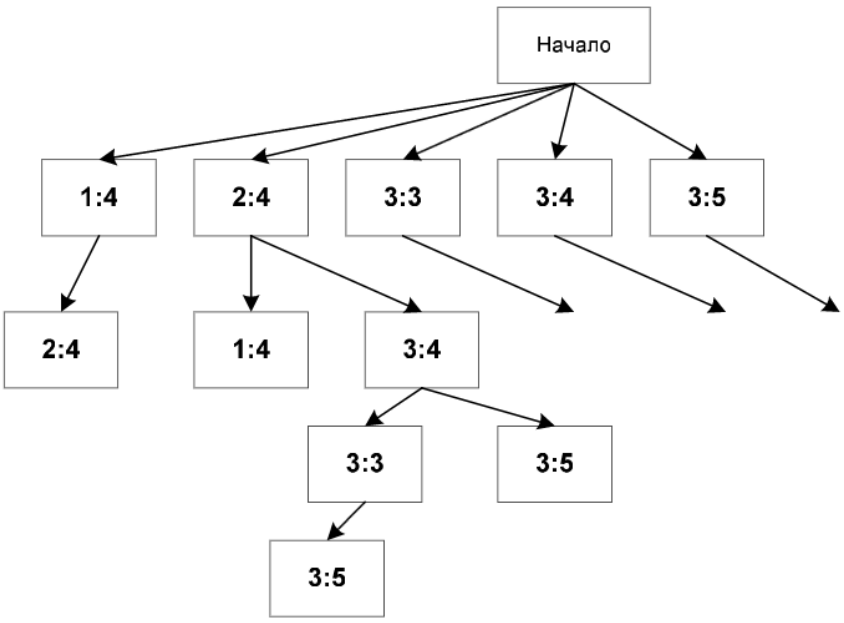


Рис. 3.2. Фрагмент дерева решений для задачи о костях домино

4. АЛГОРИТМ С ВОЗВРАТОМ ДЛЯ ЗАДАЧ НА МИНИМУМ. АЛГОРИТМ ВЕТВЕЙ И ГРАНИЦ

Задача коммивояжера

УСЛОВИЕ. Заданы конечное множество *C* городов, целые положительные расстояния $A[c_1,c_2]$ для каждой пары городов c_1,c_2.

ВОПРОС. Найти маршрут минимальной длины, проходящий через все города ровно один раз и возвращающийся в исходный пункт.

Это задача на минимум – среди всех допустимых объектов ищется минимальный. Карта представляется полным графом, ребрам которого приписаны веса. Из всех $\frac{(n-1)!}{2}$ гамильтоновых циклов в этом графе нужно выбрать цикл минимальной длины. Длина определяется естественным образом – как сумма расстояний между городами, входящими в маршрут.

Можно поступить просто – сгенерировать алгоритмом с возвратом все гамильтоновы циклы и среди них выбрать минимальный. Однако, немного изменив общую схему алгоритма с возвратом, можно добиться того, что возвраты будут происходить значительно раньше и решение будет найдено быстрее.

Прежде всего сформулируем условие, которому должна удовлетворять задача на минимум, чтобы ее можно было решать не общей схемой алгоритма с возвратом, а модификацией, которая будет рассмотрена ниже. Модификацией общей схемы алгоритма с возвращениями является алгоритм ветвей и границ.

Условие применимости модифицированной схемы для решения задачи на минимум

1. Припишем каждому решению (и полному, и частичному) стоимость – ее еще называют целевой функцией. Будем обозначать целевую функцию стоимости *cost*.
2. При продолжении решения его стоимость может только увеличиваться: $cost(<X[1], \dots, X[i-1]>) \le cost(<X[1], \dots, X[i-1], X[i]>)$ для любого $i \ge 2$.

Для задачи коммивояжера целевой функцией будет стоимость маршрута

$$\text{cost}(< X[1],..., X[n] >) = \sum_{i=2}^{n} A[X[i-1], X[i]].$$

Очевидно, что ее величина при удлинении маршрута может только увеличиваться, так как по условию задачи все расстояния положительны.

Алгоритм с возвращениями при обходе дерева пространства состояний использует стеки, поскольку рекурсивные функции хранят в стеках информацию о различных вызовах. **Метод ветвей и границ** основан на очередях. Алгоритм с возвращениями осуществляет поиск по глубине в дереве пространства состояний, в то время как метод ветвей и границ обходит это дерево по уровням. Это связано с тем, что обход графа в глубину осуществляется использованием стека, а обход графа в ширину с использованием очереди. Кроме того, если алгоритм с возвратом определяет перспективные узлы на основании того, может ли этот узел привести к решению задачи, **метод ветвей и границ** находит граничное число, и по этому числу определяет перспективность узла. Более точно, если вычисленная граница не лучше наилучшего решения, найденного к моменту вычисления, то дальше обходить дерево через эту вершину не имеет смысла, поскольку это не приведет к улучшению решения. Ветви и границы достаточно эффективны при решении задач оптимизации, где понятие «лучшее решение» могло бы означать бОльшее значение в задачах оптимизации, на поиск максимума типа задачи об укладке рюкзака или меньшее значение в задачах на минимум, например, в задачах о планировании работ, об упаковке в контейнеры.

Алгоритм ветвей и границ

1. Находим первое решение и запоминаем его в *OptX*, а его стоимость в *OptCost*.

2. Пусть <X[1], …, X[*i*]> – текущее частичное решение. Найдем допустимое X[*i*+1].

Если *cost*(<X[1], … , X[*i*], X[*i*+1]>) ≥ *OptCost*, то любое его продолжение будет заведомо больше текущего минимального решения. Отбрасываем X[*i*+1] как недопустимое и ищем новое X'[*i*+1].

Таким образом, произведя «досрочный» возврат, мы избавляем себя от генерации всех заведомо неоптимальных продолжений, т.е. обрубаем целое поддерево на дереве поиска.

Вопрос 4.1. Как находить первое решение, чтобы еще «ускорить» выполнение алгоритма?

Нахождение гамильтонова цикла минимальной длины

Данные: Граф *G*=<*V*,*E*>, представленный списками ЗАПИСЬ[*v*], матрица неотрицательных весов A[*u*,*v*].

Результаты: Печать минимального гамильтонова цикла.

Глобальные переменные: X, OptX, cost, OptCost, DOP.

```
1  procedure COMMI(i);
2  begin
3    for y Є ЗАПИСЬ[X[i-1]] do
4      if cost + A[X[i-1],y] < OptCost then
5        if (I = n + 1) AND (y = k) then
6          begin OptX:= X; OptCost:= cost+A[X[i-1],y]
7          end
8        else if DOP[y] then
9          begin
10           X[i]:= y; DOP[y]:= false;
11           cost:=cost+A[X[i-1],y];
12           COMMI(i+1);
13           DOP[y]:= true;
14           cost:=cost–A[X[i-1],y];
15         end;
16 end;


1 begin {основная программа}
2    for v Є V do DOP[v]:= true;
3    X[1]:= k ; {k – произвольная вершина}
4    DOP[k]:= false;
5    cost:=0; OptCost:= +∞;
6    COMMI(2);
7 end.
```

Ответ 4.1. Первое решение обычно находят отдельно приближенным алгоритмом. Если первое найденное решение будет близким к оптимальному, то произойдет максимальное количество отсечений неоптимальных продолжений.

Задача об упаковке в контейнеры

Постановка задачи: имеется набор предметов (множество *A*), каждый *i*-й предмет имеет размеры $0 < a_i < 1$; $i = 1,2, \ldots N$ и *N* контейнеров единичной емкости.

Вопрос: найти наименьшее количество контейнеров (ящиков), необходимое для раскладки всех предметов.

Другими словами, требуется разложить предметы по контейнерам так, чтобы сумма размеров предметов в каждом контейнере была не превышала 1 и число используемых контейнеров при этом было минимальным.

Необходимо использовать следующий способ упаковки: фиксировать текущий неупакованный предмет и перебирать контейнеры в поисках подходящего, пока *i*-й предмет не упакован, не пытаться упаковат *i+1*-ый.

Ниже рассмотрим точный алгоритм упаковки предметов по контейнерам.

N – число предметов и max возможное число контейнеров.

Глобальные переменные:

A[1,N] – размеры предметов

V[1,N] – заполненность контейнеров

Cont[1,N] – текущая упаковка

Cont[i] – номер контейнера, в котором лежит i-ый предмет

OptCont[1,N] – оптимальная упаковка

Box, OptBox – число использованных контейнеров текущей и оптимальной упаковки.

Основная программа:

```
1 begin {main}
2 {заполнение массива размера A;}
3   for i=1 to N do V[i]=0;
4   for j=1 to N do Cont[j]=0;
5   {текущая упаковка}
6   Cont[1]=1; V[1]=A[1]; Box=1; {начальное оптимальное решение
                                  каждый предмет в своем контейнере}
7   for p=1 to N do OptCont[p]=p;
8   OptBox=N;
9   pack(2);
10  печать оптимальной упаковки;
11end;
```

Рекурсивная процедура упаковки:

```
1  pack (i: integer)
2   begin
3      for j=1 to N do
4        if j – ый контейнер допустим then
5        begin
6          упаковываем i – ый предмет в j – ый контейнер;
7          if i=N then {новый оптимум}
8             begin OptBox=Box; OptCont=Cont;
9             end;
```

```
10          else pack (i+1); {возврат – вынимаем i – ый предмет
                              из j – ого контейнера};
11      end;
12  end;
```

Условие допустимости упаковки в контейнер:

1. V[j] + A[i] ≤ 1+ε,
   где ε – число, значительно меньшее объемов предметов
2. Если *j*-ый контейнер новый (V[j]≤ ε) то проверим:
   — Box + 1 < OptBox
   — (j > 1) и (j – 1) – не пуст

Процедура упаковки предмета:

1. B = Box; {запоминаем число контейнеров текущей упаковки}
2. if V[i] ≤ ε {начинаем новый контейнер}
   Box = Box+1;
3. V[j] = V[j] + A[i];
   Cont[i] = j;

Шаги возврата:

1 V[i]=V[i] – A[i]; {освобождаем контейнер на текущий вес предмета}

2 Cont[i] = 0;

3 Box = B; {восстанавливаем число контейнеров до упаковки i – го предмета}

PACK

```
 1  procedure pack(i);
 2  begin
 3    for j = 0 to N do
 4      if (V[i] ≤ eps) AND (Box+1 ≥ OptBox) then break;
 5      if (j > 1) AND (V[j–1] ≤ eps) then break;
 6      if V[j] + A[i] ≤ 1+eps then
 7      begin
 8        B = Box;
 9        if V[j] ≤ eps then Box = Box+1
10      Cont[i] = j; V[j] = V[j]+A[i];
11      if i < N then pack (i+1)
12      else
13      begin
14        for k=1 to N do OptCont[k]=Cont[k];
15        OptBox=Box;
16      end; {возврат}
17      V[j] = V[j] – A[i]; Cont[i] = 0;
18      Box = B;
19      end;
20      end; {for}
21      end;
```

Контрольные вопросы:

1. Назовите естественное ограничение данной задачи.
2. Какое условие необходимо проверять на каждой итерации решения задачи?
3. Сколько контейнеров будет заполнено в худшем случае?
4. При достижении какого условия рекурсия остановится?
5. Перечислите последовательность шагов при реализации «прямого хода» решения.
6. Каким образом и когда начинается перебор возможных вариантов упаковки?
7. Какова сложность данного алгоритма?
8. В каком случае необходимо увеличить число использованных контейнеров?

5. АЛГОРИТМ С ВОЗВРАТОМ ДЛЯ ЗАДАЧ НА МАКСИМУМ. ПРИНЦИП ВКЛЮЧЕНИЯ-НЕВКЛЮЧЕНИЯ

Задача о рюкзаке

УСЛОВИЕ. Заданы конечное множество A, положительные целые веса $w(a)$, стоимости $s(a)$ для каждого $a \in A$ и общее ограничение на вес K.

ВОПРОС. Найти из всевозможных выборок $A' \subseteq A$ такую, чтобы суммарный вес входящих в него элементов не превосходил K, а суммарная стоимость была максимальна.

Задача о рюкзаке – задача об определении оптимальной выборки из N объектов, подчиненной некоторому ограничению. Поскольку из N объектов возможно сделать 2^N различных выборок, для решения подобной задачи с помощью алгоритма с возвратом необходимо очень сильное ограничение.

***Вопрос* 5.1.** Изменим стоимости предметов на противоположные. Выборка максимальной стоимости превратилась в выборку минимальной стоимости. Почему нельзя эту задачу решить алгоритмом для решения задач на минимум, изложенным в предыдущем разделе?

Однако есть способ значительно сократить количество переборов – в процессе получения возможных решений помнить лучшее из полученных на данный момент решений и не пытаться генерировать те решения, которые будут заведомо хуже известного на данный момент оптимального решения.

Принцип включения–невключения

Каждый i-й объект проверяется на допустимость (согласно нашему ограничению). Затем внутри одной процедуры допустимый объект вначале включается в выборку, и с этим включенным объектом рекурсивно генерируются всевозможные решения и лучшее из них запоминается как оптимальное. После этого происходит возврат и i-й объект удаляется из выборки.

Классический алгоритм с возвратом стал бы рекурсивно генерировать всевозможные решения без i-го объекта – принцип включения–невключения действует иначе.

Прежде, чем пытаться получать решения без i-го объекта, проверим, можно ли, не включив этот объект, получить решение лучшее, чем найденное оптимальное с i-м объектом. Если нет – то не стоит и пытаться.

Как проверить возможность получения без i-го объекта более ценной выборки, чем текущая оптимальная, если мы не собираемся сразу строить эти выборки?

Нужно посчитать стоимость, которую мы, **может быть**, наберем без i-го объекта. Для этого просуммируем стоимости всех предметов, **уже вошедших в выборку**, и стоимости всех **еще неисследованных предметов** (разумеется, без i-го объекта).

Если эта стоимость, которую, может быть, удастся набрать (а, может, и не удастся – ведь неисследованные объекты могут не пройти ограничение по весу!) будет больше оптимальной – есть смысл генерировать решения без i-го объекта.

Опишем рекурсивную процедуру *TRY* с включением-невключением.

```
1  procedure try(i);
   {i – номер объекта}
2  begin
3    IF ВКЛЮЧЕНИЕ ПРИЕМЛЕМО THEN
4      begin
5        включить i-тый объект;
6        if i=N then проверить оптимальность
7        else try(i+1); {продолжить выборку}
8        удалить i-тый объект; {возврат}
```

```
9      end;  {оптимум с i-тым объектом запомнен, а сам объект удален}
10   IF НЕВКЛЮЧЕНИЕ ПРИЕМЛЕМО THEN
       {подсчет возможной стоимости без i-того
        объекта и сравнение ее с оптимальной}
11    if i = N then проверить оптимальность
12    else try(i+1);  {строим выборки без i-того объекта}
13 end;
```

Пусть имеется 3 предмета и ограничение по весу 16:

НОМЕР	1	2	3
ВЕС	5	8	11
СТОИМОСТЬ	10	8	15

Первый раз из основной программы вызывается try(1). Рассмотрим трассировку алгоритма:

try(i)	№ 1	№ 2	№ 3	Суммарный вес	Суммарная стоимость	Возможная стоимость	Оптимальная стоимость	Примечания
try(1)	+	?	?	5	10	33	0	Вкл. 1
try(2)	+	+	?	13	18	33	0	Вкл. 2
try(3)	+	+	–	13	18	18	18	Оптимум
try(2)	+	–	?	5	10	25	18	25>18 Невкл. 2
try(3)	+	–	+	16	25	25	25	Оптимум
try(1)	–	?	?	0	0	23	25	23<25 Невкл. 1 неприем.

Последний оптимум, равный 25 достигается на выборке [1, 3] и далее уже не меняется.

Прежде, чем записать алгоритм решения задачи о рюкзаке, сделаем полное описание глобальных переменных.

```
CONST N = 3;
    VES = 16;
    obj = record
              v,st: integer {вес и стоимость}
          end;
VAR A: array[1..N] of obj; {массив объектов}
      X, OptX: array[1..N]of boolean;
                {выборка, оптимальная выборка}
      sumcost: integer;{общая стоимость всех объектов}
      OptCost: integer; {оптимальная стоимость}
```

Рюкзак

Данные: Массив *A* записей *obj*, ограничение по весу *VES*.

Результаты: Выборка max ценности с ограничением по весу.

Глобальные переменные: A, X, OptX, sumcost, OptCost.

```
1 procedure My_try(i : index; sum_v, pos_st  : integer);
 {i – текущий объект, sum_v – вес текущей выборки,
```

```
  pos_st – возможная стоимость}
2  begin {включение}
3    if sum_v + A[i].v <= VES then
4      begin {проверка допустимости}
5        X[i] := false; {включаем i-й}
6        if (i = N) and (pos_st > OptCost) then
7          begin {новый оптимум}
8            OptX := X; OptCost := pos_st
9          end
10       else
11           My_try(i + 1, sum_v + A[i].v, pos_st);
         {строим решения с i-тым объектом, pos_st не изменилась,
         т.к. i-тый объект из неисследованных перешел во включенные}
12       X[i] := true;{возврат}
13     end; {проверка допустимости}
{невключение}
14   st1:= pos_st–A[i].st; {возможная ценность  без i-того объекта}
15   if st1 > OptCost then
16     if i=N then
17       begin {новый оптимум}
18         OptX:= X; OptCost:= st1
19       end
20     else     {строим решения без i-того объекта}
21       My_try(i+1, sum_v, st1);
22 end;
1  begin {основная часть}
2    sumcost := 0;
3    for i:=1 to N do
4      begin
5        sumcost := sumcost+A[i].st;
6        X[i]:= true; OptX[i]:= true;
7      end;
8    OptCost:=0;
9    My_try(1, 0, sumcost);
10   печать OptX, OptCost;
11 end.
```

***Вопрос* 5.2.** Почему первый вызов *My_try* происходит с возможной стоимостью, равной sumcost?

***Вопрос* 5.3.** Почему мы передаем суммарный вес рюкзака и возможную стоимость в виде параметров, а не вычисляем их внутри процедуры *My_try* в цикле за O(*N*) шагов?

***Ответ* 5.1**. При удлинении решения будет добавляться отрицательная стоимость, в результате чего будет происходить уменьшение целевой функции *cost*, в то время как она не должна убывать.

***Ответ* 5.2**. Все предметы не исследованы, sumcost – их суммарная стоимость.

***Ответ* 5.3**. В наихудщем случае обращение к *My_try* будет происходить 2^N раз и, соответственно, этот цикл будет выполняться 2^N раз.

Задание. Для закрепления решения задач с применением принципа включения-невключения, необходимо помимо программной реализации протрассировать алгоритм, как показано выше в примере. Исходные данные взять по вариантам из таблицы.

Таблица 5.1

Варианты решения задачи

№ варианта	Задание			
1	Номер предмета	1	2	3
	Вес предмета	5	8	11
	Стоимость предмета	10	8	15
	Суммарный вес рюкзака	16		
2	Номер предмета	1	2	3
	Вес предмета	7	4	12
	Стоимость предмета	9	8	10
	Суммарный вес рюкзака	15		
3	Номер предмета	1	2	3
	Вес предмета	5	7	13
	Стоимость предмета	15	4	11
	Суммарный вес рюкзака	18		
4	Номер предмета	1	2	3
	Вес предмета	6	9	14
	Стоимость предмета	9	7	12
	Суммарный вес рюкзака	17		
5	Номер предмета	1	2	3
	Вес предмета	4	6	10
	Стоимость предмета	7	8	13
	Суммарный вес рюкзака	15		
6	Номер предмета	1	2	3
	Вес предмета	7	3	13
	Стоимость предмета	9	3	17
	Суммарный вес рюкзака	18		
7	Номер предмета	1	2	3
	Вес предмета	6	3	15
	Стоимость предмета	5	7	9
	Суммарный вес рюкзака	19		
8	Номер предмета	1	2	3
	Вес предмета	9	10	7
	Стоимость предмета	8	12	5
	Суммарный вес рюкзака	16		
9	Номер предмета	1	2	3
	Вес предмета	8	6	5
	Стоимость предмета	6	7	8
	Суммарный вес рюкзака	17		

Окончание табл. 5.1

№ варианта	Задание			
10	Номер предмета	1	2	3
	Вес предмета	4	7	5
	Стоимость предмета	6	9	8
	Суммарный вес рюкзака	15		
11	Номер предмета	1	2	3
	Вес предмета	3	9	5
	Стоимость предмета	7	6	8
	Суммарный вес рюкзака	14		
12	Номер предмета	1	2	3
	Вес предмета	5	8	16
	Стоимость предмета	9	5	10
	Суммарный вес рюкзака	20		
14	Номер предмета	1	2	3
	Вес предмета	13	5	9
	Стоимость предмета	9	11	8
	Суммарный вес рюкзака	15		
15	Номер предмета	1	2	3
	Вес предмета	10	13	12
	Стоимость предмета	7	6	9
	Суммарный вес рюкзака	24		

Таблица 5.2

Пример таблицы для трассировки алгоритма

try(i)	N1	N2	N3	∑ вес рюкзака	∑ стоим. рюкзака	Возможная стоимость	Оптимальн. стоимость	Примечание
try(1)	+	?	?	Вес N1	Стоим. N1	∑ стоим. предметов	–	Включили 1-й предмет
					…			

Контрольные вопросы:

1. Объясните принцип включения-невключения.
2. В чем его отличие от классического алгоритма с возвратом?
3. Как рассчитывается «возможная стоимость» рюкзака?
4. Что такое «оптимальная стоимость»?
5. В какой момент завершается ветка «включения» и какие действия начинаются далее?
6. Что означают понятия «приемлемое включение» и «приемлемое невключение»?

6. ПРИБЛИЖЕННЫЙ МЕТОД РЕШЕНИЯ ЗАДАЧ ИЗ КЛАССА NP НА ПРИМЕРЕ ЗАДАЧИ ОБ УПАКОВКЕ В КОНТЕЙНЕРЫ

Задачи из класса NP имеют большое число приложений, поэтому их решение представляет значительный интерес. Поскольку полиномиальные алгоритмы решения этих задач неизвестны, существуют альтернативные возможности, дающие лишь достаточно хорошие ответы. Такой приближенный алгоритм может дать иногда и оптимальный ответ, однако на такую случайность рассчитывать нельзя.

Если размерность задачи такова, что вычисление с помощью точного алгоритма не укладывается в реальное время, а для большинства реальных задач именно так и бывает, то следует отказаться от попыток найти оптимальное решение и отправиться на поиски решения хорошего и устраивающего заказчика.

Приближённые методы решения имеют смысл только для оптимизационных задач.

Пусть определена *P*-задача о нахождении экстремума (либо минимума, либо максимума).

Обозначение:

1. *Dp* – множество индивидуальных задач задачи *P*.

2. Для каждой индивидуальной задачи *Sp(I)* – конечное множество допустимых решений.

3. *Mp* – целевая функция сопоставляющая каждому допустимому решению $x \in Sp(I)$ величину решения *Mp(I,x).*

4. *X** – оптимальное решение, на котором целевая функция достигает максимума (минимума). OPT(I) – величина оптимального решения задачи I.

Для задачи об оптимальной упаковке в контейнеры существует приближенный алгоритм:

FFD (first fit decreasing) – первый подходящий элемент в порядке убывания.

Идея – отсортировать предметы в порядке убывания размеров и упаковать предметы в первый подходящий контейнер.

Человек практичный никогда не останавливается на доказательстве того, что задача является NP-полной. Очевидно, у него были какие-то причины решить ее. Эти причины не исчезли, когда он узнал, что для решения задачи не существует алгоритма с полиномиальным временем исполнения. Ему все равно нужна программа для решения этой задачи. Все, что известно, так это лишь невозможность создания программы для быстрого оптимального решения задачи в наихудшем случае.

Однако для достижения цели остаются три варианта:

✓ алгоритмы, эффективные для средних случаев задачи. В качестве примеров таких алгоритмов можно назвать алгоритмы поиска с возвратом, в которых выполняются значительные отсечения;

✓ эвристические алгоритмы. Эвристические методы, такие как имитация отжига или жадные алгоритмы, можно использовать, чтобы быстро найти решение, но без гарантии, что это решение будет наилучшим;

✓ аппроксимирующие алгоритмы. Теория NP-полноты только оговаривает сложность получения решения задачи. Но используя специализированные, ориентированные на конкретную задачу эвристические алгоритмы, скорее всего можно получить близкое к оптимальному решение для всех возможных экземпляров задачи.

Аппроксимирующие алгоритмы обеспечивают решение с гарантией того, что оптимальное решение не будет намного лучше. Таким образом, используя аппроксимирующий алгоритм для решения NP-полной задачи, вы никогда не получите крайне неудовлетворительный ответ. Независимо от входного экземпляра задачи, вы неизбежно будете совершать

правильные действия. Кроме того, аппроксимирующие алгоритмы с удачно подобранными границами концептуально просты, работают быстро и легко поддаются программированию.

Однако неясным остается следующее обстоятельство. Насколько решение, полученное с помощью аппроксимирующего алгоритма, сравнимо с решением, которое можно было бы получить с помощью эвристического алгоритма, не дающего никаких гарантий? Вообще говоря, оно может оказаться как лучше, так и хуже. Положив деньги в банк, вы получаете гарантию невысоких процентов прибыли без всякого риска. Вложив эти же деньги в акции, вы, скорее всего, получите более высокую прибыль, но при этом у вас не будет никаких гарантий.

Один из способов получения наилучшего результата от аппроксимирующего и эвристического алгоритмов заключается в том, что вы решаете данный экземпляр задачи с помощью каждого из них и выбираете тот алгоритм, который дает лучшее решение. Таким образом, вы получаете гарантированное решение и дополнительный шанс на получение лучшего решения. При применении эвристических алгоритмов для решения сложных задач вы можете рассчитывать на двойной успех.

Задание. Реализовать FFD-алгоритм для задачи об упаковке в контейнеры на языке программирования высокого уровня.

Контрольные вопросы:

1. Объясните понятия «вычислительная сложность алгоритма» и «сложность задачи».
2. Что такое NP-полная задача?
3. Какая задача называется индивидуальной?
4. Что представляет собой целевая функция для данной задачи?
5. Какой алгоритм называется приближенным?
6. Что такое величина оптимального приближения и как она рассчитывается?
7. В чем отличие данного алгоритма от точного?
8. Когда целесообразнее использовать приближенный алгоритм решения задачи?

ПРИЛОЖЕНИЕ
СПИСОК NP–ПОЛНЫХ ЗАДАЧ

Многие задачи из этого списка допускает две возможных постановки:

– задача оптимизации;

– задача распознавания.

В качестве примера рассмотрим задачу о коммивояжере.

Оптимизационная постановка выглядит так:

УСЛОВИЕ. Заданы конечное множество C городов, целые положительные расстояния $d(c_1,c_2)$ для каждой пары городов c_1, c_2.

ВОПРОС ОПТИМИЗАЦИИ. Найти маршрут минимальной длины, проходящий через все города и возвращающийся в исходный пункт.

Чтобы получить теперь задачу распознавания, введем дополнительный параметр – числовую границу B.

ВОПРОС РАСПОЗНАВАНИЯ. Существует ли маршрут, проходящий через все города, длина которого не превосходит B, где B – целое положительное число? (Если решается задача на максимум, то условие «не превосходит B» заменяется условием «не меньше B».)

Традиционно к NP-полным задачам относятся задачи распознавания, однако соответствующие им задачи оптимизации эквивалентны им по сложности, поэтому при формулировке задач можно использовать любую постановку.

1. РАСКРАШИВАЕМОСТЬ ГРАФА (ХРОМАТИЧЕСКОЕ ЧИСЛО)

УСЛОВИЕ. Заданы граф $G=<V, E>$ и положительное целое число $K \le |V|$, где $|V|$ – число вершин графа.

ВОПРОС. Верно ли, что граф K–раскрашиваем? (Граф называется K–раскрашиваемым, если его вершины можно раскрасить в K цветов так, чтобы любые две соседние вершины были окрашены в различные цвета).

2. КЛИКА

УСЛОВИЕ. Заданы граф $G=<V, E>$ и положительное целое число $K \le |V|$.

ВОПРОС. Верно ли, что граф G содержит клику размера не менее K?

(Существует ли подграф графа G такой, что число вершин в нем не меньше K и любые две вершины соединены ребром?)

3. ВЕРШИННОЕ ПОКРЫТИЕ

УСЛОВИЕ. Заданы граф $G = <V, E>$ и положительное целое число $K \le |V|$.

ВОПРОС. Имеется ли в графе вершинное покрытие не более чем из K элементов? (Имеется ли такое подмножество вершин $V' \subseteq V$ мощности не более K, что для любого ребра $(u–v) \in V$ хотя бы одна из вершин u, v принадлежит V'?)

4. РАЗБИЕНИЕ

УСЛОВИЕ. Заданы конечное множество A и положительный целый вес $w(a)$ для каждого $a \in A$.

ВОПРОС. Существует ли подмножество $A' \subseteq A$ такое, что суммарный вес элементов, принадлежащих A', равен суммарному весу элементов, не принадлежащих A'? (Иными словами, можно ли A разбить на два подмножества, равные по весу?)

5. РЮКЗАК

УСЛОВИЕ. Задано конечное множество A, положительные целые веса $w(a)$, стоимости $s(a)$ для каждого $a \in A$ и общее ограничение на вес K.

ВОПРОС. Найти из всевозможных выборок $A' \subseteq A$ такую, чтобы суммарный вес входящих в него элементов не превосходил K, а суммарная стоимость была максимальна.

6. РЮКЗАК С КРАТНЫМ ВЫБОРОМ ЭЛЕМЕНТОВ

УСЛОВИЕ. Задано конечное множество A, положительные целые веса $w(a)$, стоимости $s(a)$ для каждого $a \in A$, общее ограничение на вес K и минимальная стоимость B.

ВОПРОС. Можно ли так сопоставить каждому элементу $a \in A$ целое число $c(a)$ (кратность), чтобы суммарный вес всех предметов из A с учетом кратностей не превосходил K, а суммарная стоимость тех же предметов была не меньше B?

7. УПАКОВКА В КОНТЕЙНЕРЫ

УСЛОВИЕ. Заданы конечное множество A и размеры $s(a) \in [0, 1]$ каждого предмета.

ВОПРОС. Найти такое разбиение множества A на непересекающиеся A_1, A_2, …, A_k, чтобы сумма размеров предметов в каждом A_i не превосходила 1 и k было минимальным.

8. ДИНАМИЧЕСКОЕ РАСПРЕДЕЛЕНИЕ ПАМЯТИ

УСЛОВИЕ. Заданы множество элементов данных A, целые положительные $s(a)$ – размер каждого элемента, $r(a)$ – время его поступления, $d(a)$ – время его жизни, положительное целое число D – размер области памяти.

ВОПРОС. Существует ли для множества данных A допустимое распределение памяти? Иными словами, существует ли такая функция $\xi: A \to \{1, 2, \ldots, D\}$, что для каждого элемента $a \in A$ интервал $(\xi(a), \xi(a) + s(a) - 1)$ (т.е. место, занимаемое им в динамической памяти) содержался бы в $[1, D]$ и в любой фиксированный момент времени t_0 интервалы для данных с $r(a) \le t_0 \le r(a)+d(a)$ не пересекались?

9. МНОЖЕСТВО ПРЕДСТАВИТЕЛЕЙ

УСЛОВИЕ. Задано семейство C подмножеств множества S, положительное целое число K.

ВОПРОС. Содержит ли S множество представителей для C, т.е. существует ли в S подмножество S' мощности не более K такое, что S' содержит, по крайней мере, один элемент из каждого множества семейства C.

10. УПОРЯДОЧЕНИЕ ВНУТРИ ИНТЕРВАЛОВ

УСЛОВИЕ. Задано конечное множество заданий T и для каждого задания $t \in T$ целое число $r(t) \ge 0$ – время готовности, целые положительные $d(t)$ и $l(t)$ – директивный срок и длительность.

ВОПРОС. Существует ли для T допустимое расписание, т.е. функция $\xi: T \to N$ (N – множество натуральных чисел), сопоставляющая заданию t момент начала его выполнения $\xi(t)$? (Иными словами, задание t выполняется с момента времени $\xi(t)$ до $\xi(t)+l(t)$, оно не может быть начато ранее момента $r(t)$, должно быть закончено не позднее $d(t)$, и временной интервал выполнения одного задания не может перекрываться с интервалом другого).

11. МИНИМИЗАЦИЯ ШТРАФА ЗА НЕВЫПОЛНЕНИЕ ЗАДАНИЙ

УСЛОВИЕ. Задано конечное множество заданий T и для каждого задания $t \in T$ целые положительные d(t) – директивный срок, l(t) – длительность и w(t) – вес, а также целое положительное число K.

ВОПРОС. Существует ли для T однопроцессорное расписание такое, что сумма $\sum_t (\sigma(t) + l(t) - d(t))w(t)$, взятая по тем $t \in T$, для которых $\sigma(t) + l(t) > d(t)$, не превосходит K?

(Однопроцессорное расписание – это функция ξ: $T \to N$ (N – множество натуральных чисел), сопоставляющая заданию t момент начала его выполнения $\xi(t)$ такое, что задание t выполняется с момента времени $\xi(t)$ до $\xi(t)$+l(t), оно должно быть закончено не позднее d(t), и временной интервал выполнения одного задания не может перекрываться с интервалом другого).

12. МИНИМАКСНОЕ МНОЖЕСТВО ЦЕНТРОВ

УСЛОВИЕ. Заданы граф G=<V, E> , положительный целый вес w(v) каждой вершины $v \in V$, положительная целая длина l(e) каждого ребра $e \in E$; положительное целое число $K \leq |V|$ и положительное рациональное число B.

ВОПРОС. Существует ли такое множество P мощности K «точек на G» (точка на G либо вершина графа, либо точка на ребре, где ребро e рассматривается как прямолинейный отрезок длины l(e)), что если d(v) – длина кратчайшего пути от вершины v до ближайшей к ней точке из P, то $\max\{d(v) \times w(v) : v \in V\} \leq B$?

Комментарий. Вариант этой задачи, в котором P – подмножество вершин, также NP-полон, но если K фиксировано или G – дерево, то разрешим за полиномиальное время.

13. МНОЖЕСТВО ЦЕНТРОВ С МИНИМАКСНОЙ СУММОЙ

УСЛОВИЕ. Заданы граф G=<V, E>, положительный целый вес w(v) каждой вершины $v \in V$, положительная целая длина l(e) каждого ребра $e \in E$; положительное целое число $K \leq |V|$ и положительное рациональное число B.

ВОПРОС. Существует ли такое множество P мощности K «точек на G» (точка на G либо вершина графа, либо точка на ребре, где ребро e рассматривается как прямолинейный отрезок длины l(e)), что если d(v) – длина кратчайшего пути от вершины v до ближайшей к ней точке из P, то $\sum_{v \in V} d(v) \times w(v) \leq B$?

Комментарий. Вариант этой задачи, в котором P – подмножество вершин, также NP-полон, но если K фиксировано или G – дерево, то разрешим за полиномиальное время.

14. СКОПЛЕНИЯ

УСЛОВИЕ. Заданы конечное множество X, целые положительныерасстояния d(x, y) для каждой пары x, y элементов из X, целые положительные K и B.

ВОПРОС. Существует ли такое разбиение множества X на непересекающиеся X_1, X_2, …, X_k, что для всех $x, y \in X_i$ d(x, y) $\leq B$ при $1 \leq i \leq k$?.

15. СОСТАВЛЕНИЕ УЧЕБНОГО РАСПИСАНИЯ

УСЛОВИЕ. Заданы множество H рабочих часов, множество C преподавателей, множество T учебных дисциплин, для каждого преподавателя c дано подмножество A(c) из H, называемое «допустимыми часами преподавателя c», для каждой дисциплины t подмноже-

ство A(t) множества H, называемое «допустимыми часами дисциплины t», и для каждой пары (c, t) ∈ C×T целое положительное число R(c, t), называемое «требуемой нагрузкой».

ВОПРОС. Существует ли учебное расписание, обслуживающее все дисциплины? Иными словами, существует ли функция f: C×T×H → {0,1} (где f(c, t, h) = 1 означает, что преподаватель c занимается дисциплиной t в момент h), удовлетворяющая следующим условиям:

– f(c, t, h) = 1 только тогда, когда h ∈ пересечению A(c) и A(t);
– для каждого h ∈ H и c ∈ C существует не более одного t ∈ T такого, что f(c, t, h) = 1;
– для каждого h ∈ H и t ∈ T существует не более одного c ∈ C такого, что f(c, t, h) = 1;
– для каждой пары (c, t) ∈ C×T существует ровно R(c, t) значений h, для которых f(c, t, h) = 1.

16. МИНИМАЛЬНОЕ ПОКРЫТИЕ

УСЛОВИЕ. Задано семейство C подмножеств конечного множества S и положительное целое число $K \le |C|$.

ВОПРОС. Существует ли покрытие C' из C мощности не более K? Иными словами, существует ли в C такое подмножество C', что любой элемент из S принадлежит, по крайней мере, одному подмножеству из C'?

17. МИНИМАЛЬНЫЙ НАБОР ТЕСТОВ

УСЛОВИЕ. Задано конечное множество A возможных диагнозов, набор C подмножеств множества A, представляющий тесты, и положительное целое число $J \le |C|$.

ВОПРОС. Существует ли в C поднабор C' мощности не более J, такой, что для любой пары A_i, A_j возможных диагнозов имеется некоторый тест c ∈ C', содержащий ровно один из них?

18. САМЫЙ ДЛИННЫЙ ПУТЬ

УСЛОВИЕ. Заданы граф G=<V, E>, целые положительные длины l(e) для всех ребер, выделенные вершины s и t и положительное целое число B.

ВОПРОС. Существует ли в G элементарный путь из s в t, имеющий длину не менее B?

19. НЕЗАВИСИМОЕ МНОЖЕСТВО

УСЛОВИЕ. Заданы граф G=<V, E> и положительное целое число $K \le |V|$.

ВОПРОС. Существует ли в V независимое множество мощности не менее K? Иными словами, верно ли, что существует подмножество V′ ∈ V такое, что |V′| ≥ K и никакие две вершины из V′ не соединены ребром из E?

20. КУБИЧЕСКИЙ ПОДГРАФ

УСЛОВИЕ. Задан граф G=<V, E>.

ВОПРОС. Существует ли в E непустое подмножество E' такое, что в графе G'=<V, E'> любая вершина имеет степень, равную 3 или 0?

21. МИНИМАЛЬНЫЙ РАЗРЕЗ

УСЛОВИЕ. Заданы граф G=<V, E>, вес w(e) каждого ребра e ∈ E и положительное целое число K.

ВОПРОС. Существует ли разбиение множества V на два таких непересекающихся множества V_1 и V_2, что сумма весов ребер из E, соединяющих вершины из множеств V_1 и V_2, не превосходит K?

22. МИНИМАЛЬНЫЙ РАЗРЕЗ С ОГРАНИЧЕНИЯМИ

УСЛОВИЕ. Заданы граф $G=<V, E>$ с двумя выделенными вершинами s и t, вес $w(e)$ каждого ребра $e \in E$ и положительные целые числа B и K ($B \le |V|$).

ВОПРОС. Существует ли разбиение множества V на два таких непересекающихся множества V_1 и V_2, что $s \in V_1$, $t \in V_2$, $|V_1| \le B$, $|V_2| \le B$ и сумма весов ребер из E, соединяющих вершины из множеств V_1 и V_2, не превосходит K?

23. НАДЕЖНОСТЬ СЕТИ

УСЛОВИЕ. Заданы граф $G=<V, E>$, подмножество $V' \in V$, для каждого $e \in E$ рациональное число $p(e)$, $0 \le p(e) \le 1$ – вероятность неисправности, положительное рациональное число $q \le 1$.

ВОПРОС. Верно ли, что с вероятностью, не меньшей q, любые две вершины из V соединены хотя бы одним путем, не содержащим неисправных ребер?

24. КРАТЧАЙШИЙ ПУТЬ С ОГРАНИЧЕНИЯМИ ПО ВЕСУ

УСЛОВИЕ. Заданы граф $G=<V, E>$ с двумя выделенными вершинами s и t, целые положительные веса $w(e)$ и длина $l(e)$ каждого ребра $e \in E$ и положительные целые числа B и K.

ВОПРОС. Существует ли в G элементарный путь из s в t, веса не более B и длины не более K?

25. СЕЛЬСКИЙ ПОЧТАЛЬОН

УСЛОВИЕ. Заданы граф $G=<V, E>$, целая положительная длина $l(e)$ каждого ребра $e \in E$, E' – подмножество E и положительное целое число K.

ВОПРОС. Существует ли в G цикл, включающий каждое ребро из E' и имеющий длину не более K?

26. КИТАЙСКИЙ ПОЧТАЛЬОН

УСЛОВИЕ. Заданы смешанный граф граф $G=<V, A, E>$, где A – множество ориентированных ребер и E – множество неориентированных ребер; целая положительная длина $l(e)$ каждого ребра $e \in A \cup E$, и положительное целое число B.

ВОПРОС. Существует ли в G цикл, включающий по крайней мере один раз каждое ребро и имеющий длину не более B (ориентированные ребра должны входить в цикл только с правильной ориентацией)?

27. МИНИМАЛЬНОЕ ПО МОЩНОСТИ МАКСИМАЛЬНОЕ ПАРОСОЧЕТАНИЕ

УСЛОВИЕ. Заданы граф $G=<V, E>$ и положительное целое число $K \le |E|$.

ВОПРОС. Существует ли в E непустое подмножество E' такое, что $|E'| \le K$ и E'- максимальное паросочетание, т.е. никакие два ребра из E' не имеют общего конца, а любое ребро из множества E-E' имеет общий конец с одним из ребер множества E'?

28. СУММА ВЕСОВ

УСЛОВИЕ. Заданы конечное множество A, положительные целые веса $w(a)$ для каждого $a \in A$ и положительный целый вес K.

ВОПРОС. Существует ли такое подмножество A' множества A, что сумма весов его элементов равна K?

29. ДОМИНИРУЮЩЕЕ МНОЖЕСТВО

УСЛОВИЕ. Заданы граф $G=<V, E>$ и положительное целое число $K \le |V|$.

ВОПРОС. Существует ли в V доминирующее множество мощности не более K? Иными словами, верно ли, что существует подмножество $V' \in V$ такое, что $|V'| \le K$ и для любой вершины $u \in V-V'$ сущеествует такая вершина $v \in V'$, что u и v соединены ребром из E?

30. РАЗБИЕНИЕ НА КЛИКИ

УСЛОВИЕ. Заданы граф $G=<V, E>$ и положительное целое число $K \le |V|$.

ВОПРОС. Можно ли разбить вершины графа G на $k \le K$ непересекающихся множеств $V_1,V_2,\ldots,V_k$ таких, что для всех i $(1 \le i \le k)$ подграф, индуцированный множеством V_i, был бы полным?

31. РАЗБИЕНИЕ НА ТРЕУГОЛЬНИКИ

УСЛОВИЕ. Заданы граф $G=<V, E>$ такой, что для некоторого положительного целого число q: $|V|=3q$.

ВОПРОС. Можно ли разбить вершины графа G на q непересекающихся множеств $V_1,V_2,\ldots,V_q$ таких, что каждое содержит ровно 3 вершины и для всех i $(1 \le i \le q)$ подграф, индуцированный множеством V_i, был бы треугольником?

32. РАЗБИЕНИЕ НА ЛЕСА

УСЛОВИЕ. Заданы граф $G=<V, E>$ и положительное целое число $K \le |V|$.

ВОПРОС. Можно ли разбить вершины графа G на $k \le K$ непересекающихся множеств $V_1,V_2,\ldots,V_k$ таких, что для всех i $(1 \le i \le k)$ подграф, индуцированный множеством V_i, не содержит циклов?

33. МОНОХРОМАТИЧЕСКИЙ ТРЕУГОЛЬНИК

УСЛОВИЕ. Заданы граф $G=<V,E>$.

ВОПРОС. Можно ли разбить множество ребер E на два непересекающихся подмножества E_1 и E_2, таких, что ни один из графов $G_1=<V, E_1>$ или $G_2=<V, E_2>$ не содержит треугольника?

34. ИНДУЦИРОВАННЫЙ ПУТЬ

УСЛОВИЕ. Заданы граф $G=<V, E>$ и положительное целое число $K \le |V|$.

ВОПРОС. Существует ли в V подмножество V' такое, что $|V'| \ge K$ и подграф, индуцированный множеством V', является элементарным путем на $|V'|$ вершинах?

35. ДВУДОЛЬНЫЙ ПОДГРАФ

УСЛОВИЕ. Заданы граф $G=<V, E>$ и положительное целое число $K \le |E|$.

ВОПРОС. Существует ли в E подмножество E' мощности не менее K такое, что $G'=<V, E'>$ – двудольный подграф?

36. СВЯЗНЫЙ ПОДГРАФ ОГРАНИЧЕННОЙ СТЕПЕНИ

УСЛОВИЕ. Заданы граф $G=<V, E>$, неотрицательное целое число $d \le |V|$ и положительное целое число $K \le |E|$.

ВОПРОС. Существует ли в E подмножество E' мощности не менее K такое, что подграф $G'=<V, E'>$ связен и не имеет вершин степени более d?

37. ГАМИЛЬТОНОВО ПОПОЛНЕНИЕ

УСЛОВИЕ. Заданы граф $G=<V, E>$ и положительное целое число $K \le |V|$.

ВОПРОС. Существует ли множество E', содержащее множество E такое, что $|E'-E| \le K$, а граф $G'=<V, E'>$ имеет гамильтонов цикл?

38. КАРКАС ОГРАНИЧЕННОГО ДИАМЕТРА

УСЛОВИЕ. Заданы граф $G=<V, E>$, вес положительный целый вес $w(e)$ каждого ребра $e \in E$ и положительные целые числа B и $D \le |V|$.

ВОПРОС. Существует ли в графе G каркас T такой, что сумма весов ребер из T не превосходит B и в T нет элементарного пути, число ребер которого не превосходит D?

39. РАСЩЕПЛЕНИЕ МНОЖЕСТВА

УСЛОВИЕ. Задан набор C подмножеств множества S.

ВОПРОС. Существует ли такое разбиение множества S на два подмножества, что ни одно подмножество из C не содержится целиком ни в S_1, ни в S_2?

40. ЗАДАЧА О *P*-ЦЕНТРЕ

УСЛОВИЕ. Заданы граф $G=<V, E>$, целый неотрицательный вес $w(v)$ каждой вершины $v \in V$, целая неотрицательная длина $A(e)$ каждого ребра $e \in E$, положительное целое число $K \le |V|$ и положительное рациональное число B.

ВОПРОС. Существует ли такое множество P мощности K , состоящее из точек на G (точкой на G называется либо вершина графа G, либо точка на ребре $e \in E$, где e рассматривается как прямолинейный отрезок длины $A(e)$), что если $D(v)$ – длина кратчайшего пути от вершины v до ближайшей к ней точке из P, то $Max\{D(v) \times w(v)\} \le B$?

Комментарий. При $P \subseteq V$ (выборе центра среди вершин) задача остается NP-полной. Если граф G – дерево, или K – фиксировано, то задача разрешима за полиномиальное время.

41. ЗАДАЧА О *P*-МЕДИАНЕ

УСЛОВИЕ. Заданы граф $G=<V, E>$, целый неотрицательный вес $w(v)$ каждой вершины $v \in V$, целая неотрицательная длина $A(e)$ каждого ребра $e \in E$, положительное целое число $K \le |V|$ и положительное рациональное число B.

ВОПРОС. Существует ли такое множество P мощности K , состоящее из точек на G (точкой на G называется либо вершина графа G, либо точка на ребре $e \in E$, где e рассматривается как прямолинейный отрезок длины $A(e)$), что если $D(v)$ – длина кратчайшего пути от вершины v до ближайшей к ней точке из P, то

$Max\{D(v) \times w(v)\} \le B$?

Комментарий. При $P \subseteq V$ (выборе медианы среди вершин) задача остается NP-полной. Если граф G – дерево, или K – фиксировано, то задача разрешима за полиномиальное время.

42. МАКСИМАЛЬНОЕ ЧИСЛО ОГРАНИЧЕННЫХ НЕПЕРЕСЕКАЮЩИХСЯ ПУТЕЙ

УСЛОВИЕ. Задан граф $G=<V, E>$ с выделенными вершинами s и t и заданы положительные целые числа B и K (B и $K \le |V|$).

ВОПРОС. Верно ли что в G содержится не менее B путей из s в t, попарно не имеющих общих вершин и включающих не более K ребер?

43. КАРКАС ОГРАНИЧЕННОЙ СТЕПЕНИ

УСЛОВИЕ. Заданы граф $G=<V, E>$ и положительное целое число $K \le |V|$.

ВОПРОС. Существует ли в графе G каркас, у которого степени всех вершин не превосходят K?

44. КАРКАС С МАКСИМАЛЬНЫМ ЧИСЛОМ ЛИСТОВ

УСЛОВИЕ. Заданы граф $G=<V, E>$ и положительное целое число $K \le |V|$.

ВОПРОС. Существует ли в графе G каркас, у которого не менее K вершин степени 1?

45. ДЕРЕВО ШТЕЙНЕРА

УСЛОВИЕ. Заданы граф $G=<V, E>$, положительный целый вес $w(e)$ каждого ребра $e \in E$, подмножество R вершин графа и положительное целое число B.

ВОПРОС. Существует ли в графе G дерево, содержащее все вершины из R, что сумма весов ребер этого поддерева не превосходит B?

46. МАКСИМАЛЬНОЕ ЧИСЛО ОГРАНИЧЕННЫХ НЕПЕРЕСЕКАЮЩИХСЯ ПУТЕЙ

УСЛОВИЕ. Заданы граф $G=<V, E>$ с выделенными вершинами s и t, а также положительные целые числа J и K.

ВОПРОС. Верно ли, что в G содержится не менее J различных путей из s в t, попарно не имеющих общих вершин и включающих не более K ребер?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Липский В. Комбинаторика для программистов. М.: Мир, 1988.
2. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. М.: ИД «Вильямс», 2005.
3. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.
4. Филлипс Д., Гарсиа-Диас А. Методы анализа сетей. М.: Мир, 1984.
5. Кристофидес Н. Теория графов. Алгоритмический подход. М.: Мир, 1978.
6. Майника Э. Алгоритмы оптимизации на сетях и графах. М.: Мир, 1981.
7. Вирт Н. Алгоритмы + структуры данных = программы. М.: Мир, 1985.
8. Свами М., Тхуласираман К. Графы, сети и алгоритмы. М.: Мир, 1984.
9. Кнут Д. Искусство программирования: В 3 т. 3-е изд. ИД «Вильямс», 2001.
10. Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы. Вильямс, 2000.
11. Торчинский В.Е., Файнштейн С.И. Структуры и алгоритмы обработки данных на ЭВМ. Магнитогорск, МГТУ, 2007.
12. Г.-Д. Эббинхауз, К. Якобс, Ф.-К. Ман, Г. Хермес Машины Тьюринга и рекурсивные функции. М.: Мир, 1972.
13. Касаткин В.Н. Информация, алгоритмы, ЭВМ. М.: Просвещение, 1991. – 192 с.
14. Кузнецов В.Е. Представление в ЭВМ неформальных процедур: продукционные системы. М.: Наука. – 1989. – 160 с.
15. Хопкрофт Д., Мотвани, Раджив, Ульман, Джеффри, Д.. Введение в теорию автоматов, языков и вычислений, 2-е изд.. : Пер. с англ. – М. : Издательский дом «Вильямс», 2008. – 528 с.
16. Миков А.Ю., Файнштейн С.И. Алгоритмы на сетях и графах: учебное пособие. – 2016. – 48 с.
17. Скиена С. Алгоритмы. Руководство по разработке. – СПб.: БХВ-Петербург,2016. – 720 с.
18. Клейнберг Дж., Тардос Е. Алгоритмы: разработка и применение. Классика Computers Science / Пер. с англ. Е. Матвеева. – СПб.: Питер, 2016. – 800 с.
19. Дж. Макконнелл. Анализ алгоритмов. Активный обучающий подход. – М.:Техносфера, 2013. – 416 с.

Учебное текстовое электронное издание

Миков Анатолий Юрьевич
Сибилева Наталья Сергеевна

АЛГОРИТМЫ И ТЕОРИЯ СЛОЖНОСТИ

Практикум

0,87 Мб
1 электрон. опт. диск

г. Магнитогорск, 2017 год
ФГБОУ ВО «МГТУ им. Г.И. Носова»
Адрес: 455000, Россия, Челябинская область, г. Магнитогорск,
пр. Ленина 38

ФГБОУ ВО «Магнитогорский государственный
технический университет им. Г.И. Носова»
Кафедра вычислительной техники и программирования
Центр электронных образовательных ресурсов и
дистанционных образовательных технологий
e-mail: ceor_dot@mail.ru